Package Development Guide

Magic Cap 3.0 (Rosemary Release)

General Magic

License

Your use of the software discussed in this document shall be permitted only pursuant to the terms in a software license between you and General Magic.

Trademarks

The General Magic logo, the Magic Cap logo, the Telescript logo, Magic Cap, Telescript, and the A rabbit-from-a-hat logo are trademarks of General Magic, and may be registered in certain jurisdictions. All other trademarks and service marks are the property of their respective owners.

Limit of Liability/Disclaimer of Warranty

THIS BOOK IS SOLD "AS IS." Even though General Magic has reviewed this book in detail, GENERAL MAGIC MAKES NO REPRESENTATION OR WARRANTIES, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THE ACCURACY OR COMPLETENESS OF THE CONTENTS OF THIS BOOK. GENERAL MAGIC SPECIFICALLY DISCLAIMS ANY IMPLIED WARRANTIES OR MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE AND SHALL IN NO EVENT BE LIABLE FOR ANY LOSS OF PROFIT OR ANY OTHER COMMERCIAL DAMAGE, INCLUDING BUT NOT LIMITED TO SPECIAL, INCIDENTAL, CONSEQUENTIAL, OR OTHER DAMAGES, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. Some states do not allow for the exclusion or limitation of implied warranties or incidental or consequential damage. So, the exclusions in this paragraph might not apply to you. This warranty gives you specific legal rights. You may also have other rights which vary from state to state.

It's 4:30 a.m. Do you know where your children are?

Restricted Rights. For defense agencies: Use, duplication, or disclosure is subject to the restrictions set forth in subparagraph (c)(1)(ii) of DFAR section 252.227-7013 and its successors. For civilian agencies: Use, duplication, or disclosure is subject to the restrictions set forth in subparagraphs (a) through (d) of FAR section 52.227-19 and its successors. Unpublished—rights reserved under the copyright laws of the United States.

Patent Pending

Portions of the Magic Cap software and the Telescript software are patent pending in the United States and other countries.

General Magic

420 N. Mary Ave.
Sunnyvale, California 94086
USA

Telephone:	408 774 4000
Fax:	408 774 4030
E-mail:	dev-info@genmagic.com
URL:	http://www.genmagic.com/

Table of Contents

1

Introduction to Objects

This chapter provides a brief overview of some fundamental Magic Cap concepts, including objects, classes, and operations. This overview is suitable for all Magic Cap programmers.

You should read this chapter as you start programming for Magic Cap. You will likely find that you don't remember everything in this chapter on first reading. Later, as you read the rest of this book and begin programming, you can refer back here for fundamental definitions and information.

Objects

Magic Cap provides an environment for supporting persistent objects. An **object** consists of a structure that includes data elements and a set of actions that operate on the data. Each object provides interfaces to its actions via **operations** and to its data elements via **attributes**. Internally, objects specify functions that implement operations and storage locations that implement attributes. These internal functions are called **methods** and the storage locations are called **fields**.

Objects are described by templates called **classes**. Each class is defined in terms of its difference from another class, called its **superclass**. Each class inherits operations and attributes from its superclasses, and the new class can redefine any operation to enhance or change its behavior. All classes that define objects ultimately descend from class `Object`.

When you call an operation, the method that executes in response to the call is defined by that object's class or by one of its superclasses. When an operation is called, Magic Cap selects the method to run by examining the object's class and, if necessary, its superclasses. The object that has its operation called is the **responder**.

Software Packages

Magic Cap includes several major sets of features, such as the Datebook, Notebook and Name card file, that provide services usually handled by application programs on conventional computer systems. In addition to its built-in features, Magic Cap acts as a platform for additional features to be added. When you program in Magic Cap, you make **software packages**, collections of objects that perform functions and provide features for Magic Cap users. Software packages are often called simply **packages**.

Whenever you deliver any Magic Cap software to users, you'll provide a software package.

Objects in Memory

Personal communicators based on Magic Cap are designed to keep permanent user data primarily in RAM rather than on an external storage device such as a hard disk. This RAM is called **persistent** memory because it retains its contents whether main power is turned on or off, as long as some power source is applied (main battery, backup battery, or AC power). Persistent memory is useful for permanent user data, such as electronic mail messages, name cards, appointments, and so on.

Sometimes, Magic Cap will encounter an error that cannot be fixed. When this happens, Magic Cap will **reset** itself to attempt to return to a stable state. Magic Cap defines another kind of memory that does not retain its contents when Magic Cap resets. This kind of memory, called **transient** memory, is useful for objects that are created and destroyed within the scope of an executing function, objects that can be recreated from persistent data, and other temporary objects. When you create new objects, you can specify whether they should exist in persistent or transient memory. If you create an object in transient memory, your package must be prepared to recreate it if Magic Cap resets.

All Magic Cap objects in memory, whether in main RAM, ROM, or on a PCMCIA card, are collected into groups called **clusters**. Each object belongs to exactly one cluster. Like most structures in Magic Cap, clusters are themselves objects. Each cluster in RAM controls only one kind of memory, either persistent or transient.

How Objects are Addressed

Every object in Magic Cap can be addressed at runtime with a 32-bit value called a **reference**. You use an object's reference when you call one of its operations, get access to its fields, or pass it as a parameter. When you use a reference, Magic Cap's object runtime quickly and efficiently converts the reference into a pointer to the object.

When you define objects at build time, you don't use references. Instead, you use **instance tags**, symbolic names that should represent what the object will be used for. These objects are assigned reference values automatically at runtime.

Your package can use references to address any object. You will typically see references as hexadecimal numbers displayed by debugging tools.

Indexicals

Magic Cap has a collection of well-known objects that you can address at build time and at runtime via special references called **system indexicals**. Magic Cap defines indexicals for hundreds of different system objects, such as images, windows, object lists, scenes, and many more kinds of objects. You can create **package indexicals** that define key objects in your own package.

Extra Data

In addition to its fields, every object can have one variable-length area, called its **extra data**. The extra data can be any information that can change its length at runtime. For example, Magic Cap defines many classes that include lists of references. These lists are usually kept in the objects' extra data, because the size of the extra data can vary as the list adds or removes members.

Although every object has only one extra data area, it can be shared among classes in a class hierarchy as long as the first class in a hierarchy that declares that it will use the extra data area also defines that it can share the extra data area with subclasses. For more information about how to use the extra data area of an object, see the section on the object definition syntax in the Object Tools chapter of *Guide to Magic Cap Development Tools*.

2

Object Runtime

This chapter discusses the Magic Cap object runtime. You should already be familiar with object oriented design concepts such as classes, methods, inheritance and overriding methods. Additionally, you should also be familiar with the tools that are used to create Magic Cap packages. If you are not yet familiar with these tools, see *Guide to Magic Cap Development Tools*.

Magic Cap provides an environment for supporting persistent objects. An **object** is a structure that includes data elements and a set of actions that operate on the data. Each object provides interfaces to its actions via **operations** and to its data elements via **attributes**. Internally, objects specify functions that implement operations and storage locations that implement attributes. These internal functions are called **methods** and the storage locations are called **fields**.

Objects are described by templates called **classes**. Each class is defined in terms of its difference from another class, called its **superclass**. Each class inherits operations and attributes from its superclasses, and the new class can redefine any operation to enhance or change its behavior. All classes that define objects ultimately descend from class `Object`.

When you call an operation, the method that executes in response to the call is defined by that object's class or by one of its superclasses. When an operation is called, Magic Cap selects the method to run by examining the object's class and, if necessary, its superclasses. The object that has its operation called is the **responder**.

Memory and Clusters

Personal communicators based on Magic Cap are designed to keep permanent user data primarily in RAM rather than on an external storage device such as a hard disk. This RAM is called **persistent** memory because it retains its contents whether main

power is turned on or off, as long as some power source is applied (main battery, backup battery, or AC power). Persistent memory is useful for permanent user data, such as electronic mail messages, name cards, appointments, and so on.

To help ensure that this persistent data isn't damaged, it is protected from change through a set of internal routines provided by Magic Cap. Packages can't write directly to persistent memory. Instead, packages write changes to a buffer area of memory. Magic Cap updates persistent memory from this buffer area periodically, ensuring that packages don't corrupt structures in persistent memory. Because persistent memory is protected from direct change, writing to it is somewhat slower than if the memory were not protected.

As a performance enhancement, Magic Cap defines another kind of RAM that is not protected from change and does not retain its contents when Magic Cap resets. This kind of memory, called **transient** memory, is useful for objects that are created and destroyed within the scope of an executing function, objects that can be recreated from persistent data, and other temporary objects. When you create new objects, you can specify whether they should exist in persistent or transient memory. If you create an object in transient memory, your package must be prepared to recreate it if Magic Cap resets.

Clusters

All objects in memory, whether in RAM, ROM, or on a PCMCIA card (which Magic Cap calls **storage cards**), are collected into groups called **clusters**. Clusters group objects belonging to a particular application and can enforce memory policies - a cluster can have a size limit, all objects in a cluster can be erased, a cluster may be considered transient, etc. Every object belongs to exactly one cluster. Like most structures in Magic Cap, clusters are themselves objects. Each cluster in RAM controls only one kind of memory, either persistent or transient.

Each object in a cluster occupies a contiguous area of memory. As new objects are created and others are destroyed, the memory in a cluster can become fragmented, with many free spaces occurring between the objects. Because each object requires a contiguous space in memory, a fragmented cluster effectively reduces the amount of available memory. To relieve this problem, Magic Cap moves objects at reasonable times to close up the gaps in free space.

Objects in clusters are tracked by special pointers called **locators**. When an object is relocated, the locator to that object is updated to reflect the object's new location. Magic Cap accesses objects by converting references to locators when they are used.

Magic Cap includes two clusters created by the system, called the **system persistent cluster** and the **system transient cluster**. Every package can directly address objects in the system clusters. In addition to these system clusters, when you create a new package, your package includes its own persistent cluster and transient cluster. Packages can also address objects in clusters belonging to other packages. Some Magic Cap implementations have no transient RAM. On those implementations, all clusters are persistent.

Shadow Clusters

Magic Cap stores many objects in ROM that users must be able to customize and change. To allow Magic Cap and software packages to change these objects, the object runtime can store changed versions of ROM-based objects in a special RAM cluster that is associated with the ROM cluster. This RAM cluster that holds changed versions of objects is called a **shadow cluster**.

Every ROM cluster that allows changes to its objects has a shadow cluster associated with it. When Magic Cap or a package makes changes to an object stored in ROM, the object runtime modifies a copy of the object in the shadow cluster instead. If the object is changed again, the copy in the shadow cluster is modified each time the object is changed.

Shadow clusters are typically kept in transient RAM. Because the contents of transient RAM are lost when Magic Cap needs to reset, the object runtime periodically preserves the objects in the shadow cluster by transferring them to another shadow cluster in persistent RAM. This process of transferring objects from the transient shadow cluster to the persistent shadow cluster is called **committing** the changed objects. The objects themselves are called **committed objects**. In contrast, objects in the transient cluster are called **uncommitted objects**.

The object runtime commits changes at various times. In particular, changes are committed whenever the user goes to a new scene and whenever the communicator shuts off. If the communicator unexpectedly loses power, uncommitted changes in the current scene are lost.

Note that the system objects stored in ROM actually may be found in any of three clusters: the system source cluster in ROM; the system shadow cluster in RAM, which contains committed changes to the ROM; and the system changes cluster in RAM, which contains uncommitted changes to the ROM. These three clusters form a chain.

When you address an object, the object runtime determines if the object you're addressing exists in a cluster that has one or more shadow clusters. If so, the cluster with uncommitted changes has the most recent versions of objects, followed by the committed changes, and finally the source. The object runtime automatically finds the current version of the object from among these clusters.

The object runtime has two three-cluster chains: the system persistent chain and the package persistent chain. In each case, the first cluster in the chain contains uncommitted changes, the middle cluster has committed changes, and the last cluster holds the original versions of the objects.

Note that the same reference is used for all versions of the object, so it is impossible to directly address one particular version. The reference always addresses the current value of the object.

References

Every object in Magic Cap can be addressed at runtime with a 32-bit value called a **reference**. You use an object's reference when you execute one of its operations, get access to its fields, or pass it as a parameter. When you use a reference, the object runtime quickly and efficiently converts the reference into a pointer to the object.

When you define objects at build time, you don't use references. Instead, you use symbolic names that represent what the object will be used for. These objects are assigned reference values automatically at runtime. See "Creating Objects at Build Time" on page 10 for more information.

Your package can use a reference to address any object. Magic Cap defines several kinds of references that can be used for various special purposes. For more information on references, see "Addressing Objects" on page 14.

You will typically see references as hexadecimal numbers displayed in the Inspector and in debuggers.

Object Relationships

Object fields can contain references to other objects. Normally, when you copy an object, any objects it refers to are also copied. Similarly, when you destroy an object, any objects it refers to are also destroyed. This is known as a **strong reference** relationship. An object that strongly refers to another object is said to own that object.

When you define a class, you can specify that fields will have a **weak reference** to other objects. When you copy an object, objects it weakly refers to are not copied; the same fields in the object copy will also weakly refer to the same objects as the original. When you destroy an object, objects it weakly refers to are not destroyed. An object that weakly refers to another object does not own that object.

Special Objects

Some Magic Cap objects have special properties that help to make more efficient use of memory, and make writing Magic Cap packages easier. This section discusses these special types of objects and their relationships with other objects.

Shared Objects

There are certain types of objects, such as images and sounds, that tend to be very large. Because of their size, having multiple copies of these objects in memory would be wasteful. To reduce the amount of memory these types of objects take up, they are usually **shared**.

Normally, when you copy an object, any objects it strongly refers to are also copied. When an object that strongly refers to a shared object is copied, the shared object is not copied. A shared object maintains a **reference count** to keep track of how many

other objects refer to it. When an object that strongly refers to a shared object is copied, instead of making a copy of the shared object, the shared object's reference count is increased by one.

A shared object can only be referenced by objects in the same cluster the shared object is in. When a shared object, or an object that refers to a shared object, is copied into a different cluster, a new shared object is created in that cluster. See "Creating Objects at Runtime" on page 11 for more information on copying objects.

Normally, when you destroy an object, any objects it strongly refers to are also destroyed. When an object that strongly refers to a shared object is destroyed, the shared object's reference count is decreased by one. If the reference count remains greater than zero, that means that other objects still contain references to this shared object, so the shared object is not destroyed. If the reference count reaches zero, no other objects refer to the shared object, so the shared object is destroyed along with the other object. See "Destroying Objects" on page 13 for more information on destroying objects.

Because many objects can refer to and rely on a single shared object, you should not modify the data contained in a shared object. If you need to make changes to a shared object, you should make a non-shared copy of the desired object to work with. This is done by calling the `MakeModifiableNear` operation. `MakeModifiableNear` takes two parameters, a shared object and an example object. `MakeModifiableNear` creates a non-shared copy of a shared object in the cluster the example object is located in. The reference returned by `MakeModifiableNear` will address the non-shared copy.

After modifying the copy of the shared object, it is desirable to make the copy a shared object again, so that if it is subsequently copied, no additional copies of the object will be made. This is done by calling the `MakeSharedNear` operation. `MakeSharedNear` takes two parameters, a non-shared object and an example object. `MakeSharedNear` looks in the cluster the example object is located in for a shared object that exactly matches the non-shared object. If such a shared object exists, its reference count is increased by one, and its reference is returned. In this case, the non-shared object is destroyed, and references to it are no longer valid after `MakeSharedNear` returns. If no shared object in the cluster is identical to the non-shared object, the non-shared object is turned into a shared object.

Shared objects can reference other objects. These objects are said to be **owned by shared**. These objects are considered part of the shared object and should not be modified either.

Ephemeral Objects

Because objects aren't stored on a stack as conventional variables are, objects don't follow the usual storage-class rules of procedural programming languages. Objects that you create while a function is executing are never automatically destroyed when the function ends, unlike automatic (local) variables in C. You must normally destroy them yourself. The exception to this rule is the **ephemeral object**.

Ephemeral objects are extremely short-lived objects. In some respects, ephemeral objects are similar to static variables in C. Static variables are only valid in certain program scopes. Similarly, ephemeral objects only exist within a particular chain of function invocations. Once the top of that call chain is reached, Magic Cap will reclaim any ephemeral objects created in that chain.

Because ephemeral objects are automatically reclaimed by Magic Cap, you never explicitly destroy ephemeral objects. For the same reason, you are not allowed to store references to ephemeral objects in other objects. If you want to save the data represented in the ephemeral object, you must convert it into a persistent or transient object. This is done by calling the `MoveNear` operation. `MoveNear` takes two parameters, a responder object and an example object. `MoveNear` will move the responder object into the cluster the example object is located in.

You cannot create ephemeral objects directly in your Magic Cap package, but Magic Cap itself will always use ephemeral objects to return text to your program.

Creating and Destroying Objects

This section discusses creating objects, both at build time and at runtime, and destroying objects in a running Magic Cap package.

Magic Cap provides two fundamentally different ways to create objects: you can specify them in an instance definition file, or you can create them programmatically at runtime. Objects specified in an instance definition file are created when the package is built and are loaded along with the package. You can create objects programmatically at runtime by calling various operations defined by Magic Cap.

Objects created at build time by defining them in an instance definition file are said to be created **statically**. Objects created programmatically at runtime are said to be created **dynamically**.

Creating Objects at Build Time

To create an object statically, you define the object at build time using the name of its class and a symbolic **instance tag**, along with values for its fields, in an **instance definition file** (an .odef file). This is an example of an object specified in an instance definition file:

```
Instance LyricText theyMightBeGiantsVerse;
   textValue: ‘Everything right is wrong again.’;
End Instance;
```

This example specifies a new object of class `LyricText` that will be created in the package. The first line of the instance definition specifies the class of the object, followed by the object’s instance tag. The instance tag is a symbolic label that uniquely identifies the instance within the instance definition file’s namespace. Instances have tags because instance definitions often refer to other definitions in the file. For example, the definition of a list object contains the instance tags of other objects, as follows:

```
Instance SongSet songs;
   firstSong: (Song particleMan);
```

```
  middleSong: (Song instanbul);
      finale: (Song puppetHead);
End Instance;
```

Objects created at build time are installed in the package's persistent cluster.

Instance definitions must include values for every field of the objects they specify. Because the definition above for an object of class `LyricText` includes a value for one field, *textValue*, we can assume that class `LyricText` defines only that one field for its objects. Similarly, the definition for class `SongSet` should show that class `SongSet` defines three fields for its objects.

Objects are uniquely identified by tags. In addition to these required tags, each object can optionally have a text value called an **object name**.

Each class specifies how its objects handle their names. Some viewable classes display their objects' names to users. For example, the text on a button is the button's object name. Other classes don't display their objects' names to users. Names for these objects are useful only for programming and debugging.

Unlike an instance tag, an object name need not be unique. More than one object in an instance definition file or a running Magic Cap system can have the same object name. Because of this, you can't rely on just an object name, or even a class and object name, to uniquely identify an object in your package or in Magic Cap.

You can name an object in an instance definition by including the text of the name after the class name and the instance tag, like this:

```
Instance SongSet songs 'Trios';
   firstSong: (Song particleMan);
  middleSong: (Song instanbul);
      finale: (Song puppetHead);
End Instance;

Instance Song instanbul 'Not Constantinople';
    songData: $ 4445 4178;
End Instance;
```

The song set and song objects defined by this example have object names, 'Trios' and 'Not Constantinople'. Note that the object name is only used in the instance definition itself; it is not used when an instance with an object name is referenced from another instance definition.

The names and types of fields for each class are specified in class definition files. For more information on the format of instance definitions, see the chapter "Object Tools" in *Guide to Magic Cap Development Tools.*

Creating Objects at Runtime

Magic Cap provides two ways to make objects dynamically at runtime: you can create a copy of an existing object, or you can make a new, empty object. Magic Cap defines several operations for copying objects and several for creating new objects. You'll usually make new objects by copying existing ones because the operations that copy an object also copy the objects it strongly refers to in its fields, which ensures

that the new copy is fully functional. For example, when you copy a viewable object that includes another viewable object contained inside it, the contained object will also be copied.

The operations that copy objects and create new objects are defined by class `Object`. Because all classes in Magic Cap that create objects include class `Object` as an ancestor, you can use these calls to create any object you want.

You can use the `NewNear` and `NewTransient` operations to create new objects. Call `NewTransient` to create a new object in the package transient cluster.

When you call `NewNear`, you'll pass an example object. `NewNear` will then create a new object in the same cluster as the example object. For example, if you call `NewNear` with an example object in the package persistent cluster, the new object will also be in the package persistent cluster.

When you call `NewNear` or `NewTransient` to create a new object, Magic Cap creates the new object in memory, then calls the new object's `Init` operation to give the object a chance to set up its fields.

Often, you'll create a new object by copying an existing one using the `CopyNear` or `CopyTransient` operations. These operations are analogous to the New operations described above. Their names indicate where the newly copied object will be created. These operations also copy any objects that the copied object strongly refers to. These operations will not copy objects that the copied object has weak references to.

When you call `CopyNear` or `CopyTransient` to create a new object, Magic Cap creates the new object in memory, then calls the new object's `Copying` operation to give the object a chance to set up its fields. Because these operations also copy any objects referred to in the copied object's fields, Magic Cap calls the `Copying` operation of each newly copied object.

When you create objects dynamically with a New or Copy operation, the operation you call will return the reference of the newly created object. You can use this reference to address the object.

Creating New Objects on Storage Cards

A user can specify that new objects should be created using the memory on a storage card instead of using Built-in memory. This is known as **new items go here**. Objects created in storage card memory are located in a cluster called the **new items package**. To make your Magic Cap program respect the user's preference for where new objects should be created, you can pass the indexical `iNewItemsGoHere` as the example object parameter for `NewNear`, `CopyNear`, or `MoveNear`.

When you use `iNewItemsGoHere` as the example object, but the new items package does not exist for whatever reason, Magic Cap will create new objects in the system persistent cluster. In this instance, if you would rather have objects you create reside in your package's persistent cluster, you can call the `NewItemNear` operation, and use the reference that is returned as the example object for `NewNear`, `CopyNear`, or `MoveNear`. `NewItemNear` takes an example object as parameter `self`. If the new

items package exists, `NewItemNear` returns `iNewItemsGoHere`. If the new items package does not exist, `NewItemNear` will return the cluster the example object is located in.

Not all objects should be explicitly created in the new items package. Instead, only objects that can be filed by the user, such as cards and datebook tasks, should be created in the new items package. Objects that are attached to these objects should then be created near the fileable object to be created in the correct place.

Here is an example of how you would create a card object with attachments and respect the user's preference for where new items should be created:

```
Method void
ExampleStackScene_CreateNewCardWithAttachments(Reference self,
                  Reference nearThis, Reference attachments)
{
   Reference newCard;

   // Create a new card object in the new items package or in this package.
   newCard = CopyNear(ProtoCard(self), NewItemNear(nearThis));

   // Copy the attachments into the same cluster as the new card.
   AddAttachments(newCard, CopyNear(attachments, newCard));
}
```

Destroying Objects

Because objects aren't stored on a stack as conventional variables are, objects don't follow the usual storage-class rules of procedural programming languages. Objects that you create while a function is executing are never automatically destroyed when the function ends, unlike automatic (local) variables in C. You must destroy them yourself. Any objects you create in a transient cluster might be destroyed at any time.

You can call `Destroy` to destroy an object explicitly and free its storage. Because `Destroy` is defined by class `Object`, you can use it with all Magic Cap objects. When you call `Destroy`, Magic Cap also destroys any objects that the given object strongly refers to in its fields. Although Magic Cap provides a garbage collection system to delete unused objects, you must make sure that you destroy objects that you create and no longer need.

When you call `Destroy` to destroy an object, Magic Cap calls the given object's `Finalize` operation to give the object a chance to perform some action before it is deleted. Because `Destroy` also destroys any objects strongly referred to in the given object's fields, Magic Cap calls the `Finalize` operation of each referenced object before it is destroyed.

When an object is about to be destroyed, Magic Cap calls `Finalize` on the given object first, then on any objects it strongly refers to, and then destroys the given object and the objects it strongly refers to. You can use the `Finalize` call to save referenced objects from being destroyed by setting the appropriate field to

`nilObject` in your `Finalize` code. Then, when the given object is destroyed, the object you wanted to save will be left alone because it is no longer referenced by the destroyed object.

Note: Indexicals are treated specially. When you pass an indexical to `Destroy`, the object that the indexical references is not destroyed.

Using Statically Created Objects

Instance tags are only used in instance definition files. You can't refer to them from your source code. Because of this, you must take special steps to address statically created objects in source code. Most of the objects that you create statically should be designed to be self-sufficient; that is, you shouldn't have to refer to them from outside their own operations. Every operation can refer to its responder.

Your package may have a few special objects that you want to create statically, but still address with operations other than the object's own. To handle these kinds of objects, Magic Cap defines a form of global addressing that you can use to address your package's statically defined objects from your source code. This form of addressing is known as an **indexical**. Indexicals defined by packages are known as **package indexicals**. You can specify the static objects that you want to address from your source code with package indexicals. To create an indexical, it must be defined in your instance definition file and then declared in your class definition file. To define an indexical, you provide a symbolic name for the indexical and the static object it references. Following is an example of how you define an indexical:

```
indexical iMilkyWay = (Galaxy milkyWay);
```

To declare the package indexical, you provide the same name you used in the instance definition file and the class of the object the indexical refers to. Here is an example of how you declare a package indexical:

```
indexical iMilkyWay: Galaxy;
```

After setting up the definition and declaration of an indexical, you can use the symbolic name in your source files as if it a were a reference to a dynamically created object.

Because indexicals are similar to global variables, you should use them sparingly, as they have many of the same drawbacks as global variables. Magic Cap defines many system indexicals that provide access to well-known system objects. See "Indexicals" on page 16 for more information on indexicals.

Addressing Objects

This section discusses the ways you can address objects in a Magic Cap package. Magic Cap provides various techniques for addressing objects both at build time in instance definition files and at runtime in source code.

Addressing Objects at Build Time

Objects defined in instance definition files often refer to other objects. Magic Cap provides two ways for objects to refer to others at build time in instance definition files: instance tags and indexicals.

Package objects in an instance definition file can address other objects directly by using instance tags, as shown in "Creating Objects at Build Time" on page 10. This form of address only works for statically created objects in the same package. The instance definition file's syntax for this form of reference consists of placing the referenced objects' instance definition in parentheses when defining a field, like this:

```
// start of instance definition here
        firstSong: (Song 239);     // another static object
// rest of instance definition here
```

Package objects in an instance definition file can refer to well-known objects in the system or the package itself by using indexicals. You can have your package address an indexical by simply writing the indexical's symbolic name as the field's value, like this:

```
// start of instance definition here
        favoriteSound: iNewYearSong;     // indexical reference
                                         // to system object
// rest of instance definition here
```

Symbols for system indexicals, which are defined automatically when you build your package, always start with the letter *i*, as with the example `iNewYearSong` shown above. You should use names starting with *ip* as symbols for your package indexicals. The addition of *p* helps avoid potential name conflicts with system indexicals.

Addressing Objects at Runtime

Magic Cap provides various ways for you to address objects from your package's source code by using various kinds of **component numbers**. The kinds of component numbers include indexicals, references, class numbers, intrinsic numbers, class operation numbers, and operation numbers.

References

A **reference** is used to address any object in memory. The object can live in RAM, ROM, or on a storage card. A reference that addresses an object in ROM is called a **ROM reference**. A reference that addresses an object in RAM is called a **RAM reference**.

When you create a new object by calling one of the New or Copy operations, the operation will return a reference to the new object.

The object runtime defines a special reference that represents the absence of an object. This reference is defined by the symbol `nilObject`.

Indexicals

Indexicals provide a way to address well-known objects in the system, your package, and other packages. You can address indexicals defined by Magic Cap itself, or you can use indexicals to address important objects defined by other packages.

Indexicals are an indirect form of a reference. When you use an indexical, the object runtime maps the indexical to the appropriate reference, then converts the reference to a pointer to access the object.

Because indexicals use an extra level of indirection, they are somewhat less efficient than direct references, although this inefficiency isn't significant for most uses. If you use an indexical repeatedly, as in a tight loop, you can improve performance by using a direct reference instead of the indexical. The object runtime provides the `DirectID` operation, which converts an indexical to a direct reference. You can then use the direct reference for faster access to the object.

`DirectID` always returns the most efficient, most direct reference for an object, and it's safe to call on all kinds of references. If the reference can't be made more direct, `DirectID` simply returns the given reference unchanged. This happens, for example, if you call `DirectID` on a reference that is already a direct reference.

You should always call `DirectID` before comparing two references, except in the rare case when you can guarantee that both references are not indexicals. You don't ever have to call `DirectID` on the responder inside its own operation. The object runtime ensures that the responder is a direct reference inside its operations.

Magic Cap defines hundreds of system indexicals for addressing objects that include images, songs, text styles, windows, gadgets, and virtually all interesting system objects. Magic Cap defines symbolic names for most of these objects so that you can use indexicals to address them from your instance definition files and source code.

In addition to indexicals that address unchanging objects, you can use indexicals with values that change dynamically depending on the state of Magic Cap. These **current indexicals** let you address such dynamic objects as the current user, the scene that is visible on the screen, or the user's default stationery for new electronic mail messages.

If you want to capture the value of a current indexical, you can call `DirectID` on the indexical. `DirectID` will return a reference to the object that represents the current value of the indexical at the time you make the call.

You can find a complete list of system indexicals, including current indexicals, in the file Indexicals.cdef.

In addition to system indexicals, you can define package indexicals. Package indexicals are useful in two ways. First, you can define objects as indexicals in instance definition files, then address them from your source code. Second, objects that you define as package indexicals can be easily addressed by other packages.

Using Indexicals

Indexicals are primarily used to address well-known system objects and objects in other packages. You can use indexicals in all the ways you use other references. Indexicals are most commonly used as values in objects' fields and as parameters to operation calls.

Magic Cap uses system indexicals to define a set of **prototypes**, example objects that you can copy and use. These prototypes are blank or empty versions of objects that are useful in your packages. Because these prototype objects are often fairly complex, copying one by using an indexical provides a quick way to create a whole set of objects that work together.

For example, the indexical `iPrototypeNameCard` represents a new name card for a person and all its associated objects. You can call one of the Copy operations on this indexical to create a new name card for a person. The new card comes complete with labels for home and work phones, home and work addresses, and work fax. The name on the card is new person, and the image is the standard image for a person. Note that all these items on the new card are the same ones that appear when the user creates a new name card with the new button in the Name card file. Also, note that by simply copying the prototype via its indexical, you've created a complex set of objects that represent a name card, including more than 10 different interrelated objects.

You can use package indexicals to create prototypes for objects that you use often in your packages. This technique is most effective for complex objects that refer to many other objects. You shouldn't use prototypes for very simple objects. Instead, just use one of the New operations to create the objects, then fill in the appropriate fields.

When you use an indexical, you can't assume that you know the type of the object that the indexical addresses. Any subclass of the expected type might be substituted instead. For example, if you use an indexical that refers to a window, the actual object addressed might be an instance of a subclass of class Window. In general, you should never assume the class of any object. Instead, you can check to see whether an object implements the interface of a given class. See *Guide to Magic Cap Development Tools* for more information about inheritance.

When you create your own package indexicals, remember that indexicals provide a level of indirection that you should consider when choosing what to name them. Try to choose names for your indexicals that refer to high-level architectural concepts, rather than specific implementation details.

Class Numbers

Class numbers are component numbers that provides a way to refer to classes directly. Some operations are invoked by passing a class number instead of a responder of that class. These operations are called **class operations**.

The object runtime defines a special class number that represents the absence of an class. This number is defined by the symbol `nilClass`.

Operations

Every class defines interfaces to its actions called **operations**. You can call these operations to perform actions on the class's objects. Some operations perform actions that do not affect a particular instance of the class. These are called class operations. The operations of a class are defined by declaring them in the class's definition in a class definition file. Operations are identified at runtime by dynamically assigned **operation numbers**. The object runtime defines a special operation number that represents the absence of an operation. This number is defined by the symbol `nilOperation`.

Many objects contain values that can be set and changed. For example, every viewable object includes a Boolean value that indicates whether it can be moved by the user and a pair of 32-bit values that indicate the object's height and width. You can specify values like this as **attributes** of the object in the class's definition.

Taken together, operations and attributes form the public interface of a class. Fields, which are often used to implement attributes, are not part of a class's public interface. They should be considered private and should only be used when defining objects in instance definition files and within code of operations of the class.

Defining Operations and Attributes

Operations for each class are defined in class definition files. In the class's definition, operations are defined with the keyword `operation`, as in the following examples:

```
operation StartSongs();
operation DrawWithContents(canvas: Canvas; clip: Path);
operation DirectID(): Object;
```

Each operation definition begins with the keyword operation, followed by the name of the operation, and the operation's parameters in parentheses. If the operation returns a value, the definition ends with a colon and the type of the return value. All operation definitions end with a semicolon, like all lines in an class compiler input file. For historical reasons, the syntax for defining operations and parameters in class definitions is similar to that of Pascal.

You can define class operations much like you define normal operations, except that you use the keyword `class operation`.

You can define an attribute if you want to specify a value that can be read and set, as in the following examples:

```
attribute Last: Object;
attribute CanExtendBottom: Boolean;
attribute ContentHeight: Micron;
```

When you define an attribute in a class definition, you implicitly define two operations, one to get the value and another to set it, called a **getter** and a **setter**. Attributes simplify class definitions by grouping the getter and setter operations. Because each attribute defines two operations, attributes are part of the class's interface. Attribute definitions are simpler than operation definitions because attributes have no parameters.

The getter has the same name as the attribute. The setter prefixes "Set" to the attribute name. The getter takes no parameters other than the responder, and returns the value of the requested attribute. The setter takes the new value as a parameter, and returns no value.

The getter and setter operations created implicitly by the last attribute definition shown above work as if they were defined as follows:

```
operation ContentHeight(): Micron;             // getter
operation SetContentHeight(newValue: Micron);  // setter
```

The class compiler defines various keywords that can be used optionally to modify operation and attribute definitions. See *Guide to Magic Cap Development Tools* for complete information on the syntax of operation definitions.

Implementing Attributes

Attributes provide an interface to their getter and setter operations, but you must provide an implementation for these operations. Because attributes often use fields to hold their values, the class compiler lets you define **automatic field accessors** as a way to connect attributes to fields automatically, without having to write any getter or setter methods yourself.

An automatic field accessor that reads the value of a field is called an automatic getter, and an automatic field accessor that sets the value is called an automatic setter. When you use automatic field accessors, you don't have to write any code to implement the attribute's operations - the code is created for you.

If you don't use automatic field accessors to implement the operations defined by an attribute, you must one of the other techniques for implementing operations: C functions or scripted functions.

See this chapter's Automatic Field Accessors section for complete information about creating automatic field accessors, including syntax for class definition files.

Calling Operations

You ask a class to perform an action on one of its objects by calling an operation defined by the class or one of its superclasses. You call an operation by making an ordinary function call with the same name as the operation, passing the object's reference as the first parameter. For example, every viewable object includes a sound, and you can play the sound by calling the viewable's `PlaySound` operation. Assuming that `viewObject` is defined as reference, you can call its `PlaySound` operation as follows:

```
PlaySound(viewObject);
```

`PlaySound` is the name of the operation as defined by class `Viewable`. In this example, `viewObject` is the responder.

You call class operations in a similar fashion. Instead of passing a reference as the first parameter, you would pass the symbolic label that represents the class number. This label will always be the class name followed by an underscore:

```
RunNext(Scheduler_);
```

`RunNext` is the name of the class operation as defined by class `Scheduler`. You can pass the class number for any class that is a subclass of the class that defines the class operation.

Dispatching

When an operation is called, the object runtime performs an action in response. The object runtime determines what action to take by examining the class of the responder through a process called **dispatching**, handled by a part of the object runtime called the **dispatcher**. The dispatcher first checks to see if the responder's class implements the operation. If so, the dispatcher executes that class's implementation of the operation.

If the responder's class doesn't implement the operation, the dispatcher searches through the class's ancestors, beginning with the class's immediate superclass. When the dispatcher finds a class in the class's inheritance hierarchy that implements the operation, the object runtime executes that class's implementation. If no class implements the operation, the object runtime generates a **method not found** error, which activates the debugger in debug versions of Magic Cap, or continues silently in non-debug versions. You can use the `Implements` operation before calling an operation to determine if an object implements a particular operation.

You can see the order that the dispatcher will use to search for a class's operation by using Magic Cap's inspector to look at an object of the class. The inspector displays the object's inherited classes in reverse of the order that the dispatcher searches.

Intrinsics

Classes can define operations that don't require an instance of the class in order to execute. These operations are called simple intrinsics. Simple intrinsics are faster to invoke than other operations but can't be overridden by subclasses. Utility operations and other global functions are often implemented as **simple intrinsics**. Simple intrinsics do not take a responder as a parameter.

For example, operation `CalcCRC32`, which calculates a checksum value, is implemented as a simple intrinsic. When you call `CalcCRC32`, you don't pass a responder as a parameter.

Like operations, intrinsics are identified at runtime by dynamically assigned intrinsic numbers. The object runtime also defines a special intrinsic number that represents the absence of an intrinsic. This number is defined by the symbol `nilIntrinsic`.

Implementing Operations

When you create a class and operations, you can choose from among several ways to implement each operation that your class defines:

- Write a function in C.
- Create automatic functions for getting and setting attribute values.
- Write a script using Magic Script.

You use the same syntax to call an operation no matter which of these techniques is used to implement an operation. In fact, the choice for the operation's implementation isn't readily visible to callers.

The following sections discuss each of these techniques for implementing an operation.

C Functions

By far the most common way to implement an operation is to write a function in your C source. To do this, you define a C function that has as its name the class name and operation name connected with an underscore. For example, a C function named `TestClass_DoSomething` would provide the code for the `DoSomething` operation of class `TestClass`. When you use this technique to implement an operation, the code in your C function is called a **method**.

Following is an example of the `PlaySound` operation, shown above, implemented with a C function:

```
Method void
Viewable_PlaySound(Reference self)
{
   Reference sound = Sound(self);

   if (sound != nilObject)
      Play(sound);
}
```

To use a C function to define a method, start the function declaration with the identifier `Method`, a macro defined by the development environment, followed by the rest of the normal function declaration. The function name should be the class name and the operation name, as specified in the class definition file, connected with an underscore.

Remember that when you call an operation, you always pass the responder as the first parameter:

```
PlaySound(viewObject);
```

However, the responder is never shown when the method's operation is defined in the class definition file. The reason why the responder isn't shown in the definition files is simply to make the definitions easier to read and to type. The responder is always implied in the class definition file and must be used when you call an operation or declare a C function as a method.

For example, the operation `PlaySound` shown above would be defined as follows in a class definition file:

```
operation PlaySound();
```

Note that there's no mention of the responder in the definition, even though you must always pass it, as shown in the example call. For comparison, here's how the same call would appear in Object Pascal, a popular object-oriented language:

```
viewObject.PlaySound;   (* won't work in Magic Cap *)
```

In C++, the call would look like this:

```
viewObject.PlaySound()  // won't work in Magic Cap
```

The Magic Cap version in C is repeated here for comparison:

```
PlaySound(viewObject);  // works in Magic Cap
```

Note that Object Pascal and C++ don't require you to pass the responder explicitly. Instead, it is passed implicitly in every operation call. When writing Magic Cap programs, you must pass the responder explicitly.

This difference between the appearance of operations in C and class definition files can cause build-time errors that may be confusing. Because this point is so important, it's repeated and summarized in the following warning.

WARNING! When an operation is declared or called in C source, the first parameter must be the responder, the object whose operation is being called. When an operation is defined in a class definition file, the responder is implied but is never shown.

For example, this operation definition:

```
operation PlaySound();   // class compiler syntax
                         // the responder is implied, not shown
```

matches an operation declared in C like this:

```
Method void
Viewable_PlaySound(ObjectID self)   // C syntax
                                     // the responder is explicitly shown
```

Here's how you would call the same operation in C:

```
PlaySound(viewObject);           // C syntax
                                 // the responder must be included
```

When you're writing an implementation of an operation, you can use the first parameter to refer to the responder. By convention, this parameter is usually named `self`, but there's no requirement that it have that name.

Scripts

You can use a script written in Magic Script to provide the implementation for an operation. You can provide a Magic Script version of any operation. See *Guide to Magic Cap Development Tools* for more information on using scripts for operations.

Accessing Object Data

Every object in memory consists of a series of fields containing values. Objects include the fields defined by their classes and superclasses. The fields of objects are private. You should avoid reading or changing the field, except from operations defined by the field's class. There are two fundamental ways to read from or write to the fields of an object:

- Call an operation that reads from or writes to the field.
- Call an **accessor** operation that gives you direct access to the field.

Whenever possible, you should work with fields by calling operations that access the field for you. In cases where that isn't possible, or when you must work with the fields of your own objects, you'll use accessor operations defined by the object runtime to read and change fields.

A class's fields are defined along with the rest of a class in a class definition file. The definition for a class includes the `field` keyword, the name of the field, its type, and optional modifiers.

Each object's fields are organized into groups, with each group defined by one of the classes in the object's class ancestry. Not every class defines fields. For example, Magic Cap defines class `StatusAnnouncement`, which inherits from class `Announcement`, which in turn inherits from class `Object`. Every object of class `StatusAnnouncement` includes the following fields:

- fields inherited from class Object:
  (none)
- fields inherited from class Announcement:
  bootList, flags, info, stamp, sound, publicAddress
- fields defined by class StatusAnnouncement:
  currentScope, currentValue, serverToAbort

Every object can have one variable-length area, called the object's **extra data**. The extra data can be any information that can change its length at runtime. For example, Magic Cap defines many classes that include lists of references. These lists are usually kept in the objects' extra data, because the size of the extra data can vary as the list adds or removes members.

Extra data can be **structured** or **unstructured**. When extra data is structured, it can be treated as an array, with each entry being the same size and holding data of the same type. Many of Magic Cap's list classes, such as `ObjectList`, `IntegerList`, `ClassNumberList`, and `ShortIntegerList`, are implemented using structured extra data. When extra data is structured, the first word of an object's extra data space is used to describe the structure of the data. This allows Magic Cap to interpret the structured extra data without the need to call any operations defined by the class. Object references must be stored in structured extra data.

When extra data is unstructured, entries can be of variable size, and Magic Cap must rely on operations provided by the class to be able to interpret the extra data content of such classes. Card objects store the data in text fields on their forms as unstructured extra data.

Although every object has only one extra data area, it can be shared among classes in a class hierarchy as long as the first class in a hierarchy that declares it will use the extra data area also defines that it can share the extra data area with subclasses.

Objects can have both structured and unstructured extra data, but the structured data always comes first. If a class stores structured data in its objects, a subclass can also store structured data in the object as long as the subclass' structured data is the same size as the data stored by parent class. Additionally, if a class stores unstructured extra data in objects, no classes can inherit from that class and store structured extra data. For example, the Magic Cap class `Viewable` uses structured extra data to maintain a subview list in each viewable object. Each structured extra data entry is a reference to a viewable object. Class `Card`, which inherits from class `Viewable`, also uses structured extra data to store text style information for entered text. Each structured data entry is a reference to a text style object. Class `Card` also stores the

data from form items as unstructured data after the structured text style information. Subclasses of `Card` can store additional unstructured extra data if they wish to, but are not allowed to store any structured extra data.

For more information about how to use the extra data area of an object, see the section on the class definition syntax in the Object Tools chapter of *Guide to Magic Cap Development Tools*.

Automatic Field Accessors

You should avoid reading and writing fields directly, instead using operations wherever possible. To help enforce this behavior, you can ask the class compiler to create operations automatically that read and write a field associated with an attribute: automatic field accessors. You can define these operations for a field by using the following syntax in your class definition file:

```
// start of class definition here
field justAboutGlad: Boolean, getter, setter;
// other fields defined here
attribute JustAboutGlad: Boolean;
// rest of class definition here
```

The attribute declaration automatically specifies two operations. If you want to keep the values of this attribute in a field, you can create automatic field accessors (an automatic getter and setter), for the attribute. The automatic getter simply returns the value of the field. The automatic setter just sets the field to a value that you pass when you call it.

To create automatic field accessors, use the same name for the field and the attribute, capitalizing the attribute name.

You can use this technique to create an automatic setter and getter for any attribute and associated field that contain an object, unsigned, signed, or boolean value. When you create automatic field accessors, they can be overridden by subclasses just like other kinds of operations, and the overridden versions can be C functions.

Because dealing with text is a common operation, automatic field accessors that get and store text objects are handled specially. When you call an automatic getter to get the text object stored in a field, an ephemeral copy of the text object created and the reference to the copy is returned to you. When you pass a text object to an automatic setter, a copy of the text object is made and stored in the field, leaving your original text object untouched. This is done to avoid possible confusion about who owns the text object. With this design, the code that calls an automatic text getter is responsible for the returned text object while ownership of the original text object is retained by the object that refers to it. Similarly, the code that calls an automatic text setter is responsible for the text object passed to the setter while the object that stores the reference to the text object is responsible for making a copy for its own use. See "Ephemeral Objects" on page 9 for more information about ephemeral objects.

You should only create getters and setters for fields that must provide a public interface. Most fields can safely be kept private, and so don’t need getters and setters. You might want to create an interface for some fields that allows their values to be read, but not changed. To do this, use only the `getter` keyword in the field’s definition; only a getter operation will be created.

If the getter and setter routines you create will be returning and storing object references, you should consider following the same strategy used by the automatic text getter and automatic text setter to avoid possible confusion about object ownership.

Accessors

If no operation is defined to access a field, you can use accessor operations to read and write the field directly. The object runtime defines four families of accessor operations:

- **Single-field accessors** that read or write a single field of an object.
- **Whole-object accessors** that read or write all the fields of an object at one time.
- **Direct accessors** that provide you with a pointer to the object's storage in memory, which you can then use to read from or write to the object.
- **Extra-data accessors** that let you read or write the object's extra data.

Many of the accessor operations require that you pass a **field number**, a value that identifies the field within its class. To help you pass field numbers to accessors, the class compiler defines symbolic names for all fields at build time. The symbolic name for a field number consists of its class name, followed by an underscore, followed by the field name. The following are all examples of symbolic names for fields:

```
Viewable_color
Game_currentInning
Form_image
Shelf_shelfBorder
Angel_timeOnHold
```

In addition to these field numbers, the class compiler also defines local field numbers for all fields at build time. Local field numbers are used by accessors that always operate on the responder's fields. The symbolic name for a local field number consists of just the field's name, with no reference to its class, as follows:

```
formItems     // a field of class Form
superview     // a field of class Viewable
activeDrawer  // a field of class Drawer
```

Making Objects Usable and Storable

A ROM reference can be described in different formats depending on whether the reference is located in a local variable, or is stored in an object's field. A ROM reference can only access the object it refers to when it is located in a variable. This is known as the reference's **usable format**. To make sure that the reference from a field can be used to access the object, the object runtime defines the `MakeUsableReference` operation of class `Object`. Because object fields can store both RAM references and ROM references, you must make the reference stored in the field usable with the `MakeUsableReference` operation before fore using it.

Similarly, class `Object` defines a `MakeStorableReference` operation to convert reference before storing it into a field. Before storing an reference in a field, you must make the reference storable with the `MakeStorableReference` operation.

Depending on the accessor you use, you may not have to call the conversion operations `MakeUsableReference` or `MakeStorableReference`; some accessors call them for you automatically. Specifically, the single-field accessors call the conversion operations for you automatically. When you use whole-object accessors, you must call the conversion operations directly. When you use extra-data accessors to read references in extra data, you must also call the conversion operations directly.

To modify a reference in a field you'll typically follow this sequence:

1. Call an accessor to get access to an object.
2. If you didn't use a single-field accessor, call `MakeUsableReference` on the appropriate reference to convert it.
3. Use it however you want.
4. Call `MakeStorableReference` to convert the reference.
5. Store the reference in the field.

Using Accessors

The following sections describe the operations in each of the four families of accessors and discuss when you might use them.

Single-Field Accessors

You can call a single-field accessor to read or set the value of any field that contains a component number or a value of type Object, Unsigned, Signed, UnsignedShort, SignedShort, or Boolean. To access a field of any other type, you must use whole-object or direct accessors. See the appropriate sections below for details.

Accessor name	What it does
`Field`	returns the value of an object's field; used within a method of that object's class only
`SetField`	sets the value of an object's field; used within a method of that object's class only
`FieldOf`	returns the value of a field of an object of any class
`SetFieldOf`	sets the value of a field of an object of any class

Call `Field` to get the value of a field from within a method of that object's class. `Field` takes two parameters: the responder, which you normally address in your C source as `self`, and the local field number of the field that has its value read. `Field` returns the value of the field as its function result.

Similarly, you can call `SetField` to set the value of a field of an object from within a method of that object's class. `SetField` takes the responder, which should be `self`, the local field number, and the field's new value as parameters.

Following are examples of `Field` and `SetField`:

```
corridorSize = Field(self, corridorSize);
Reference rightArrow = Field(self, rightArrow);
return ((Field(self, dwFlags) & kBunnyShowingMask) != 0);
```

```
SetField(self, mode, 0);
SetField(self, numDrawsLong, Field(self, numDrawsLong) + 1);
```

You can use `Field` and `SetField` only to work with fields of objects from within the methods of that object's class; that is, the first parameter must always be the responder. For other objects, call `FieldOf` to get the field's value and `SetFieldOf` to set its value.

When you call `Field` or `SetField`, you can only read or set the fields that are defined by the object's class - not any of its superclasses. For example, if you call `Field` with an object of class `Box`, you can only access fields defined by that class, not by its superclasses, `Viewable` or `HasBorder`. For other objects, call `FieldOf` to get the value of the object's field and `SetFieldOf` to set the new value to the field.

`FieldOf` takes two parameters: the object that has its field read, and the field number of the field. `FieldOf` returns the value of the field as its function result. `SetFieldOf` takes three parameters: the object, the field number to be set, and the new value.

These are examples of `FieldOf` and `SetFieldOf`:

```
return FieldOf(self, HasBorder_border);
userLevelList = FieldOf(iCommandWindow, CommandWindow_userLevelList);
SetFieldOf(self, NameBar_pageArrow, FieldOf(iNameBar, NameBar_pageArrow));
SetFieldOf(self, Shelf_shelfBorder_h, newBorderSize);
SetFieldOf(iDesk, Scene_screen, officeScreen);
```

Note that `Field` and `SetField` always work on the responder and take a local field number (just the field name), while `FieldOf` and `SetFieldOf` work on any object and take a field number (the class name, an underscore, and the field name).

When you call `FieldOf` or `SetFieldOf`, you can only read or set the fields that are defined by the class you specify- not any of its superclasses.

All single-field accessors automatically call `MakeUsableReference` or `MakeStorableReference` if necessary. You shouldn't call them yourself when you use single-field accessors to work with fields.

When you change an object, the object may be shadowed, using up precious RAM for the shadowed version. If you call `SetField` or `SetFieldOf` without changing the value of the field, the object runtime doesn't actually modify the object, so no shadow object is created. Because of this, you can call `SetField` or `SetFieldOf` without checking to see if the field will actually change. The object runtime will avoid modifying or shadowing the object unnecessarily.

`Field` and `FieldOf` return will return typed values. If a field is defined to contain a reference, a reference will be return. If a field is defined to contain a boolean value, a boolean value is returned. You should use caution if you cast the return value to a different type.

Whole-Object Accessors

You can use a whole-object accessor to read or set the values of any fields of an object. Whole-object accessors provide a way to use fields of types other than those that work with the single-object accessors; that is, types other than Object, Unsigned, Signed, UnsignedShort, SignedShort, and Boolean. Whole-object accessors are also efficient when you want to read or set more than one or two fields of an object.

Whole-object accessors work by reading or setting all the fields of an object at once. To set one or more fields of an object, you'll typically call an accessor to read all the fields of an object, change the fields you want, then call another accessor to write the changes back.

Accessor name	What it does
`ReadFields`	returns the values of all the fields of an object; used within a method of that object's class only
`WriteFields`	sets the values of all the fields of an object; used within a method of that object's class only
`ReadFieldsOf`	returns the values of all the fields of an object of any class
`WriteFieldsOf`	sets the values of all the fields of an object of any class

Call `ReadFields` to get the values of all the fields of an object from within a method of that object's class. `ReadFields` takes two parameters: the responder, which should be `self`, and a pointer to storage for the field values.

Similarly, you can call `WriteFields` to set the values of all the fields of an object from within a method of that object's class. `WriteFields` takes the same two parameters as `ReadFields`: the responder, which should be `self`, and a pointer to the storage for the field values.

To help you create storage for the field values when calling `ReadFields` and `WriteFields`, the class compiler defines a symbolic name for each class that specifies a structure containing the class's fields. The symbolic name for this structure is the class name, followed by an underscore and the word fields. You can use this symbolic name in your C source to allocate storage for the fields as follows:

```
ClownStrike_Fields fields;  // allocate a structure to hold the fields'
                            // values
```

Once you've created a structure to store the fields, you can use the whole-object accessors to work with the object. Assume that class `ClownStrike` is defined as follows in a class definition file:

```
Define Class ClownStrike;
    field strikeStart: Unsigned;    // unsigned long - 32 bits
    field strikeLevel: SignedByte;   // field of unusual size - 8 bits
    field onStrike: Boolean;        // Boolean - 1 bit
End Class;
```

The following example shows how to change the value of field *strikeLevel*. Because this field doesn't occupy exactly 1, 16, or 32 bits, you can't use the single-field accessors described above. Instead, the example shows how to use whole-object accessors to change the field.

```
ReadFields(self, &fields);       // read all fields
fields.strikeLevel += 1;         // increment 8-bit field
WriteFields(self, &fields);      // write fields back to object
```

The example first reads the values from the object into the storage that fields defines. Then, the value of field *strikeLevel* is incremented in memory. Finally, the values of the fields in memory are stored in the object.

When you call `ReadFields` or `WriteFields`, you can only read or set the fields that are defined by the object's class - not any of its superclasses. For example, if you call `ReadFields` with an object of class `Box`, you can only access fields defined by that class, not by its superclasses, `Viewable` or `HasBorder`.

You can use `ReadFields` and `WriteFields` only to work with objects from within the methods of that object's class; that is, the first parameter must always be the responder. For other objects, call `ReadFieldsOf` to get the values of the object's fields and `WriteFieldsOf` to write the new values to the fields.

`ReadFieldsOf` takes three parameters: the object from which the fields are read, the class number of the object's class, and a pointer to storage for the field values. `WriteFieldsOf` takes three the same three parameters: the object to which the fields are written, the class number of the object's class, and a pointer to storage for the field values.

When you call `ReadFieldsOf` or `WriteFieldsOf`, you can only read or set the fields that are defined by the class you specify- not any of its superclasses.

Unlike single-field accessors, whole-object accessors do not automatically call `MakeUsableReference` or `MakeStorableReference` when necessary. You must call them yourself if you're reading or writing fields that contain references. See "Making Objects Usable and Storable" on page 25 for more information.

When you change an object, the object may be shadowed, depending on its cluster. If you call `WriteFields` or `WriteFieldsOf` without changing the value of a field, the object runtime accesses the object anyway, and a shadow object is created, using up precious RAM for the shadowed copy even though the object hasn't changed. Because of this, you should be careful to avoid calling `WriteFields` or `WriteFieldsOf` if you haven't actually changed a field.

Direct Accessors

For more efficient access to an object's fields, you can use direct accessors to get a pointer to the fields. You can then use that pointer to get direct access to the object's fields, which you can then read or change. Direct accessors are more efficient than whole-object accessors because they don't cause reading or writing of all the object's fields.

To read or change one or more fields of an object with direct accessors, you'll typically call an accessor to get a pointer to the object, read or change the fields you want, then call another accessor to signify that you're finished with the pointer. There are two sets of direct accessor operations: one set that provides pointers that can be used to read and change an object's fields, and another set that provides pointers that can only be used to read an object's fields.

Accessor name	What it does
BeginReadFields	returns a read-only pointer to an object's fields; used within a method of that object's class only
EndReadFields	releases the object and ends the pointer's access to the object
BeginModifyFields	returns a read-write pointer to an object's fields; used within a method of that object's class only
EndModifyFields	releases the object and ends the pointer's access to the object
BeginReadFieldsOf	returns a read-only pointer to any object's fields
BeginModifyFieldsOf	returns a read-write pointer to any object's fields

After you've called the accessor to begin working with the object's fields, the object is **locked**. It is temporarily prevented from being moved by the object runtime until you finish the access. New memory allocations cannot occur while objects are locked.

To avoid these allocation problems, you should never perform any action that allocates new objects in memory while an object is locked. Many common actions, including calling almost any operation, and changing field values, can cause memory to be allocated. Because of this severe restriction, you should only use direct accessors when absolutely necessary, such as when you must access an object's extra data. When you must use direct accessors, you should minimize the period during which the object is accessed by doing as much work as possible before beginning the access or after finishing the access.

Call `BeginReadFields` to get a pointer to the object's fields when you only want to read fields, not change them. `BeginReadFields` takes one parameter, which must be the responder. `BeginReadFields` returns an untyped pointer that holds the address of the object's fields. This pointer can only be used to read the fields, not change them. After you're done reading the object's fields, call `EndReadFields` to finish the access.

Call `BeginModifyFields` to get a pointer to the object's fields when you want to read or change fields. `BeginModifyFields` takes one parameter, which must be the responder. `BeginModifyFields` returns an untyped pointer that holds the address of the object's fields. This pointer can be used to read or change the fields. After you're done reading and changing the object's fields, call `EndModifyFields` to finish the access.

To help you create storage for the field values when calling `BeginReadFields` and `BeginModifyFields`, the class compiler defines a symbolic name for each class that specifies a structure containing the class's fields. The symbolic name for this structure is the class name, followed by an underscore and the word fields. You can use this symbolic name in your C source to allocate storage for the fields as follows:

```
BigShark_Fields fields;   // allocate a structure to hold
                          // the fields' values
```

When you call `BeginReadFields` or `BeginModifyFields`, you can only read or change the fields that are defined by the object's class - not any of its superclasses. For example, if you call `BeginReadFields` with an object of class `Box`, you can only read fields defined by that class, not by its superclasses, `Viewable` or `HasBorder`.

Don't assume that you can use a pointer provided by a direct accessor to get to the fields defined by a superclass or to an object's extra data - the object runtime may store these fields and extra data separately in future versions of Magic Cap.

You can use `BeginReadFields` and `BeginModifyFields` only to access objects from within the methods of that object's class; that is, the first parameter should be `self`. For other objects, call `BeginReadFieldsOf` or `BeginModifyFieldsOf` to get pointers to the objects' fields.

`BeginReadFieldsOf` takes two parameters: the object from which the fields are read and the class number of the object's class. `BeginModifyFieldsOf` takes the same two parameters. Both operations return untyped pointers to the object that is accessed.

When you access an object with direct accessors, you can only read or change the fields that are defined by the class you specify- not any of its superclasses.

As with whole-object accessors, direct accessors do not automatically call `MakeUsableReference` or `MakeStorableReference`. You must call them yourself if you're reading or writing fields that contain references. Remember not to call `MakeUsableReference` and `MakeStorableReference` while an object is locked. See "Making Objects Usable and Storable" on page 25 for more information.

When you change an object, the object may be shadowed, depending on its cluster. If you use `BeginModifyFields` or `BeginModifyFieldsOf` without changing the value of a field, the object runtime accesses the object anyway, and a shadow object is created, using up precious RAM for the shadowed copy even though the object hasn't changed. Because of this, you should be sure to call `BeginReadFields` or `BeginReadFieldsOf` if you're only reading fields. Additionally, because memory is allocated when an object is shadowed, you cannot nest calls to `BeginModifyFields` or `BeginModifyFieldsOf`.

Extra-Data Accessors

Every object can have extra data, variable-length information that isn't stored in its fields. You can use extra-data accessors to get a pointer to the extra data. You can then use that pointer to read or change the object's extra data.

To read or change an object's extra data, you'll typically call an accessor to get a pointer to the extra data, read or change it, then call another accessor to signify that you're finished with the pointer. There are two pairs of extra-data accessor operations: one pair that provides pointers that can be used to read and change an object's extra data, and another pair that provides pointers that can only be used to read an object's extra data.

Accessor name	What it does
`BeginReadBytes`	returns a read-only pointer to an object's extra data
`EndReadBytes`	releases the object and ends the pointer's access to the object
`BeginModifyBytes`	returns a read-write pointer to an object's extra data
`EndModifyBytes`	releases the object and ends the pointer's access to the object

Call `BeginReadBytes` to get a pointer to the object's extra data when you only want to read extra data, not change it. `BeginReadBytes` takes one parameter, the object which will have its extra data read. `BeginReadBytes` returns an untyped pointer that holds the address of the object's extra data. This pointer can only be used to read the extra data, not change it. After you're done reading the object's extra data, call `EndReadBytes` to finish the access.

Call `BeginModifyBytes` to get a pointer to the object's extra data when you want to read or change the extra data. `BeginModifyBytes` takes one parameter, the object which will have its extra data read or changed. `BeginModifyBytes` returns an untyped pointer that holds the address of the object's extra data. This pointer can be used to read or change the extra data. After you're done reading and changing the object's extra data, call `EndModifyBytes` to finish the access.

After you've called the accessor to begin working with the object's extra data, the object is locked. It is temporarily prevented from being moved by the object runtime until you finish the access. New memory allocations cannot occur while objects are locked.

To avoid these allocation problems, you should never perform any action that allocates new objects in memory while an object is accessed. Many common actions, including calling almost any operations, sometimes or always cause memory to be allocated. Because of this severe restriction, you should minimize the period during which the object is accessed by doing as much work as possible before beginning the access. The object runtime provides no other way to access extra data, so you'll have to be aware of these restrictions if your objects use extra data.

3

Software Packages

This chapter describes **software packages**, collections of objects that perform functions and provide features for Magic Cap users. Software packages are often simply called **packages**. Before reading this chapter, you should be familiar with the fundamental concepts of Magic Cap's object runtime.

About Software Packages

Magic Cap includes several major sets of features, such as the datebook, notebook and name card file, that provide services usually handled by application programs on conventional computer systems. In addition to these built-in features, Magic Cap provides a platform for additional features to be added via software packages. Whenever you deliver any additional features to Magic Cap users, you'll provide a software package.

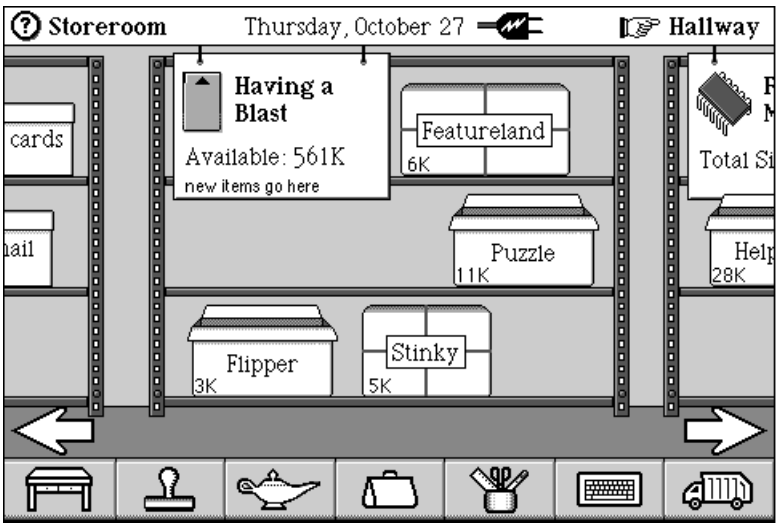


Figure 1 Software packages are represented by storage boxes on storeroom shelves

In addition to collecting the objects that implement your features, software packages also provide an interface in the storeroom for users to install and remove your objects.

Software packages can install their objects in many different places in Magic Cap. You can have your package install its objects wherever appropriate. Some typical locations for package objects are as follows:

- Packages that are generally useful or are similar to the items on the desk, such as calculators or specialized notebooks, often go in the desk accessories drawer.
- Specialized packages or sets of packages can add rooms in the hallway.
- Packages that connect to remote services appear as buildings downtown.
- Games go in the game room in the hallway.
- Sets of stamps, sounds, coupons, and other useful objects go in the Magic Hat.
- Custom pieces of stationery for telecards are installed in the stationery drawer.
- Books that are generally useful can go in the library in the hallway.
- General configuration and hardware setup features go in the control panel.

The list above shows suggested installation points for some kinds of objects. Many more installation points are available. Each package can designate where to install its objects. In many cases, users can customize the location of package objects by moving viewables in Magic Cap. For example, a user who wants quicker access to the calculator can move it out of the desk accessories drawer and place it directly on the desk. Similarly, a user can get faster access to a favorite game by copying the game's icon in the game room and placing the copy in a more convenient location.

Packages appear to users as storage boxes on shelves in the storeroom. Users can pack up, unpack, copy, send, throw away, and otherwise work with packages by manipulating their storage boxes. The user can look inside the storage box in the storeroom to learn more about a package. You can customize the scene that appears when the user looks inside the storage box for your package.

Your software packages install their objects when the user unpacks them. Many types of objects already know where they should be installed. For example, doors know to install themselves in the hallway. However, you can specify in your package the destination for objects you install. When your package is packed up or removed, Magic Cap removes your package's objects from their installation points.

When the communicator encounters technical difficulties, the contents of transient RAM aren't maintained, so your package must be designed to handle the sudden loss of transient objects. Magic Cap provides various operations that provide notification when the communicator resets so that you can allocate buffers, restart servers, and rebuild other transient objects.

Magic Cap defines an extensive set of interpackage operability features that allow users to exchange objects between packages where appropriate. Using these features, your package can use objects and classes defined by other packages. Your package can use these features to provide other packages with access to its own objects.

Kinds of Packages

A package's contents can range from a small set to a large, complex collection, providing users with a single stamp or an entire application. Every package includes exactly one object that is a member of class `SoftwarePackageContents`.

Software packages can be categorized according to how they appear to users, although there are no technical differences in creating and building the various kinds of packages. Using this categorization, there are three kinds of packages: applications, feature collections, and object collections.

Applications are packages that perform the functions of traditional software applications. Examples of applications include packages for e-mail clients, spreadsheets, and games. Applications generally provide their own scenes and usually appear to users as a door in the hallway or a building downtown, or in the desk drawer or the game room.

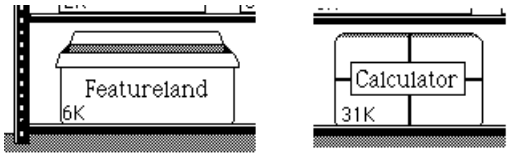
Feature collections provide tools, commands, or other functions that are available throughout Magic Cap or in particular places. For example, a feature collection package might provide a new set of drawing tools for the tool holder, a spell checker button for the Magic Lamp, a directory for the storeroom that lists all installed packages, a group of coupons that align viewables, or a handwriting recognizer pad that appears as a type of keyboard. Feature collections generally don't provide their own scenes, but instead appear inside other viewables.

Object collections install items that users can add to messages, name cards, and other objects they create. Example of these packages include collections of stamps, sets of songs, a group of datebook appointments, coupons for fonts or colors, or new stationery types for telecards. Object collections are distinguished from feature collections in that object collections add mostly content, not functions, and contain little or no code, while feature collections mainly add functions, not content, and can contain significant amounts of code.

Note that Magic Cap doesn't distinguish among these kinds of packages. They are all created in the same way, and they all contain a single object that is a member of class `SoftwarePackageContents`. The categories exist only to help users and developers understand what packages can do. Applications, feature collections, and object collections are all built by developers and manipulated by users in the same way. Some packages contain aspects of more than one of these types, while other packages defy categorization entirely.

Packages and Storage Boxes

Users occasionally need direct control over packages for such important but rare activities as installing and removing them. Users see packages as storage boxes on shelves in the storeroom. They appear in one of two states, as shown in the following figure:



At the left is a storage box for a package that is **unpacked**. An unpacked package has its objects installed and ready to use. The storage box on the right represents a package that is **packed up**. All of its objects have been removed from their places in Magic Cap, including any door in the hallway or building downtown, and none of its features can be used until the user unpacks it (although its objects still occupy memory). The only vestige of a packed-up package that is visible to users is its storage box, which remains in the storeroom.

Users can permanently remove a package by sliding it to the trash and emptying the trash. Most Magic Cap implementations include some packages in ROM. These packages can't be packed up or removed.

When the user touches a package in the storeroom, it opens to fill the screen with a **package storeroom scene**. The package storeroom scene displays information about the package and provides a user interface for other housekeeping functions.

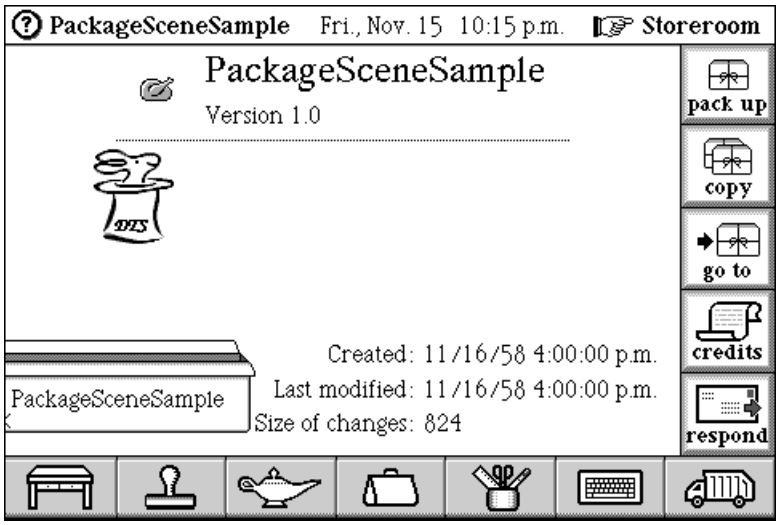


Figure 2 Package storeroom scene

In the package storeroom scene above, the name and version number of the package are displayed along with an image (in this case, a roast turkey). The scene also shows the name of the package's author (General Magic DTS) and publisher (General Magic), along with their associated images.

You specify all this information in your instance definition file when you create your package.

The right side of the scene includes five buttons used for various actions. The top button, *pack up*, packs the package and removes its objects from their places in Magic Cap. When the package is packed up, the top button changes its name and action to *unpack*. Touching the second button, *copy*, allows the user to make a copy of the package. Touching the *go to* button takes the user to the package.

The fourth button, *credits*, goes to a scene that gives more information about who made the package. Touching *respond* creates a new message card from stationery provided by the package. This card can be used for registering the software, providing comments, or any other purpose.

Magic Cap provides these buttons automatically. You don't have to specify them in your packages. The first two buttons, *pack/unpack* and *copy*, appears for every package. The other three buttons are optional and appear only if your package provides the appropriate objects. Otherwise, the buttons are disabled and are not drawn.

To give your package a go to button, specify the main entrance into your package in the *startupItem* field of your `SoftwarePackageContents` object. If your package has a custom credits scene, refer to it in the *creditsScene* field of the `SoftwarePackageContents` object. To enable the *respond* button, include an object of class `Stationery` in your package and set the `SoftwarePackageContent` object's *responseCardStationery* field to refer to it. You should also include a telecard,

which the stationary object refers to. When the user touches *respond*, Magic Cap will create a new message card from your stationery and take the user to the message scene.

When the user packs or unpacks the package, Magic Cap displays a public address announcement describing the action that just took place. Magic Cap supplies simple default announcements for these actions. If you want to provide custom announcements when the user packs, create a subclass of `SoftwarePackageContents` and override the `PackMessage` attribute. This attribute should return a text object containing the message you want displayed. To provide a custom announcement when the user unpacks, override the `UnpackMessage` attribute of class `SoftwarePackageContents`.

Every package creates new objects as it is used. In fact, just installing a package causes some new package objects to be created. Packages in RAM and in ROM cards store their changes in RAM by default, while packages on RAM storage cards keep their changes on the same card by default. The value given in the package storeroom scene as *Size of changes* indicates how much memory is occupied by changes to the package persistent cluster.

Magic Cap installs a *reset package* button in the Magic Lamp for every package storeroom scene. When the user touches *reset package*, Magic Cap throws away any changes to the package by emptying its package changes and package shadow clusters. This button and its function are provided automatically by Magic Cap. You don't have to add anything to your package to support it.

Most of Magic Cap's major built-in features, including the datebook, notebook, name card file, Internet mail client, and phone, are called built-in packages because they are true packages represented by storage boxes on the Built-in packages shelf in the storeroom.

For information on what happens to packages as the user manipulates package storage boxes, see "Package States" on page 41.

How Packages Install Objects

When the user unpacks a package, some of the objects it contains are placed in particular scenes, windows, and other Magic Cap objects. These are the package's **installed** objects, and they are placed in **receivers**. After the objects are installed, they become available to the user, implementing the features of the package. When the package is packed up, the objects it installed are removed from their receivers and are no longer available to the user.

The Installation List

Packages specify each object to be installed and each receiver in an **installation list**. The package content objects's *installationList* field refers to a the list of object pairs. The second object in each pair refers to the object to be installed, and the first object refers to the object that will get it. Magic Cap recognizes certain types of receive-install pairs and takes appropriate action to install the specified object. For some receive-install pairs, Magic Cap adds the specified object to its receiver. For example,

if the receiver is an object list, Magic Cap adds the installed object as a new entry in the object list. If the installed object is a task and the receiver is the datebook, the task is added to the datebook's collection.

For other receive-install pairs, Magic Cap takes special action to install the given object. For example, you can specify the hallway and a scene as an receive-install pair. When your package is unpacked and Magic Cap finds a hallway-scene pair, it installs a door in the hallway that takes the user to the installed scene.

The Installation Queue

Magic Cap is modular in nature. This modularity allows manufacturers to choose which Magic Cap user features to include in a product. For example, Magic Cap running on a communicator might have the desk scene, the hallway, and downtown, while a cellular phone might run another version of Magic Cap that only has the desk scene.

This modularity can be problematical for software packages that try to install objects into a receiver that does not exist on a particular version of Magic Cap. To solve this problem, many types of objects in Magic Cap inherently know where to install themselves by the specification of a **default receiver**. For example, the default receiver for doors is the hallway, the default receiver for buildings is downtown, and the datebook is the default receiver for tasks. In case the default receiver does not exist on a particular version of Magic Cap, can also specify a fallback location. For example, entrances will use in the desk accessories drawer in the desk scene as a receiver if their preferred receiver is not present. Objects that know provide default receivers inherit from the class `CanInstallSelf`.

Magic Cap does not guarantee the order in which packages are unpacked and their objects are installed. Since Magic Cap modules are also packages, it is possible that Magic Cap might try to install a package's objects before the module that package relies on has been installed into Magic Cap. To guarantee that all available modules have been set up before other package objects are installed, packages will commonly specify the **installation queue** as the receiver for their install objects. The installation queue is specified with the indexical `iInstallationQueue`. Objects that are installed into the installation queue have their installation deferred until all Magic Cap module packages have completed their installations.

All objects installed in the installation queue must inherit from class `CanInstallSelf`. Objects that provide default receivers for themselves can be directly specified in the installation list with `iInstallationQueue` as the corresponding receiver. Objects that do not specify a default receiver need to be wrapped inside an **install specifier** which is then used as the install object. Install specifiers inherit from class `InstallSpecifier`. Install specifiers describe the object to be installed, the desired receiver for the object, and a fallback object to install in case the module which provides the preferred receiver is not installed. This allows install specifiers to provide nested fallback cases so that alternative receivers can be tried until one is found.

Here is an example of a package installation list:

```
instance ObjectList installationList;
       entry: iInstallationQueue;
```

```
        entry: (Scene myPackageScene);
        entry: iHallway;
        entry: (Scene myPackageScene);
        entry: iInstallationQueue;
        entry: (InstallSpecifier cardInstaller);
        entry: iContactsMasterList;
        entry: (FullContact myContactInformation);
end instance;

instance InstallSpecifier cardInstaller;
     install: (Card myInformationCard);
    receiver: iMainFolderTray;
    fallback: (InstallSpecifier alternateCardInstaller);
end instance;

instance InstallSpecifier alternateCardInstaller;
     install: (Card myInformationCard);
    receiver: iNotebook;
    fallback: nilObject;
end instance;
```

The first receive-install pair installs a scene into the installation queue. After all Magic Cap modules have been unpacked, Magic Cap will try to install the scene by installing a door in the hallway that leads to this scene. If the hallway was not present in any of the modules, an alternate entrance that leads to the scene will be installed in the desk accessories drawer. The second receive-install pair will also install a door in the hallway. However, because the scene is installed directly into the hallway in this second example, no fallback installation can be performed if the hallway was not installed. In this case, the installation fails and the scene is not installed.

The third receive-install pair shows the use of an install specifier to install a card. The first install specifier tries to install a card into the mail folder tray in the file cabinet. If the file cabinet was not installed, the alternate install specifier is invoked. This second install specifier then tries to install the card into the notebook. If the notebook is also not present, the installation fails and the card is not installed.

The last receive-install pair installs a contact object into the list that maintains a user's contact information. The contact can be installed directly into `iContactsMasterList` because this list will be present in all Magic Cap versions. An alternative way of installing the contact object would be to create an install specifier that installed the contact into `iContactsMasterList`, and install the specifier into the installation queue.

For more information on installing objects, see "Packing and Unpacking" on page 41.

Note: Magic Cap defines a separate mechanism, known as scene additions, for installing items that are only available when a particular scene in your package is current. You can use scene additions to install the following kinds of items that appear only when a particular package scene is current: commands and rules in the Magic Lamp, tools in the tools holder, and stamps in the stamper. For more information, see the Scenes chapter of this book.

Package States

Magic Cap packages go through various state transitions as users work with them, such as the transition from being packed up to being unpacked, or the drastic transition that occurs when the user throws the package away and empties the trash. Some transitions occur very rarely, such as the one that takes place when the user copies a package from a storage card to main memory, or the emergency transition that happens when Magic Cap encounters technical difficulties and is forced to reset. Others, like the transition that happens when the communicator shuts off, can happen many times a day. This section describes what happens to your packages when these transitions occur and how you can react appropriately to each situation.

Loading Packages

Magic Cap users can add new packages and remove existing ones. Packages can be stored in various places in memory, as follows:

- in the communicator's RAM
- in the communicator's ROM
- in RAM on a storage card in a slot

A package stored in main memory or on a storage card is said to be **present** in the communicator. As soon as Magic Cap detects the presence of a new package, it **loads** the package by creating clusters for it. Magic Cap can execute code on storage cards directly, so users don't have to copy packages to main RAM in order to use them.

When the user receives a package via electronic mail or beam, the package is packed up. When the user puts a package on a RAM card, ejects the card, then reinserts the card, the package remains in the same state it was in (unpacked or packed up) when the card was ejected.

The package remains unpacked or packed up according to the state is was in when created or the last time it was removed from Magic Cap

When a package is loaded for the first time, it can be unpacked or packed up. You can set your package to unpack automatically upon loading. Otherwise, the package will be packed up or unpacked according to its state before it was loaded.

As soon as a package is loaded, its objects can be used by other packages and by Magic Cap, even if the package is packed up. However, users have no way to directly access the objects in a package until it is unpacked. Packages in the communicator's ROM can be copied to RAM or to RAM cards, but can never be removed.

Packing and Unpacking

Virtually all packages provide a way for users to manipulate their contents and perform actions, whether by installing new stamps in the stamper, adding a set of tools, or providing a door in the hallway that leads to a scene. Every package can specify a set of objects to be installed and a set of receivers that will get the objects. When the package is unpacked, Magic Cap installs the objects in their receivers.

You specify the objects to be installed and their receivers in your package's instance definition file. The *installationList* field of the package content object refers to a list of objects pairs. The first object in each pair is the object which receives installed objects. The second object in each pair is the object to be installed.

When the package is unpacked, Magic Cap calls `InstallInto` for every receiver in the installation list, passing the corresponding install object as a parameter. The receiver's class is responsible for installing the object appropriately. For example, class `StackOfCards` overrides `InstallInto` to add the install object to its stack after first checking to make sure the object is a card. Class `InstallationQueue` overrides `InstallInto` to add the install object to its list of objects that should have their installation deferred. You can create a class that overrides `InstallInto` if you want it to act as a receiver of install list objects. See "Specifying a Package Content Object" on page 48 for more information.

For each install object in the installation queue, Magic Cap calls the install object's `DefaultReceiverForSelf` operation to get the object's default receiver. If this receiver exists, Magic Cap will call its `InstallInto` operation. If the receiver does not exist, and the install object is not an install specifier, Magic Cap will call `InstallFallback` on the install object to allow the object to install itself in a fallback receiver. If the install object is an install specifier, Magic Cap will repeat the installation sequence on the object referred to in the specifier's *fallback* field.

As each object is installed, the receiver's `InstallInto` operation calls `InstallingInto` for the install list object. You can create a class that overrides `InstallingInto` if you want your objects to be notified just after they've been installed. Additionally, install objects that inherit from class `CanInstallSelf` will have their `FinishInstallingSelf` operation called after the object was successfully installed.

Your package isn't notified when it is being unpacked. You can find out when your package is unpacked by including an object of one of your package's classes as a receiver in either the installation list or in an install specifier. Then, when the package is unpacked, Magic Cap will call the package class's `InstallInto` operation, notifying it that your package is being unpacked.

When the package is packed up, Magic Cap removes the install list items from their receivers. When a package is packed up, any objects it installed are removed from the user's view, but its objects are still loaded and available to Magic Cap and to other packages. The package isn't notified when the install list items are removed. For more information on accessing package objects from another package, see "Dynamic Linking" on page 44.

Technical Difficulties

Magic Cap may encounter unexpected errors during normal operation. When such a situation arises, Magic Cap signals the error by throwing an **exception**. Magic Cap throws different types of exceptions for different kinds of errors. Packages can set up **exception handlers** to catch specific exceptions in an attempt to recover from the error and continue normal operations.

If no exception handler has been set up that is able to handle a thrown exception, Magic Cap will **reset** itself in an attempt to return to a stable state. After a reset occurs, Magic Cap will throw away and rebuild all transient clusters. In this situation, the most recent changes in transient RAM may not yet have been committed to persistent RAM, causing these changes to be lost. If you use transient objects in your packages, you must use special design considerations to recover from a reset.

Whenever Magic Cap needs to rebuild the transient clusters, it will activate each package in turn and call `ReinitializeClass` on all package classes. Typically, you'll use `ReinitializeClass` to recreate any transient buffers and other transient objects that are lost when the communicator resets. Magic Cap also check object fields of persistent objects that refer to other objects, making sure that the fields refer to valid objects in persistent RAM. If any fields are found to have invalid reference – which most likely happened because they were references to transient objects – they are set to `nilObject`.

Any buffers you created with `NewTransientBuffer` or `NewLockedBuffer` are lost when Magic Cap resets. Because you refer to transient and locked buffers with direct pointers rather than references, Magic Cap won't change the direct pointers, so these pointers will be invalid after Magic Cap resets.

Power Off and On

By far the most common transitions for a package are the ones that occur when the communicator's power turns off or on. During the power-off process, Magic Cap commits changes before turning off, causing objects to move from the package changes cluster in transient RAM to the package shadow cluster in persistent RAM.

When the user turns the power on, Magic Cap calls `ResetClass` for every class. `ResetClass` is used to notify servers and other objects that manage hardware that the communicator has just powered up. For example, the object that manages the speaker overrides `ResetClass` to make sure that any sound object that was being played at power off is deleted when power returns so that it doesn't continue to take up memory after it has been used.

You should override `ResetClass` if your class controls a server or other hardware resource that you want to set up when the communicator turns on. Unless you're creating servers or other hardware-controlling software, you'll rarely override the `ResetClass` operation. `ResetClass` is called after `ReinitializeClass` when the communicator resets.

Removing Packages

The user can remove any package that isn't in the communicator's ROM. If the package is in main RAM or on a RAM storage card, the user can throw the package away and empty the trash, destroying the software package and its changes. If the package is on a RAM storage card, the user can eject the card, unloading the package and removing the package's install list objects from their receivers.

Packages in main RAM store their changes in RAM by default, while packages on RAM storage cards keep their changes on the same card by default. If a RAM storage card is inserted and the user has checked its *new items go here* option, changes will go on the designated RAM card if the package uses the `iNewItemsGoHere` indexical or calls `NewItemNear` to create new objects. If the packages doesn't use these calls, the new objects will go to the default location indicated above. See "Creating New Objects on Storage Cards" on page 12 in the Object Runtime chapter of this book for more information.

Dynamic Linking

Magic Cap uses **component numbers** to identify classes, objects, operations, class operations, intrinsics, and indexicals. Whenever Magic Cap starts up, it will dynamically assign new numbers to every component in the system and in all present packages. When a package is integrated into a running Magic Cap environment, any components defined by that class will be assigned new component numbers that do not conflict with numbers already existing in the system. This process of renumbering components is called **dynamic linking**. Dynamic linking provides features that allow packages to share information. Packages can use these features to implement several kinds of information sharing techniques, including:

- Creating objects of other packages' classes.
- Creating subclasses of other packages' classes.
- Calling operations defined by other packages' classes.
- Importing and using objects stored in other packages.
- Exporting information about local package objects for use by other packages.
- Storing objects from other packages that are referred to by local objects.

This section provides information about how you can make components in your package available for use from other packages.

WARNING! You should never use components defined by another package unless the author of the other package has provided appropriate information about the shared objects. Using objects from other packages without thorough information can cause severe software problems, including corruption of user data in persistent RAM.

Exporting a Package Interface

When you create a package, you can create a **package interface** composed of package components that can be used by other packages. The package interface is a list of classes, operations, class operations, intrinsics, and indexicals that can be used from outside your package. When you make a package interface available for use by other packages, the interface is said to be **exported**. You specify the components of your package you want to export with an interface definition an a class definition file.

Here is an example of an interface definition:

```
// The ExportSample class is a class that can be used by other packages
```

```
define class ExportSample;
  inherits from Object;

  // public stuff
  operation InstallIntoDrawer(toBeInstalled: Viewable);

  // private stuff - not available to client packages
  operation DoSomePrivateStuff();
  field secretNumber: Unsigned;
end class;

// Indexicals declared in this package
indexical iPrototypeStampBank: ObjectList;
indexical iDrawerNames: Text;

// The package interface definition lists the components of this package
// which can be used from other packages. Interface definitions are
// usually defined in separate class definition files.
define interface ExportSampleInterface '/~genmagic.com/ExportSample/';
  class ExportSample;
  operation InstallIntoDrawer;
  indexical iDrawerNames
end interface;
```

This example defines a package interface called `ExportSampleInterface`. This name is known as the **short name** for the package interface. When other packages wish to use components exported by this interface, they use the short name to identify the interface. Magic Cap identifies package interfaces by unique **long names**. The long name for this package interface is `/~genmagic.com/ExportSample/`. Long names for package interfaces are collected by Magic Cap into tables called **cliques** when a package is integrated into a running Magic Cap environment. When more than one package defines the same interface, Magic Cap does not guarantee that one will always be used over the other.

This interface defines three components which are available for use by other packages. Specifying a class in a package interface, like this example does with class `ExportSample`, allows other packages to create new objects of this class, or to define subclasses of `ExportSample` in a class definition file.

Specifying an operation allows other packages to call that operation. In this example, other packages are able to call the `InstallIntoDrawer` operation. Note that because the operation `DoSomePrivateStuff` is not listed in the package interface, other packages cannot call this operation, even though the `ExportSample` class which defines it is also exported.

Specifying an indexical in a package interface allows other packages to access the object referenced by that indexical. This is how you would grant another package access to objects in your instance definition file. In this example, the package indexical `iDrawerNames` can be used from other packages as easily as this package does.

Although it was not done in this example, you can specify class operations and intrinsics in package interfaces in the same way.

When the class compiler encounters an interface definition, it will create a file containing symbolic information that can be included by other packages. When another package includes this information in its class definition file, the class compiler recognizes that it is using components defined by another package. Because

other packages must include information generated by the class compiler for your interface definition, you usually keep interface definitions in a separate file from class definitions in your package so that you don't provide information about components that should remain private to your package. For more information on using exported package interfaces, see the following section, Importing a Package Interface. More information is available about creating interface definitions in the section on the class definition syntax in the Object Tools chapter of *Guide to Magic Cap Development Tools*.

Importing a Package Interface

When you create a package, you may want to use components that are defined by another package. To use these components, you **import** a package interface. Once you've imported another package's interface, the components exported by that interface can be used just as if they were defined by your package or by Magic Cap itself.

Strong Imports

Before Magic Cap activates a package, it checks to make sure that all package interfaces imported by that package are present. If any of the imported interfaces is not installed, Magic Cap will not activate the package. This type of dependency on imports is known as a **strong import**, and is normally how packages import interfaces.

To use components from another package, you import the interface from that package in a class definition file. This is done by specifying the `import` keyword followed by the short name of the package interface. Here is an example of how you would import the interface of another package:

```
import ExportSampleInterface;
```

By importing ExportSampleInterface, you will be able to use any class, operation, class operation, indexical, or intrinsic listed in this interface as if these components were defined by your package, or by Magic Cap itself.

Magic Cap makes sure that all interfaces your package rely on are available for use before your package is activated. You can specify a custom message that will be shown to the user in case the interfaces your package relies upon are not all present. You create this message in your instance definition file and define an indexical to refer to it:

```
indexical iMissingNeededInterface = (Text missingInterfaceMessage);

instance Text missingInterfaceMessage;
        data: 'This package cannot be used because it relies on another'
              'package which is not installed in your communicator.';
end instance;
```

This indexical is specified in the import statement in the class definition file:

```
import ExportSampleInterface or say iMissingNeededInterface;
```

Whenever this package is integrated into a running Magic Cap environment, Magic Cap will present an announcement window containing this text if the interface named ExportSampleInterface can not be found.

Weak Imports

If your package can still provide services even if some package interfaces it normally uses are not present, Magic Cap can activate your package if you've specified a **weak import** dependency on other package interfaces. If your package weakly imports other package interfaces, Magic Cap will activate your package, but your package code is then responsible for checking for the existence of components from other package interfaces before using them.

To weakly import an interface, use the `weakly import` keyword in your class definition file:

```
weakly import ExportSampleInterface;
```

Because a Magic Cap will not prevent a package from being activated if a weak import is missing, you never use the `or say` keyword in conjunction with a weak import.

Before you use a weakly imported component in package code, you must check to make sure that it is available by comparing it against the appropriate nil component value. Here are examples of how you would make that check:

```
// Weakly imported components are resolved to the corresponding nil
// component value if the package that defines their interface is not
// available.
if (ExportSample_ != nilClass) {
   // The ExportSample class is available for use.
}

if (ExportSample_InstallIntoDrawer != nilOperation) {
   // The InstallIntoDrawer operation defined by ExportSample can be
   // called from this package
}

if (iDrawerName != nilObject)
   // The iDrawerName package indexical can be used directly from another
   // package.
}
```

You can mix weak and strong imports in your class definition file. Doing this would mean that your package would not be activated if any of the strong imports are not available, but would still be activated even if some of the weak imports are not available. Remember that your package code is responsible for checking for the availability of weakly imported components before any attempt is made to use the. Here is an example of using both weak and strong imports in a class definition file:

```
// The package will not be activated unless ExportSampleInterface is
// available.
import ExportSampleInterface;

// The package will still be activated even if WeakExportSampleInterface is
// not available.
weakly import WeakExportSampleInterface;

// The package will not be activated unless ExportSampleInterface is
```

```
// available. If this interface is not available, the message specified
// by iArrangerNotAvailable is displayed to the user.
import ArrangerInterface or say iArrangerNotAvailable;
```

Creating a Package

This section describes the objects you must include in your instance definition file when you create a new package. The most important object is the package contents object itself, but there are various other required objects.

Required Objects

When you create a software package, you must declare exactly one member of class `SoftwarePackageContents` in your instance definition file. The package content object refers to many other objects, acting as the root of a tree of objects. Taken together, all the objects you specify in the instance definition file form the package in its initial state when the package is loaded by Magic Cap.

Most packages will include the following objects, referred to by the package content object:

- An object list of the objects to be installed into Magic Cap when the package is unpacked and the corresponding receivers for each of these objects.
- Two objects of class `FullContact` that specify the author and publisher of the package.
- A text object that specifies the version number of this package.
- An object list of indexicals that refer to all of the package's scenes, including the scene that should appear when the user starts using the package. Some packages have no scenes, and so don't have this object list.
- An object list of indexicals that refer to all of the package's stacks of cards. Some packages have no stacks, and so don't have this object list.
- A list of helpful objects to be placed in information windows, and the corresponding scenes and windows that display the information windows. Packages with no scenes or windows don't have this list.

The smallest typical package would contain a package content object, the installation list, a version number, and the objects to be installed.

Specifying a Package Content Object

The package content object contains many fields. This section describes each field of the package content object. When you create a software package, you'll probably clone an existing package, including its package content object. You can use this section as a guide to interpreting or changing the values in the package content object's fields.

The first four fields, *dateCreated*, *timeCreated*, *dateModified*, and *timeModified*, are inherited from superclass `HasDate`. These fields specify the date and time the package was built and the date and time the package was last changed by the user.

You should set these fields to zero. The object compiler will fill in the correct date and time for the package's creation, and Magic Cap will maintain the fields that specify the date and time for user changes.

The next two fields are inherited from superclass `PackageContents` and help determine what happens when the package is loaded and unpacked. These fields are *autoActivate* and *installationList*.

Use the *autoActivate* field to indicate what will happen when the package is loaded. If *autoActivate* is true when the package is loaded, Magic Cap will unpack the package and go to the scene indicated by the *startupScene* field (described below). If *autoActivate* is false when the package is loaded, the package will be packed up when it is loaded.

The *installationList* field refers to a list of object pairs. The first object in each pair is the object that will be the receiver of an install object. The second object in each pair is the install object itself. When the package is unpacked, Magic Cap calls InstallInto on each receiver to perform the installation. You should list in the installation list every object you want to install into Magic Cap when your package is unpacked, coupled with its receiver, which directly precedes it in the installation list. See "How Packages Install Objects" on page 38 for more information.

When the user packs up your package, Magic Cap updates the installation list by removing any objects it installed that are no longer in your package. These objects may have been thrown away, moved into main memory, or otherwise detached from the package. In addition to removing missing objects from the installation list, Magic Cap changes these lists to reflect the current state of your package's cards and stacks. If the user has added cards to or removed cards from your stacks, the installation list will reflect those changes.

For example, if your package installs a card in the name card file with the installation list and the user deletes that card, Magic Cap removes the card and name card file from the installation list. Similarly, if the user adds any cards to stacks defined by your package, Magic Cap changes your package's install and receiver lists to include the new cards and stacks.

When the user unpacks your package again, Magic Cap again installs the installation list cards into their stacks. For this reason, you should include your package's cards and stacks in the installation. In addition, you should define indexicals that refer to all your package's stacks, and list them in the in *stackIndexicalList* field of your package. When Magic Cap cleans up following an emergency shutdown or other serious error condition and finds a package card that has somehow become disconnected from its package, it uses the stacks in this list to put the card back in its package.

The rest of the fields in the package content object are defined by class `SoftwarePackageContents` itself and provide miscellaneous details about the package, many of them used internally by Magic Cap.

The *author* field refers to contact information for the author of the package. You should include in your package a full contact that specifies you as the author. Magic Cap uses this information to display the package author's name in the package storeroom scene.

You can use the *publisher* field if your package has separate entities for author and publisher. If you want to specify a publisher, include a full contact for the publisher and set this field to refer to it. Magic Cap uses this contact to display the package publisher's name in the package storeroom scene.

If you want the author and publisher's name and address information to appear in the name card file, you can provide contacts that contain the author and publisher information and install the contacts in the contact master list (`iContactsMasterList`) by using the installation list.

The *versionText* field is used to specify a text representation of the version of your package. This will usually by a version number, such as "1.0," or "7.5", but can be you can use any string that the user can use to differentiate between different releases of your package, like "PackageSceneSample for Workgroups," and "PackageSceneSample '97." Magic Cap displays this information in the package storeroom scene.

The *helpOnObjects* field contains a list of objects with helpful tips on how to use various parts of your package. The list is composed of object pair entries, much like the installation list. The first object in each pair is known as the **object needing help**, and is usually a scene or a window. The second object in each pair is the **help object**. The help object can be a text object, or a list of viewables that contains graphical elements as well. Whenever Magic Cap displays a scene or window from your package, it will search the help list to see if that scene or window is an object needing help. If so, Magic Cap will put a circled question mark in the upper left corner of the scene or window. If the user touches this question mark, Magic Cap displays the correct help object in a window.

You can specify objects other than scenes and windows in your package's *helpOnObjects* list. If you do this, you should display the circled question mark image whenever any of those objects are visible so the user knows that help is available for those objects. The circled question mark is available through the `iHelpImage` indexical.

By default, Magic Cap looks only in the list specified in the *helpOnObjects* field of packages to determine if help objects are available. To specify help in other locations, you can override the `Info` attribute of the object needing help.

The *sceneIndexicalList* field should contain a list of indexicals that refer to any scenes created by your package that Magic Cap should know about. Magic Cap can add any rules for scenes appearing in this list to the book of rules in the library.

The *stackIndexicalList* field should contain a list of indexicals that refer to any stacks of cards in your package. When Magic Cap cleans up following a reset and finds a package card that has somehow become disconnected from its package, it uses the stacks in this list to put the card back in its package.

If your package uses cards and stacks of cards, you should specify each card individually in your package's installation list, with the stack of cards it should be located in as the corresponding receiver. The stack should be specified with an indexical from the stack indexical list. If the user ever moves a card out of one of your stacks, Magic Cap updates the installation list entry for the card with the stack the user has moved the card to.

The *startupScene* and *startupItem* fields of the software package are used by Magic Cap to determine where to take the user when the *go to* button in the package storeroom scene is touched. Use the *startupItem* field to specify the main entrance to your package. When the user touches the *go to* button, Magic Cap will go to the scene containing the entrance and draw attention to it by flashing it. If Magic Cap cannot determine where the startup item is, it will go to the scene specified by the *startupScene* field. You would usually specify `nilObject` in this field unless your package's main entrance is in a non-standard location.

The *creditsScene* field should contain a place, usually a scene, that displays more information about your package. Magic Cap goes to this scene when the user touches the *credits* button in the package storeroom scene. If you specify nilObject in this field, Magic Cap will not display the *credits* button.

The *logo* field of the software package content can refer to an image to be displayed next to the name of your package in the package storeroom scene.

The *responseCardStationery* field of the software package can be used to specify a stationery object in your package. If this field is not `nilObject`, the *respond* button in the package storeroom scene is enabled. When the user touches *respond*, Magic Cap will create a new message card from your stationery and take the user to the message scene.

Use the *hidden* field to indicate whether the package appears on the shelf in the storeroom. If you set *hidden* to true, your package won't appear on the shelf in the unless the user turns on the *Show hidden packages* rule in the storeroom. You should set *hidden* to false, unless your package is an accessory to another package and that main package provides a user interface for removing both packages.

The *dontDeactivate* field is used internally by Magic Cap. You should set it to false in your instance definition files.

The following figure shows the package's storeroom scene and indicates the source for each item in the scene.

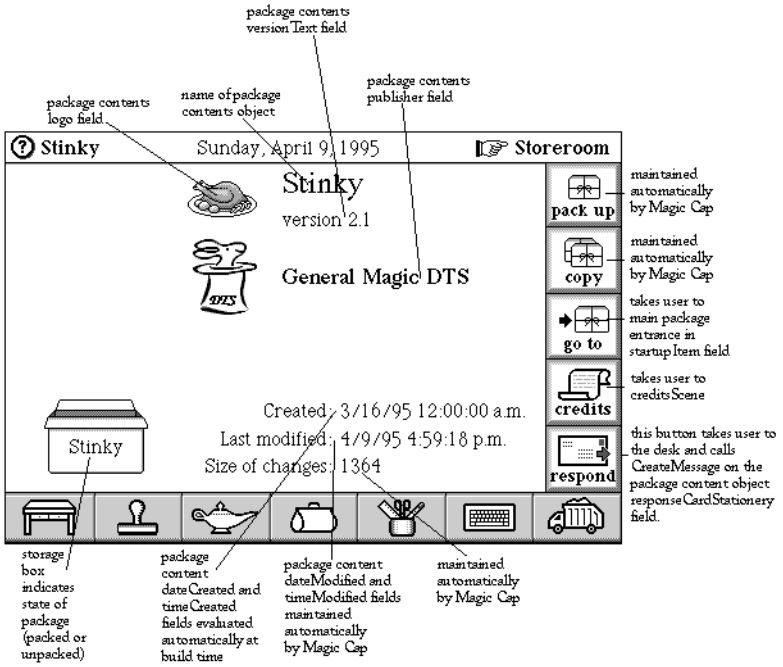


Figure 3 Storeroom scene with sources of information

Indexicals

Magic Cap provides several indexicals that contain useful objects related to packages. You can use `iNameBarPackage` to get the tiny image of a storage box that is placed in the name bar to show that a card or other object belongs to a package is. You can read `iPackageScene` to get the scene used by Magic Cap for a package's storeroom scene.

You can get or change `iResetPackagePrompt`, which contains the text that appears in an announcement when the user touches the *reset package* button in the Magic Lamp.

When the user mails a package by touching the *mail* button in the Magic Lamp in the package's storeroom scene, Magic Cap creates a new telecard from `iSendPackageStationery` to enclose the package. When the user beams the package, Magic Cap uses `iBeamPackageStationery` to create the telecard. You can read or change these objects by using the indexicals.

4

Viewables

Magic Cap provides a rich graphical environment for both users and developers. Viewable objects make up the core of this environment. This chapter describes **viewables**, objects that are members of class `Viewable`. All objects displayed on the screen by Magic Cap are members of class `Viewable`; that is, they are instances of classes that descend from class `Viewable`. In this chapter, you will learn how viewables are organized and drawn, and how they can respond to user interactions.

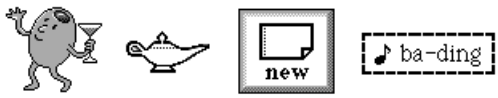


Figure 4 Some viewables

This chapter assumes you are already familiar with the Magic Cap development environment, and already know how to create basic Magic Cap packages. To learn about the Magic Cap development environment, see *Guide to Magic Cap Development Tools*. See this book's chapter on Software Packages to find out how to create a basic package. You should also be familiar with basic concepts of the Magic Cap runtime, as described in the Object Runtime chapter of this book.

Viewable classes perform the tasks of managing objects on the screen. Many of these tasks are related to drawing and redrawing viewables and involve x-y coordinates, viewable parts, highlighting, labels, images, shadows, and visibility and other viewable states. In addition, viewables work with tools to handle user touches, including selection, moving, copying, stretching, and dropping viewables.

`Viewable` is an abstract class; you can't create instances of it. Instead, you will always create instances of subclasses of `Viewable`. Class `Viewable` defines the basic fields and operations that are common to all viewable objects. Many subclasses will

override operations to customize behavior that occurs when these operations are called. The fields and operations discussed in this chapter are all defined by `Viewable`.

Geometry and Viewable Parts

This section presents fundamental concepts that include basic data structures for describing x-y coordinates and the component parts of viewables. These topics together describe the geometry of viewables.

Dots and Boxes

Magic Cap imposes an x-y coordinate plane and places viewables on that plane. Magic Cap represents points on the coordinate plane with the data type `Dot`. For simplicity and unlike most elements in Magic Cap, dots are simple data structures rather than objects. Type `Dot` is defined as follows:

```
typedef struct
  {
  longh;
  longv;
  } Dot;
```

A dot simply consists of the x and y coordinates (represented by the *h* and *v* fields, respectively) that identify a point on the coordinate plane.

Magic Cap also defines type `Box` for describing rectangles on the coordinate plane. Boxes, like dots, are defined as simple data structures rather than as a class of objects. Following is the definition for type `Box`:

```
typedef struct
  {
  longleft;
  longtop;
  longright;
  longbottom;
  } Box;
```

A box is defined by specifying the x-y coordinates that describe its upper-left and lower-right corners.

Note: Magic cap also defines a class `Box`, which is a simple framed viewable. Type `Box` and class `Box` are not related, so be sure not to confuse the two.

Magic Cap uses x-y coordinates to specify the location of viewables. For more information on x-y coordinates and the coordinate plane, see the section "X-Y Coordinates" on page 61.

Parts of Viewables

Viewables are divided into various parts. Some of a viewable's parts are associated with distinct objects related to the viewable. For example, each viewable can have a shadow, described by a shadow object that is an attribute of the viewable. The following figure shows an example viewable: an image with a border, a shadow, and a label.

Figure 5 An example viewable

Viewable boxes

There are three important rectangles associated with each viewable. Magic Cap uses type `Box` to represent each rectangle. The **content box** describes the main part of the viewable; for example, a stamp's content box encloses its image. The **border box** indicates the part a viewable where the outer edge of its border is drawn if it has a border. Many viewables have no borders; the border boxes of those viewables are the same as their context boxes. The **bounds box** encloses the viewable and all adornments drawn with the viewable, including its label, border, and shadow.

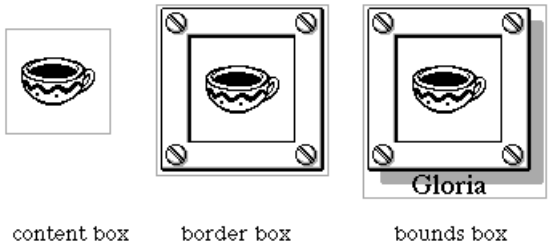


Figure 6 Boxes are indicated by thin gray lines

Class `Viewable` provides many operations that work with these three boxes. The most basic operations are used to get the boxes in viewables: `ContentBox` for the content box, `BorderBox` to get the border box, and `BoundsBox` to get the bounds box. You can also change the content box by calling `SetContentBox`. The border box and bounds box can't be set directly. Instead, they are changed indirectly when you change the content box with `SetContentBox`.

Viewables objects don't actually store these boxes. Instead, each viewable object stores its height and width in its *contentSize* field. When you call an operation to get the box around a viewable, the box is actually calculated from the value in this field. You can customize how the boxes are computed by overriding `CalcContentBox`, `CalcBorderBox` or `CalcBoundsBox`.

You can determine the width and height of the content box separately by calling `ContentHeight` and `ContentWidth`. You can also set the content's width and height separately by calling `SetContentWidth` and `SetContentHeight`.

The content box, bounds box, and border box are coordinate based. If you move a viewable on the screen, the values returned by `ContentBox`, `BoundsBox`, and `BorderBox` will reflect the viewable object's new location on the screen. You can determine the absolute size of the content box in pixels by calling `ContentSize`, and you can set the size of the content box with `SetContentSize`.

You can use `AdjustSize` to have objects of your viewable subclass recalculate their content box when you perform some action on them. To do this, override `AdjustSize` to perform the calculation that resizes the content box, and call `AdjustSize` after taking the action that should cause the viewable to resize its content.

You can find out the absolute height and width of a viewable's content box by calling `Thickness`.

Borders

Each viewable can have a **border** drawn outside its content box. The viewable's border box encloses the viewable and its border. The viewable's `Border` attribute provides access to its border. Users can add borders to most viewables by dropping border coupons on them. The following figure shows several examples of viewables with borders.

Figure 7 Viewables with borders

Note: Although viewables provide a `Border` attribute, support for borders is incomplete in class `Viewable`. To add borders, your viewable subclass should inherit from mixin class `HasBorder`. See "Borders and Shadows" on page 74 for more information.

Shadows

When the user slides a viewable on the screen, Magic Cap draws the viewable with a **shadow** as a visual cue to indicate that it has temporarily been lifted above other objects on the screen as it is being moved. When the user stops sliding the viewable, the shadow disappears. In addition to this temporary shadow, viewables can have a shadow that is always drawn, even when the viewable isn't being moved. For

example, windows are drawn with a shadow, and users can add shadows to most viewables by dropping shadow coupons on them. The viewable casts its shadow downward and to the right, as if the light source were at the upper-left corner of the screen. The viewable's `Shadow` attribute provides access to its shadow. The following figure shows some examples of viewables with their shadows.

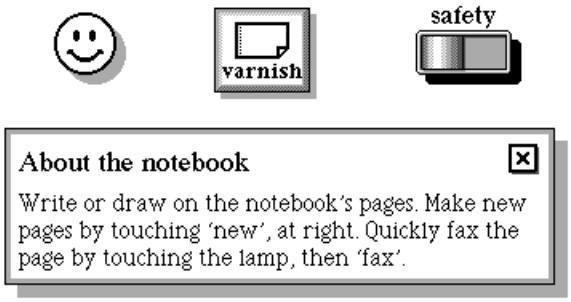


Figure 8 Viewables with shadows

Labels

Each viewable has a label that can display the viewable's name. A viewable can display its label at any one of 15 different positions, as shown in the following figure.

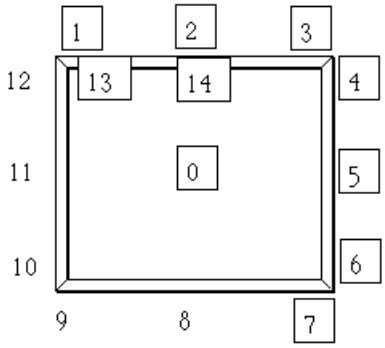


Figure 9 Available label positions

Labels can have frames. In the preceding figure, label positions 0 through 7, 13, and 14 are shown with frames. Although every viewable has a label, many viewables don't display their labels in any position.

You can show or hide the label, change the label's location, change the label's text with a text coupon from the keyboard's label maker, or turn its border on or off by tinkering with the viewable in Magic Cap simulator. You can call `ApplyText` to change the label's text; call `Label` to get the label's text. You can use the `ShowLabel` attribute to get and set whether the label is shown, and the `LabelLoc` attribute gets and sets the position of the viewable's label. The `BorderLabel` attribute gets and sets whether the label is drawn with a frame. The `CanShowLabel` attribute determines if the viewable's label should ever be shown. You can override `CanShowLabel` in your viewable subclasses to prevent the label from ever appearing.

A viewable can use a different text style for its label and for text in the viewable's content. You can get or set the label's text style by using the `TextStyle` attribute. You can get or set the content's text style by using the `ContentTextStyle` attribute.

When the user touches a viewable on the screen, Magic Cap determines which viewable was touched, then asks the viewable to determine which part of it was touched. These topics are covered in detail in "Touching Viewables" on page 67. See "Hit Testing" on page 73 for more information about parts of viewables.

Ordering and Containment

This section describes how viewables can be related and connected. These relationships determine how viewables are drawn and how they behave when users touch them.

Viewables can be associated by a containment relationship. If a viewable is contained by another, it is drawn inside its **container**, and its appearance is clipped to its container; that is, no part of it is drawn that would appear outside its container. Graphically, a viewable contained by another appears to be tucked inside its container, as shown in the following figure.

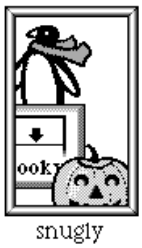


Figure 10 Viewables clipped by their container

A viewable contained by another is the **subview** of its container. A viewable that contains another is the **superview** of its contained object. Given a viewable, you can call its `Superview` operation to get its containing viewable, its `FirstSubview` operation to get its rearmost contained object, or its `LastSubview` operation to get its frontmost contained object. If a viewable has no container or no contained object, the appropriate operation returns `nilObject`.

You can find out if a particular viewable is contained by another viewable by calling the container's `IsSubviewOf` operation. You can see if a viewable has any direct subviews of a particular class by using the `FirstSubviewOfClass` operation.

To perform a particular operation on all the subviews of a container by using the `EachSubview` operation. This operation takes a function as a parameter, and will call that function for each subview of the viewable you specify.

Class `Viewable` also defines many operations to find information about a viewable's superview. You can determine if a viewable is contained by a specific `Viewable` subclass by calling the `EnclosingViewableOfClass` operation. This operation returns a viewable of the specified class that is a superview of `self`. Because viewables

commonly appear on cards, in scenes, or in windows, Magic Cap defines high level operations, `EnclosingCard`, `EnclosingScene`, and `EnclosingWindow`, to find containers of those classes.

When the user rearranges viewables on the screen and changes their containment relationship by sliding viewables into and out of other viewables, Magic Cap updates the viewables to reflect their changed status, and the many containment operations return the updated information. You can call `SwitchContainer` to remove a viewable from its container and install it in another container. You can override `ChangedContainers` to be notified when a viewable's superview has changed. You can override `ChangedContents` to be notified when a subview is about to be added or removed from a viewable.

Viewables that share the same superview form a **view chain**. View chains are stored as ordered lists inside the containing object. When viewables in a view chain are drawn, this list determines their back-to-front positions on the screen. The closer a viewable is to the front of the list, the further back it will appear on the screen. You can also keep a list of viewables in an object of class `ObjectList` to create view chains with no superviews.

Given a viewable, you can call its `Next` operation to get the viewable in front of it in its view chain, or its `Previous` operation to get the viewable behind it in its view chain. If a viewable is rearmost or frontmost in its list, the appropriate operation returns `nilObject`. When the user rearranges viewables on the screen and changes their back-to-front positioning, Magic Cap updates the list to reflect the changed order, and `Next` and `Previous` return the updated information.

A viewable in a view chain does not directly refer to the viewables in front of or behind it; the back-to-front ordering is maintained by the list. All viewables refer to their superview. The following figure illustrates the relationship between viewables.

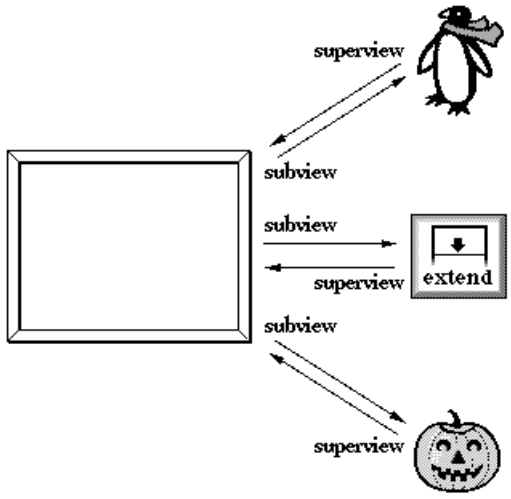


Figure 11 Viewables in a view chain

This view chain would look like this in an instance definition file:

```
instance Box allBoxedUp;
  ...  // Other fields in a viewable
  subivew: (Stamp penguin);
  subview: (Button extendButton);
```

```
  subview: (Stamp pumpkin);
end instance;
```

You can make a viewable the rearmost subview, i.e., be the first viewable in a view chain, by calling `SendToBack`. You can make a viewable the frontmost subview in a view chain by calling `BringToFront`.

You can add new viewables to a view chain by using the `SwitchContainerAt` operation. This operation will insert a viewable into a view chain at the specified position. To detach a viewable from a view chain, call `Unlink`.

You can disable a viewable's ability to act as a superview: use the `CanContain` attribute to get or change the setting of a viewable's superview capability. If you want to create a subclass of viewable with objects that should never be made frontmost in their view chain, you can override their `CanBringToFront` attribute to return `false`. For example, cards override `CanBringToFront` to always return false, ensuring that cards are never brought to the front of their view chain. Similarly, if you want to create a subclass of viewable with objects that should always be frontmost in their view chain, you can override their `AlwaysOnTop` attribute to return `true`. Viewables that are always on top can never change containers and can never be swallowed.

View Hierarchies

This section describes how viewables are grouped together into view chains and containment relationships to form complex collections of objects. This section also describes the most important such collection of viewables: the one that contains the viewables on the screen.

Viewables are often collected into a **view hierarchy**, a potentially complex group of viewables that can include multiple view chains and containment relationships. The viewables on the Magic Cap screen are an example of a view hierarchy.

Magic Cap defines exactly one object of class `Screen`. This screen object is the ultimate superview of all viewables that are drawn on the Magic Cap screen. The screen and its subviews form the **screen view hierarchy**, which contains all the objects visible to the user. If a viewable is drawn on the screen, it is in the screen view hierarchy. You can test whether a viewable is in the screen view hierarchy by using the `OnScreen` attribute. However, not every viewable in the screen view hierarchy is necessarily drawn on the screen: a viewable may have coordinates that place it off the screen, for example.

All implementations of Magic Cap have a screen size of 480 by 320 pixels. Future versions may support other screen sizes. One pixel equals 256 microns (a micron is one millionth of a meter). You can convert micron values to pixels with the `MicronToPixel` macro defined by Magic Cap. If you must put explicit coordinate values in your package, you can use this macro to specify the values in microns, then convert them to pixels at runtime, making your package more independent of changes to Magic Cap screens in future versions.

The screen isn't the only example of a view hierarchy. There are many other collections of viewables, assembled into view hierarchies offscreen, which are drawn on the screen at the appropriate time. For example, a simple view hierarchy might

be assembled for a window that displays an announcement to the user. This hierarchy might include an image, a text object, and the window itself. When the appropriate situation arises for displaying the window, it is added to the screen view hierarchy and shown on the screen.

The user can change containment and back-to-front ordering by moving viewables around on the screen. The user can deposit a viewable into a willing container simply by sliding it to that new container. In addition, whenever the user moves a viewable, the viewable is brought to the front of its view chain.

When you create packages, you'll specify most of your viewables at build time in instance definition files. You'll use these instance definitions to organize your viewables into various view hierarchies. Magic Cap will place these viewables into the screen view hierarchy when they are needed, according to user actions or your package's code.

For example, suppose you create a package that installs a door in the Magic Cap hallway. When the user opens the door to start using your package, Magic Cap installs your package's scene and its subviews in the screen view hierarchy. Your scene is installed as the first subview of the screen. The screen's other subviews (the name bar and the control bar) are installed in front of your scene.

Because the first subview is rearmost, the name bar and the control bar will always be drawn over the scene if the bars and the scene overlap, making sure they are always visible. To avoid this overlap, you should create your scene in your instance definition file with a content size of 480 by 256 pixels to ensure that it fits between the name bar and the control bar in current versions of Magic Cap.

X-Y Coordinates

Magic Cap defines x-y coordinates that indicate the positions of viewables on the screen. Each viewable denotes its position by expressing its own center point as a position relative to its superview's center point. In this way, every viewable that contains other viewables imposes its own local coordinate system on its subviews. Magic Cap assigns coordinate 0,0 (the **origin**) to the center of every superview.

Note: The location of the origin is an arbitrary choice in any graphics system. The center was chosen as the origin for viewables as part of an overall design strategy for fast viewable drawing in Magic Cap.

X-coordinate values increase from left to right, and y-coordinate values increase from top to bottom. Coordinates are always expressed in microns. You can convert from microns to pixels with the `MicronToPixel` macro.

Coordinate values are specified in instance definition files by a pair of real numbers enclosed in angled brackets. Although coordinate values are expressed in microns at runtime, they are defined as pixels in your instance definition files. At build time, the object compiler will automatically convert these values into microns.

Following are examples of coordinate values specified in pixels:

```
Instance MiniViewable unnamed;
```

```
            relativeOrigin: <3.0,-19.5>;      // pixels
               contentSize: <292.0,23.0>;     // pixels
End Instance;
```

When you use the Inspector to view objects at runtime, coordinates are always displayed as pixels. If you examine the actual values in memory using a debugger, you'll see that they are stored as microns.

Because viewables express their origin in coordinates relative to their superview, the origin is known as the **relative origin**. Magic Cap provides the `RelativeOrigin` and `SetRelativeOrigin` operations to get and set a viewable's relative origin.

You can also get and set the viewable's origin in global coordinates, which are coordinates relative to the upper-left corner of the screen. Call `Origin` to get the origin of the viewable in global coordinates; call `SetOrigin` to set a new origin for the viewable, using global coordinates. When you use global coordinates, x-coordinate values increase from left to right and y-coordinate values increase from top to bottom, as with local coordinates.

Sample Screen View Hierarchy

The following figure shows an example of a screen view hierarchy.

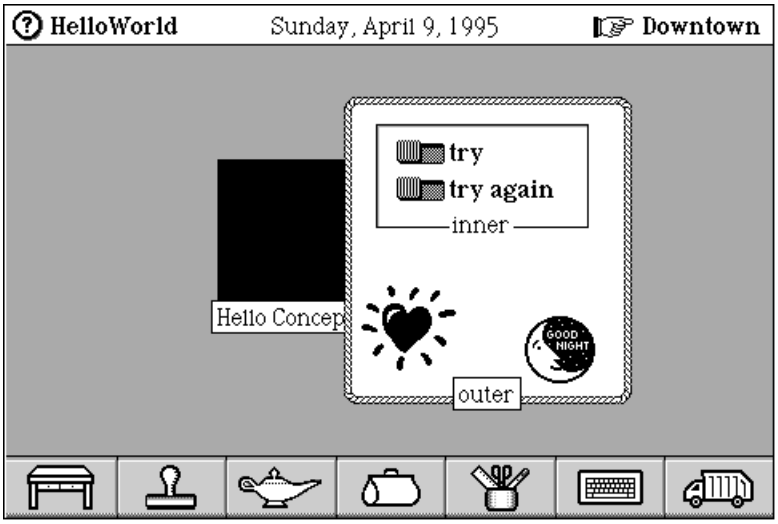


Figure 12 A sample view hierarchy

This figure shows a modified screen from the Hello World sample package. The following view hierarchy describes this screen (indented lines indicate subviews):

Screen
  Scene 'Hello World'
    Greeter 'Hello Concepts'
    Box 'outer'
      Stamp 'heart'
      Stamp 'good night'
      Box 'inner'
        Switch 'try'
        Switch 'try again'

Name Bar (and its subviews)
Control Bar (and its subviews: desk button, stamper, and so on)

The ultimate superview is the screen object, which is always the outermost container of viewables displayed on the screen. By convention, the screen's first subview is always a scene object, containing everything except the bars at the top and bottom of the screen.

The scene's first subview is the viewable named "Hello Concepts", and the next subview is the outer box. The outer box's subviews are two stamps and another box, and the inner box has two subviews itself, both switches. Note that in general, the deeper a viewable is in the hierarchy, the smaller its area: the screen object fills the entire screen; the scene encloses the area between the top and bottom bars; the subviews cover a smaller area.

Viewables that have the same superview are connected in a view chain. In this example, the scene, name bar, and control bar have the same superview (the screen), so they form a view chain. Other view chains in the example include the "Hello Concepts" viewable and the outer box; the two stamps and the inner box; and the two switches.

To demonstrate how viewables use their relative origins, the view hierarchy above is repeated below, this time including the relative origin of each viewable in pixels. Remember that the upper-left corner of the screen is location 0,0.

```
Screen <240.0, 160.0>
   Scene 'Hello World' <0.0, -8.0>
   Greeter 'Hello Concepts' <-63.0, -12.0>
   Box 'outer' <60.5, -0.5>
         Stamp 'heart' <-50.0, 50.0>
         Stamp 'good night' <45.5, 62.0>
         Box 'inner' <-4.0, -50.0>
            Switch 'try' <-39.0, -15.0>
            Switch 'try again' <-39.0, 10.0>
   Name Bar <0.0, -148.5>
   Control Bar <0.0, 140.0>
```

Each viewable's relative origin indicates the distance between its center and its superview's center. A viewable with two positive coordinates in its relative origin, such as the "good night" stamp in the hierarchy above, is centered below and to the right of the center of its superview (the "outer" box), while a viewable with two negative coordinates, such as the "try" switch, is centered above and to the left of the center of its superview (the "inner" box).

If a viewable has 0,0 as its relative origin, its center is at the same location as its superview's center.

You can change a relative viewable's location within its superview by calling the `MoveBy` operation. You can shift all of a viewable's subviews by a relative amount with the `MoveSubviewsBy` operation.

Drawing

This section discusses how to ensure that viewables are drawn and redrawn appropriately on the screen. This section also includes information on how to implement drawing if you create your own subclass of viewable.

Redrawing Viewables

When the user manipulates viewables, Magic Cap takes care of redrawing the viewables that are affected by the user's action - you don't have to take any action to make sure the screen is redrawn correctly. The process for drawing Magic Cap viewables is typically as follows:

- Some action occurs that requires a viewable to be drawn or redrawn. For example, the user drops a color coupon on a viewable. The viewable's data structures are updated to reflect the change.
- The viewable calls operations indicating that part of the screen now contains obsolete information and must be redrawn. The obsolete parts of the screen are said to be **dirty**.
- Magic Cap asks the viewables in the dirty screen areas to draw themselves.

Magic Cap asks a viewable to draw itself by calling its `Draw` operation. You should override `Draw` if you create a subclass of `Viewable` that defines objects drawn differently than objects of its superclass.

This automatic redrawing takes place whether the viewables are system objects, such as built-in stamps, or viewables created by packages, including your own packages. You can implement `Draw` in any viewable subclasses you create, and Magic Cap will call it automatically when your viewables have to be drawn.

Viewables define many operations that change their appearance, such as `SetHighlighted`, `Hop`, `SetName`, `SetVisible`, and `SwitchContainer`. Every operation that changes a viewable also ensures that the viewable is redrawn - you don't have to take any action to update the screen.

However, if you perform redrawing in an unusual way, such as by changing the fields of a viewable directly, you may have to take action yourself to make sure the changed viewable is redrawn correctly. Magic Cap provides a set of operations you can call to ensure that the viewable is redrawn, as follows:

```
operation DirtyBox(dirtyBox: Box);
    // Part of the viewable or subview has changed

operation DirtyContent();
    // The content part of the viewable has changed

operation DirtyBounds();
    // The viewable has changed; all of it must be redrawn

operation DirtyHide();
    // The viewable is going away, moving, or changing

operation DirtyShow();
    // The viewable is about to appear or reappear
```

You can call these operations when you want to force all or part of a viewable or its subviews to be redrawn. When you call one of these dirtying operations, you inform Magic Cap that part of the screen is no longer drawn correctly and must be redrawn. The dirtying calls don't actually redraw anything. Instead, they mark parts of the screen as needing redrawing. Magic Cap updates the dirty parts of the screen by calling the `Draw` operation of the appropriate viewables at **idle time** - when no other actions are pending - or immediately when the user switches scenes.

If you want to force Magic Cap to redraw the dirty parts of the screen immediately, without waiting for the next periodic update, you can call `RedrawNow`. This operation asks all dirty viewables to redraw at once. To avoid slowing down Magic Cap unnecessarily, you should call `RedrawNow` only when you want to redraw all the dirty viewables on the screen immediately.

Drawing Your Own Viewables

As mentioned above, Magic Cap calls a viewable's `Draw` operation if the viewable has been marked dirty and must be redrawn. Although class `Viewable` defines operation `Draw`, no method is included for it. Instead, concrete viewable subclasses must override `Draw` to perform their drawing action.

When you override `Draw`, you can use several techniques to perform your drawing. If your viewable contains an image that rarely or never changes, you can use the `Image` attribute to associate your viewable with an image. If you do, you should call `DrawShadowedImage` in your subclass to draw the viewable's image along with its shadow, if any. Class `Stamp` uses this technique. Each stamp has an image, and `Stamp_Draw` simply calls `DrawShadowedImage`.

If your viewable is more dynamic and can't be described by an image, you can use Magic Cap's graphics operations to perform drawing in its `Draw` override. Class `Canvas` defines many graphics operations you can use, including `FillBox`, `CopyPixels`, and `InvertBox`.

Clipping

When a viewable is drawn, Magic Cap ensures that only the appropriate area of the screen changes, prohibiting any drawing from taking place outside the bounds of the viewable. The process of restricting drawing to a given area is called **clipping**. Whenever a viewable is drawn, a clipping path defines the area in which drawing will occur.

When Magic Cap calls `Draw` to ask a viewable to draw itself, it sets up a clipping path to limit the drawing that will take place. Magic Cap forms this clipping path by computing the area of the viewable that is not covered by any objects in front of it. When you override `Draw`, any drawing you do will automatically be limited to the area described by the clipping path. You can get the clipping path that Magic Cap has set up by calling `CurrentClip`.

Although Magic Cap prevents anything from being drawn outside the clipping path, it takes some time to perform drawing, even if the drawing is clipped and never appears on the screen. If you can use logic inside your `Draw` override to determine that some part of your viewable need not be drawn, you should simply avoid drawing that part.

You may also be able to speed up your drawing if you can reduce the area of the clipping path. Normally, you'll just draw and let Magic Cap's clipping mechanism prevent anything from being drawn outside the clipping path. If you know that some additional part of the viewable should be clipped, you can modify the clipping path with operations of class `Path` before passing it on to graphics operations.

Call `Clipped` to determine if a particular box in the viewable is clipped by its superviews or subviews. The box is described in global coordinates.

Highlighting

When the user takes an action that involves a momentary selection of a viewable, such as holding a border coupon over a box before dropping it in, sliding a telecard to the out box, or tapping a button, one of the viewable involved is drawn in a special **highlighted** state to indicate that something is about to happen to it. Magic Cap provides support for highlighting viewables and redrawing them in their normal state when appropriate. Class `Viewable` defines the `Highlighted` attribute, but you must create a subclass if you want your viewables to draw in a highlighted state.

You can use the `Highlighted` attribute to get and set a viewable's highlighted state. If you create your own viewable subclasses, you can use this attribute in your `Draw` override to determine whether the viewable is highlighted.

A viewable can control whether or not subviews can be highlighted by overriding the `ContentsCanHighlight` operation. A viewable can be notified if the highlight state of any of its subviews change by overriding `SetContentsHighlighted`.

`Viewable` defines several operations that deal with viewable highlighting. Use `DrawAttentionTo` to highlight and unhighlight a viewable several times. This operations calls a lower level operation, `BlinkHighlight` to alternately highlight and unhighlight a viewable. You can call `BlinkHighlight` if you want to control the number of times a viewable is highlighted and unhighlighted, and the length of time between highlighting and unhighlighting. The operation `BlinkVisibility` is related to `BlinkHighlight`, except that instead of changing the highlight state of a viewable, it will make the viewable alternately visible and invisible. Some `Viewable` subclasses override `DrawAttentionTo` to call `BlinkVisibility` instead of `BlinkHighlight`.

For more information on the visibility attribute of viewables, see "Visibility" on page 74.

When the user slides a viewable object over certain viewables in Magic Cap, these viewables will highlight themselves with a starburst effect to tell the user that the sliding viewable can be dropped into the highlighted viewable. The starburst effect is implemented by the mixin class `StarburstHighlight`. You can create `Viewable` subclasses that inherit from this class to get the starburst highlight effect.

Colors

Each viewable includes a main color and an alternate color. Viewable subclasses decide how to use these two colors for each part of the viewable. The main color is typically used to fill the viewable. The alternate color is often not used at all, or is used if the viewable draws its parts in two different colors. The label's color is determined by the text style of the label, and the shadow's color comes from the shadow object.

Call `PartColor` to determine the color for a particular part of the viewable, or `SetPartColor` to change the color of a part. If you create a viewable subclass with additional parts, you can override `PartColor` and `SetPartColor` to handle the colors for the new parts. When the user drops a color coupon on a viewable, the coupon calls `SetPartColor` to change the viewable's color.

Although current Magic Cap communicators do not have color displays, Magic Cap supports true color. Magic Cap will use the values contained in the *color* and *altColor* fields of viewables to colorize images. Because of this, you should make sure the *color* and *altColor* fields of your viewables are not set up haphazardly for viewables that use the `Image` attribute. For these viewables, it is common for the *color* field to contain the value `rgbBlack`, and the *altColor* field to contain the value `rgbWhite`.

Touching Viewables

Magic Cap provides an elaborate system for converting user touches on the screen into actions. This section describes in detail what happens when the user touches the screen, the effects of the current tool on the touching process, and what happens when the user slides and drops viewables. This section doesn't cover lower-level details of user touches.

Overriding Touching Operations

When the user touches the screen, Magic Cap calls `Touching`, `Tap`, `Action`, `Press`, `Pressing`, and `Pressed` at appropriate times during the touching process. You can override these operations to perform an action at a particular time in the touching process. The most commonly overridden operation is `Action`.

The `Action` operation is used to perform an object's typical or usual function. For example, the keyboard gadget at the bottom of the screen overrides `Action` to display the onscreen keyboard. The overridden `Action` also checks the option key to determine whether the advanced keyboard should appear. You can override `Action` to make a viewable perform a particular function.

`Action` is often overridden by writing a Magic Script for a particular viewable. When you create a Magic Script for an object, you effectively create a subclass of the object with your script installed as an overridden implementation of the corresponding operation of the class.

You might override MovePress, CopyPress, or StretchPress if you want to perform a special action when the user touches a viewable with the move, copy, or stretch tool. See "Touching with the Arranging Tools" on page 71 for more information.

Sliding and Dropping

When the user slides a viewable across the screen, Magic Cap calls DragTrack to track the touch and handle all actions as the viewable slides.

As the user moves the viewable, DragTrack repeatedly determines which viewable is immediately under it on the screen and checks to see if that underlying viewable, called the **target**, is interested in the viewable that the user is sliding. When the user finally releases the sliding viewable, one of three things happens to it, depending on the target:

- The target becomes the sliding viewable's container and installs it as a subview.
- The target **swallows** the sliding viewable, accepting the sliding viewable and performing some action with it other than simply installing it as a subview. For example, doors in the hallway swallow objects and place them inside their rooms.
- The target accepts the sliding viewable as a coupon and performs an action indicated by the coupon.

As the user begins to slide the viewable, DragTrack calls HitTest to determine the target, then calls Swallow on the target to ask if the target is interested in accepting the sliding viewable and performing some action with it. If the target doesn't want to swallow, DragTrack calls CanAcceptCoupon on the target to ask if it will accept the sliding viewable as a coupon. If the target refuses to accept the sliding viewable as a coupon, DragTrack calls CanAccept to check whether the target will simply accept the sliding viewable as a subview.

If the target won't swallow the sliding viewable or accept it as a subview or coupon, DragTrack asks the target's superview about its interest, repeating recursively for the superview when the sliding viewable is rejected. The screen, the ultimate object in the onscreen view hierarchy, will accept any object as a subview.

This process of asking the prospective viewable about its interest in the sliding viewable is used to provide user interface feedback during the sliding process. For example, targets are usually drawn highlighted when they show an interest in swallowing or accepting the sliding viewable, and DragTrack plays sounds to indicate that the sliding viewable has been copied, has left its container, or has been dropped into a new container.

Every time the user moves the sliding viewable, DragTrack calls HitTest again to determine the target, then repeats the process of asking the target about its interest. When the user releases the sliding viewable, DragTrack finishes the tracking process by calling the sliding viewable's Pressed operation. The action that takes place depends on the interest expressed by the target.

If the target doesn't want to swallow or accept the sliding viewable, it can express two kinds of refusal, depending on the value it returns for Swallow or CanAccept. Usually, a disinterested target returns kCantSwallow or kCantAccept. If the user releases the sliding viewable at that point, it simply stays where it's released and

becomes a subview of the frontmost view that accepts subviews. Scenes always accept subviews, so the viewable becomes a subview of the scene if all viewables in front of the scene reject it.

If the user releases the touch and the target returns `kSpitOut`, the sliding viewable hops back to its original position and `DragTrack` calls `SpatOut` to give the target a chance to perform some action, such as posting an announcement. Depending on the target's preference, one of the following actions takes place:

- If the target indicated that it would swallow the sliding viewable, the sliding viewable's `Pressed` operation calls `Swallow` a second time on the target to allow it to take control of the sliding viewable and perform some action on it. To distinguish this call from the previous one which simply tests whether the target will swallow, Magic Cap calls `Swallow` with its `realSwallow` parameter set to `true`.
- If the target indicated that it would accept the sliding viewable as a coupon, the coupon's `Pressed` operation calls `ApplyAndDiscardCoupon` on the sliding viewable to apply it to the target.
- If the target indicated that it would accept the sliding viewable as a subview, `DragTrack` moves the sliding viewable to the target as a subview. `DragTrack` calls `CanChangeContainers` to find out if the target can move to another container, and `SwitchContainer` to move the target. You can override `CanChangeContainers` and `SwitchContainer` to customize their behavior.

DragTrack and StretchTrack

When a touch has been classified as one that will allow the user to slide a viewable across the screen, Magic Cap calls the viewable's `DragTrack` operation. `DragTrack` is large and complex, handling the many cases of moving viewables out of and into containers, keeping track of various flags, determining whether containers can accept objects, and much more.

You should never override `DragTrack`, except to perform some action before or after calling the inherited implementation. Instead, you can customize its action by overriding some of the many operations it calls. For example, you can override `CanChangeContainers` to indicate whether the viewable can move to a new container.

When the stretch tool is current and a touch has been classified as one which will allow the user to slide a viewable across the screen, Magic Cap calls the viewable's `StretchTrack` operation. `StretchTrack` handles the details of resizing the viewable as the user slides along the screen. You might override `StretchTrack` if you want to change the way your viewables are resized.

Advanced Touching Information

This section provides advanced information about the touching process. Most programmers won't need all the detailed information in this section.

Magic Cap maintains the current tool as a way to determine or modify what happens when the user touches the screen. Many tools are visible to users, such as the pencils, lines, shapes, move, copy, and stretch tools available by touching the tool holder at the bottom of the screen. Other tools are invisible and not directly selectable by the user. For example, when the user is going through a lesson from Magic Cap's *Getting Started* book, a special tool (`iLessonTool`) is used to ensure that the user touches the right places on the screen. You can determine the current tool by reading `iCurrentTool`.

Even if the user hasn't chosen a tool, Magic Cap still has a current tool that takes part in the touching process. This default tool is the **touch tool**, specified by `iTouchTool`.

All tools inherit from class `Tool`. `Tool` defines the operation `TouchTarget`, which is called when the user touches an object on screen. `Tool` also defines the operations that determine whether a tool is used for touching, for drawing, or both. `Tool` is a mixin class; you would never create an instance of `Tool`.

Setting up the Tool and Target

When the user touches the screen, Magic Cap determines which viewable has been touched. This viewable is called the **target** of the touch. The tool, the target, and the other viewables on the screen participate in deciding what action will take place in response to the touch.

When the user touches the screen, Magic Cap creates a **touch input** object that describes the touch. The touch input object calls the current tool's `GetToolTarget` operation to determine which viewable was touched and sets the target to be that viewable. After setting the target, the tool's `GetToolTarget` operation continues and calls the target's `ConstrainToolTarget` operation. This gives the target a chance to perform any actions, including changing the tool or the target itself.

The tool calls `ConstrainToolTarget` for the target's superview recursively; that is, it calls `ConstrainToolTarget` for all of the target's superviews up to and including the screen object. If any viewable changes the tool, the tool stops calling `ConstrainToolTarget` for that touch. If any viewable changes the target, the tool calls `ConstrainToolTarget` for the new target and its superviews. This process determines which tool and target will be used to handle the user touch.

When the tool calls `ConstrainToolTarget`, it passes a reference to the current tool. You can override `ConstrainToolTarget` to make it change the tool, customizing the behavior that occurs when the user touches your viewable. For example, class `Drawers` overrides `ConstrainToolTarget` to change the tool to the touch tool unless the user is holding down the option key. This allows touches on drawers in the stamper to open a drawer no matter what the current tool is.

After the tool and target are determined, Magic Cap calls the tool's `TouchTarget` operation. This gives the tool a chance to influence the touch handling. Many specialized tools handle the action of the user touch in their `TouchTarget` operation. For example, pencils and all other drawing tools handle the drawing process entirely in their `TouchTarget` operation.

Touching with the Arranging Tools

The **arranging** tools - move, copy, stretch, and touch - do not perform their action in `TouchTarget`. Instead, these tools call operations of the target: `MoveTouch`, `CopyTouch`, `StretchTouch`, and `Touch`, respectively. You can override these operations to further control the touching process in your viewable subclass.

The most commonly used arranging tool is the touch tool, implemented by class `TouchTool`. When the user touches a viewable with the touch tool, the tool's `TouchTarget` operation calls the viewable's `Touch` operation to classify the touch as either a **tap** or a **press**. This classification is based on whether the user moves the touch a specified distance without releasing. Once the user has moved the specified distance, 20 pixels by default, the touch is classified as a press. If the user releases the touch without moving this distance, the touch is classified as a tap.

Note: The `TapPressCriteria` operation allows you to change the default distance for classifying a touch from the default value of 20 pixels. `TapPressCriteria` also allows you to classify a touch based on the length of time the user has touched the viewable.

`Touch` calls the viewable's `TouchKind` operation to classify the touch. `TouchKind` returns immediately, without necessarily waiting long enough to classify the touch. `TouchKind` returns one of the following symbolic constants:

- If the touch is classified as a tap, `TouchKind` returns `tapKind`.
- If the touch is classified as a press, `TouchKind` returns `pressKind`.
- If the touch hasn't been classified yet, `TouchKind` returns `dontKnow`.

`TouchKind` returns `dontKnow` if the user is touching a viewable, but has not yet moved it 20 pixels or released it. If `TouchKind` returns `dontKnow`, `Touch` calls `Touching` to allow the viewable to take some action before the touch is classified, then `TouchKind`, repeating these two calls until the touch is classified.

If the touch is classified as a tap, the `Tap` operation is called, which plays the viewable's sound and call its `Action` operation.

If the touch is classified as a press, the viewable calls the `Press` operation, which calls `Touching` and `Pressing` repeatedly until the press is done, then calls Action, then calls Pressed to indicate that the press is done. You can override `Touching`, `Tap`, `Action`, `Press`, `Pressing`, and `Pressed` to customize your viewable subclass's behavior when the user touches with the touch tool and those operations are called.

`Press` calls `AutoMove` to determine if the viewable is one that the user can slide across the screen without having to use the move tool. For example, class `Stamp` overrides `AutoMove` to return `true`, allowing users to slide stamps without requiring the move tool.

The move tool's behavior is implemented by class `MoveTool`, and the copy tool's behavior is implemented by class `CopyTool`. When the user touches a viewable with the move or copy tool, the tool's `TouchTarget` operation calls the viewable's

MoveTouch or CopyTouch operations, respectively. Both MoveTouch and CopyTouch call TouchKind to classify the touch. Unlike Touch, they don't call Touching repeatedly before the touch is classified.

If the touch is classified as a tap, both MoveTouch and CopyTouch call Touch, which then calls Tap. If the touch is classified as a press, MoveTouch and CopyTouch call MovePress and CopyPress respectively. Both MovePress and CopyPress call the viewable's DragTrack operation, which tracks the user's touch across the screen, moving or copying the viewable as appropriate. DragTrack makes sure that dirty viewables are redrawn when necessary.

The stretch tool's behavior is implemented by class StretchTool. When the user touches a viewable with the stretch tool, the tool's TouchTarget operation calls the viewable's StretchTouch operation. StretchTouch calls TouchKind to classify the touch. Unlike Touch, it doesn't call Touching repeatedly before the touch is classified.

If the touch is classified as a tap, StretchTouch calls Touch, which then calls Tap. If the touch is classified as a press, StretchTouch calls StretchPress. StretchPress calls the viewable's StretchTrack operation, which tracks the user's touch, stretching the viewable appropriately. StretchTrack makes sure that dirty viewables are redrawn when necessary.

Touch Input Objects

When the user touches the screen, Magic Cap creates a touch input object that describes the touch. As the touch is handled, the touch input object is passed to many of the operations described above, including ConstrainToolTarget, Touch, Tap, and Press. The touch input object provides information about the touch, including the touch's starting and ending points and the status of the option key when the touch was made.

You can call TapHere, OptionTapCenter and TapCenter to simulate touches. These operations create touch input objects that cause Magic Cap to behave just as if the user had tapped the given location. Call TapHere to simulate a tap at a particular coordinate. Calling TapCenter simulates a tap at the center of a given viewable. Call OptionTapCenter to simulate a tap at the center of a given viewable with the option key held down.

Viewables as Tools

In addition to acting as targets, viewables can act as tools themselves. Although you probably don't think of viewables as tools, you can give a viewable the ability to act as a tool so that it can make the ultimate decision about what happens when the user touches it. The touch processing eventually calls the tool's TouchTarget operation, giving the tool final control. For example, information windows act as tools when touched; they use this ability to close themselves when the user touches anywhere inside them.

To give your viewables the ability to act as tools, inherit from class `Tool` when you create your viewable subclass. Then, override `ConstrainToolTarget` to change the current tool to the viewable itself. This will give ultimate control to the viewable's own `TouchTarget` operation.

Hit Testing

When the user touches the screen, the touch input object uses a process called **hit testing** to determine which viewable was touched. You can perform your own hit testing to determine which viewable contains a given point. For example, phone number labels on name cards use hit testing to determine what's under the spot where they are sliding in order to perform special processing when they pass over certain underlying objects.

Given a point on the screen, you can call `HitTest` to determine which viewable contains that point. Once you know which viewable contains the point, you can call `InsidePart` to determine which part of the viewable contains the point. If the viewable has an image, you can call `InsideImage` to test which part of the image includes the given point.

Viewables define the following standard part codes that can be returned by `InsidePart`:

```
#define partNothing    0    /* outside content */
#define partLabel    (-2)   /* in label */
#define partContent    1    /* inside content */
```

`InsidePart` calls `CalcInsidePart` to perform the work of determining which part was touched. If you create your own viewable subclasses, you can override `CalcInsidePart` and implement your own part codes to report which part of the viewable contains the given point. For example, corridors such as the hallway can return a part code indicating that the user touched the floor. The list above contains the part codes defined by class `Viewable`. See Utilities.h for a complete list of part codes returned by other Magic Cap viewables.

You should override `CalcInsidePart` to return your custom part codes where appropriate. In your overridden `CalcInsidePart`, you can call various operations of class `Measurement` for further hit testing. These operations include `BoxBoundsDot` to determine if a given box contains a given dot and `BoxBoundsBox` to see if one box is inside another.

Miscellaneous Viewable Features

This section describes some of the miscellaneous features provided for viewables. Some features of viewables are defined in class `Viewable`, but not fully supported except by subclasses. For example, although class `Viewable` defines attributes and operations for working with images and borders, these features are dormant in viewables unless activated by a subclass.

Hopping

Magic Cap provides a set of operations that animate the movement of viewables on the screen. This animated action, visible throughout Magic Cap's user interface, is called **hopping**. For example, when the user creates a new message, then taps *send*, the message becomes a miniature card and hops to the out box; when the user chooses a tool, the newly selected tool hops to the tool position at the bottom of the screen.

You can call `Hop` to animate a viewable moving from its current position to a given destination. If you have a destination that is another viewable, you can call `HopToViewable` to animate the viewable move. If you want a viewable to travel to the tote bag or to the trash, you can simply call `HopToToteBag` or `HopToTrash`.

For more control over the animation, call `HopFancy`, which lets you specify various parameters of the animation, including the hopping speed and acceleration.

Borders and Shadows

Class `HasBorder`, a mixin, provides full support for shadows and borders. To make a viewable subclass with shadows and borders, you should inherit from class `HasBorder`. Borders themselves inherit from class `Border`.

The `Shadow` attribute gets and sets the viewable's shadow. Even if the viewable has no shadow, calling `Shadow` returns a shadow if the user is sliding the viewable on the screen and it is being drawn with a shadow. The `ShadowOffset` attribute controls the distance of the shadow from the viewable.

Images

Each viewable can be associated with an image object. Although class `Viewable` provides attributes and operations for images, support for images is dormant unless provided by a subclass. Viewables that have an image use it when they draw themselves. Stamps draw simply by displaying their images, while buttons use their images when drawing as part of a more complex drawing operation. You can use images however you wish in any viewable subclass. There's no special class or mixin that you must inherit from.

If you want to provide images in your subclass of viewable, override the `Image` attribute to get and set the image. When working with viewables that have images, you can use the `Image` attribute to get or set the image associated with a viewable. Call `ImageBox` to determine the box that should be used to draw the image. Given a point inside an image, you can call `InsideImage` to determine which part of the image contains the point.

Visibility

A viewable can be made invisible, removing it from the screen if it's being displayed. The viewable stays in the view hierarchy, but it isn't drawn or hit tested. Use the `Visible` attribute to get and set the viewable's visibility. You can also call `ShowSubviews` to make the viewable's subview and its view chain visible. Making a

viewable invisible does not remove it from a view hierarchy. It is possible for a viewable to return `true` from `OnScreen` even while its `Visible` attribute returns `false`.

Drawing Notification

Sometimes it is convenient to perform an action just as a viewable is about to be drawn on the screen, or just before a viewable is removed from the screen. If you create a viewable subclass with objects that should perform an action at these times, you can override `AboutToShow` or `AboutToHide`. The appropriate operation will be called when the viewable is being drawn or removed. For example, song stamps have a flag that indicates whether they should play their songs whenever they are displayed. To implement this, class `SongStamp` overrides `AboutToShow` to play the song if the flag is set and overrides `AboutToHide` to stop the song if it is being played.

Text

Magic Cap provides features that support associating text with viewables. Every viewable has at least one text item: its object name, which is shown in its label. The label is drawn with its associated text style. You can use the `TextStyle` and `SetTextStyle` operations to get or set the label's text style. If you create a viewable subclass that uses other text objects, you can override the `TextStyle` and `SetTextStyle` operations to get and set the text styles of additional text objects.

Sound

Each viewable is associated with a sound object. When the user touches the viewable, Magic Cap plays the sound. The `Sound` attribute lets you get or set the sound that a viewable plays, and you can call `PlaySound` to play the viewable's sound. If you want to ensure that a sound is played, even if the viewable itself has no sound, you can call `PlaySoundWithDefault`.

Selection

Many viewables contain multiple objects, any one of which can be selected by the user for some action. For example, a choice box can include several different choices, with one choice selected at any time. If you want to implement selection in your viewable subclass, you can override the `Selected` attribute to return the selected object. If your subclass includes text objects, you can override `SelectFirstTextField` and `SelectNextTextField`, which control the selection of text fields for typing. Your overridden versions of these operations determine the order in which fields will be selected. If your viewable can scroll through a list of selectable objects, you can call `RevealSelection` to make the currently selected object visible within your viewable.

Searching

Magic Cap includes a *find* command in the Magic Lamp that searches viewables for text or other viewables. Class `Viewable` defines the operations `MatchText` and `MatchImage` which compares a viewable against the search criteria for equality.

CanBeSearched, a class that mixes in with Viewable, defines the operation SearchForObject which searches a viewable's subviews for text or images. You can perform searches programmatically by calling SearchForObject. SearchForObject calls the operation MatchTextOrImage to compare the search criteria against a given viewable. MatchTextOrImage will call either MatchText or MatchImage depending on whether the search criteria is textual or graphical. You can override MatchTextOrImage to search for other types of objects.

If you want your viewable subclass to be searchable, it should inherit from CanBeSearched.

Periodic Work

If you create a viewable subclass, you can have Magic Cap call an operation in your viewables at idle time, when no other actions are pending. When Magic Cap isn't performing any other actions, it calls the screen's Idle operation, which in turn calls Idle for all its subviews; that is, all viewables in the screen view hierarchy. If you want objects of your viewable subclass to perform some periodic action, you can override Idle. For example, clocks override Idle to update their display at idle time.

An idle method should return the minimum amount of time that should pass before being called again. This amount of time is typically the shorter of the minimum amount of time your Idle override wants between invocations and the minimum amount of time inherited Idle methods want between invocations. Idle overrides most commonly end like this:

```
Method ulong
MySubclass_Idle(Reference self)
{
  // Do stuff at idle time here
...
  // Return the smaller of the minimum amount of time superclasses
  // want between idles and the minimum amount of time this class
  // wants between calls to Idle.
  return SoonerIdle(InheritedIdle(self), myMinimumIdleTime);
}
```

The SoonerIdle operation simply returns the smaller of two unsigned values.

Scribbling and Typing

Each viewable can control whether the user is allowed to use drawing tools, such as pencils, inside it. If you create a viewable subclass, you can override the CanDrawIn attribute to explicitly prohibit or allow drawing inside the viewables, or to ask the superview to make the decision. For example, stamps override CanDrawIn to prohibit drawing inside them.

Extending

When the user creates a telecard or note card, the card can be extended by adding space at the bottom. If your viewable subclass defines objects that could be extended by adding space at the bottom, you can override operation CanExtendBottom to indicate that your viewables can be extended.

Viewables are extended when the `ExtendBottomBy` operation is called. This operation increases the space at the bottom of a viewable and all of its subviews. To extend the bottom a viewable without affecting its subviews, call `ExtendBottomShallowBy`. To extend the bottom of a viewable's subviews without affecting the viewable, call `ExtendBottomDeepBy`.

Disabling

You can use the `Disabled` attribute to make the viewable unable to accept touches, and therefore unable to be highlighted. If you create your own viewable subclasses, you can use this attribute in your `Draw` override to determine exactly how to draw the viewable. Disabling objects is dormant in viewables. The `Disabled` attribute gets and changes the disabled setting, but it has no effect on how viewables behave unless supported by a subclass. For example, buttons that set their disabled attribute to true can't be touched and don't draw their images.

Orientation

Each viewable includes an orientation setting that allows it to be drawn in a rotated position. The viewable can be drawn rotated at 90 degree positions, and each rotated image can be flipped horizontally or vertically, providing 16 different images. However, no more than 8 of these orientations will appear distinct. For example, a viewable that has been flipped vertically and rotated left 90 degrees appears the same as a viewable flipped horizontally and rotated right 90 degrees.

Figure 13 Viewables in various orientations

More symmetrical viewables have even fewer distinct images. A viewable with left and right sides that are mirror images, such as a star shape drawn with Magic Cap's shape tools, has only four distinct images.

You can use the `Orientation` attribute to get or set a viewable's orientation. `Orientation` is dormant in viewables. It is supported in animations, controls, and shapes. You can implement orientation in any viewable subclass by overriding the `Orientation` attribute. If you implement orientation, you should also use the orientation value when drawing your viewable. If your viewable uses an image to display itself and your subclass calls `DrawShadowedImage`, the orientation will be applied to the viewable when it is drawn.

Stamps, Buttons, and Controls

Magic Cap provides a large collection of **stamps**, viewable objects that users add to cards and other objects.

Figure 14 Stamps

Members of class `Stamp` are used as decorations, placed by users on telecards, name cards, notebook pages, and in other places. In addition, Magic Cap packages use stamps to provide graphical meaning in the user interface. Some stamps have additional functions. For example, song stamps play a digitized song when touched and sound stamps record sound from the communicator's microphone. Sticky notes are stamps that zoom open into small windows and contain other viewables.

Some stamps have a semantic meaning that affects the behavior of Magic Cap features. For example, users can set rules in the in box to handle specially any incoming telecards that have the *urgent*, *confidential*, or *low priority* stamps.

Even if your package defines no stamps of its own, you will likely include some stamps as subviews of your scenes and windows.

Stamps are designed mainly as graphic adornments that users can add to telecards, notebook pages, and other places. You'll rarely call any operations of stamps. Instead, you'll include stamps as subviews in some of your own viewables. If your package provides original stamps for users, you might install the stamps in the stamper. For information on installing stamps that appear in the stamper when your scene is current, see the Scenes chapter of this book.

Note: You can use Magic Cap simulator to create a stamp from any Macintosh graphic image. With Magic Cap running, select the image you want on the Macintosh, choose **Copy** from the Edit menu, select the Magic Cap window, and choose **Paste** from the Edit menu. For more information, see *Guide to Magic Cap Development Tools*.

Stamps can be set to perform some action when the user touches them, although this feature is rarely used for stamps. Instead, Magic Cap defines class `Button`, a subclass of stamp, for objects that take some immediate action when the user touches them.

Figure 15 Buttons

When the user touches a button, it highlights to enter a momentary-on state that indicates to the user that it is being touched. When the touch is released, the button returns to its usual display and the button's `Action` operation is called. Magic Cap provides various subclasses of button that implement buttons with specialized behavior.

In addition to buttons, Magic Cap defines other subclasses of stamps that are designed to perform some immediate action when the user touches, including classes `Gadget` and `Icon`. Each gadget is connected to some other viewable, usually a gadget window. When the user touches a gadget, the associated viewable is shown or hidden. Similarly, each icon is connected to another viewable, usually a scene. When the user touches an icon, the user goes to the associated scene.

After the user releases the button and it returns to its usual display, it doesn't retain any setting indicating that it was pressed. Magic Cap defines **controls** as viewables that allow users to interactively manipulate some setting, displaying information about the setting as users touch.

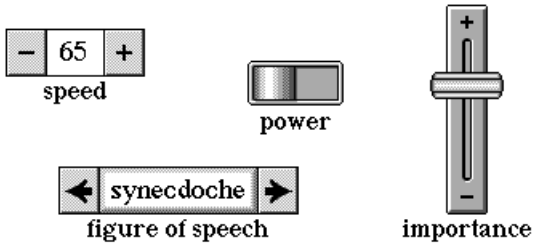


Figure 16 Controls

Magic Cap provides various kinds of controls, including switches, meters, sliders, and choice boxes. All controls display their setting and allow users to touch them to change the setting. For more information about buttons and controls, see the Buttons and Controls chapter of this book.

5

Scenes

This chapter describes **scenes**, viewable objects that contain virtually all the other viewables in the space between the name bar at the top of the screen and the control bar at the bottom of the screen. If your package displays its viewables, it will probably contain at least one scene.

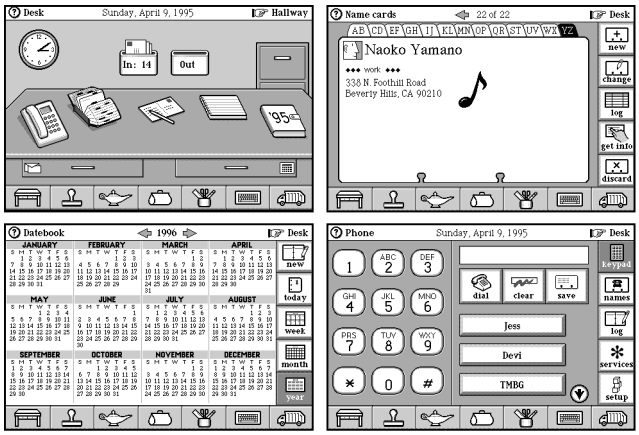


Figure 17 Examples of Magic Cap scenes. Scenes are between top and bottom bars

A scene provides the setting for the viewables you put on the screen. If your package presents more than one setting filled with viewables, you may want to include multiple scenes. For example, the notebook includes two scenes: one scene is used when displaying a page in the notebook, and another is used when showing the notebook index. As the user switches from one notebook page to another, the notebook simply installs different pages in the notebook scene.

Scenes specify a collection of viewables that you want your package to display on the screen all at once. You probably won't ever have to make a scene at runtime. Instead, you'll specify your scenes and their contents when you build your package.

When Magic Cap opens windows on the screen, the windows are not installed as subviews in the scene. Instead, windows are always subviews of the screen object. This allows windows to "float" above viewables in the scene, making windows always appear in front of the scene.

Navigation features provided by scenes allow users to look through the information in a scene and to move from one scene to another. You can create scenes that display an unchanging set of viewables, or a variety of different settings depending on user actions.

Scenes can add custom items to various system locations, such as the Magic Lamp, tools window, and stamper. Scenes can also customize the options that appear when the user touches one of the system-provided commands in the Magic Lamp, such as *mail* or *fax*.

Each scene has flags that you can use to set up specialized behavior. Magic Cap defines various indexicals for built-in scenes and other important scene-related objects, and several fields in the package contents object provide scene-related features.

Navigation

As users work with Magic Cap, they use **navigation** to move from one scene to another. Magic Cap's user interface provides various techniques for navigation. The name of a **step-back scene** appears in the upper-right corner of the screen next to the picture of a pointing hand. The user can touch there to go to the step-back scene.

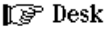


Figure 18 The hand points to the name of the step-back scene

When the user moves to a new scene, the new scene determines what its step-back scene should be. Often, the user moves to a new scene by logically "zooming in" for a closer, more detailed view. In cases like this, the new scene sets the "zoomed out" scene as the step-back scene. For example, the datebook sets the desk as its step-back scene because the datebook is a zoomed-in, detailed view of an item on the desk. Similarly, the desk sets the hallway as its step-back scene, because the desk is a zoomed-in, detailed view of an item in the hallway.

In other cases, the step-back scene is simply the previous scene. For example, the user can option-touch the Magic Lamp to move directly to the controls in the hallway. In this case, the previous scene becomes the step-back scene. In some cases, the step-back scene is chosen by other criteria. The downtown scene has no enclosing scene, so its step-back scene is arbitrarily chosen to be the hallway.

Some scenes have step-back scenes that never change. For example, the desk always has the hallway as its step-back scene, and the hallway always has downtown as its step-back scene. Such scenes are called **places**. You can call `IsPlace` to determine if a scene is a place.

Some scenes will never be the step-back scene after the user leaves. For example, the step-back scene for the in box scene is the scene the user came from to get to the in box. However, if the user came from the message scene, the in box scene's step-back scene will be the desk. A scene can specify that it should never be the step-back scene by setting the *ephemeral* field to true.

Scenes have a `StepBackScene` attribute that determines the scene that appears next to the pointing hand. If your package's scene isn't a place and the user goes to your scene, Magic Cap sets the step-back scene to the scene the user came from. If your scene is a place, Magic Cap leaves the step-back scene unchanged when the user goes to your scene. You can use the `StepBackScene` attribute to get and set this value.

If the user steps back, Magic Cap calls the scene's `StepBack` operation, which moves to the step-back scene. If the scene's `StepBackSpot` attribute isn't `nilObject`, a zooming-out effect is drawn focused on the `StepBackSpot`.

Every scene displays its name in the upper-left corner of the screen. You can override the scene's `PlaceName` operation if you want the name displayed in the upper-left corner of the screen to be something other than simply the name of the scene. For example, class `CardScene` overrides `PlaceName` to display the name of the card if the *useCardName* field is `true`.

As the user moves from scene to scene, Magic Cap keeps track of the scenes visited most recently. If the user option-taps the name of the step-back scene, a list of these scenes appears. The user can then touch a name to go directly to a scene on the list.

Figure 19 The list of recently visited scenes

Scenes can control whether or not they show up in this list by setting the *addToHistory* field. If this field is set to `false`, a scene will not be added to this list after the user leaves the scene. Note that ephemeral scenes still appear in this list.

When the user moves to a new scene, Magic Cap provides an animated visual effect that suggests zooming in to a closer view or zooming out to a wider view if that effect is appropriate. For example, when the user touches the phone on the desk, a zooming-in effect suggests that the phone is filling the screen. If the user steps back to the desk, a zooming-out effect suggests the phone scene shrinking to fit back on the desk as the user's attention widens to the entire desk.

You can go to any scene by calling `GoTo`. If you want the new scene to remember where the user came from before moving, call `GoToVia`. The new scene's step back spot will be set to the viewable you pass when calling `GoToVia`. You can override

GoTo and GoToVia to customize their behavior. These operations are called whenever the user moves from your scene to another, so you can override them to perform some custom behavior whenever the user leaves your scene.

Information Windows

Every scene can have an associated window that provides helpful information about how to work the scene's features. If the scene has an information window, Magic Cap draws a circled question mark at the left edge of the screen's name bar. The user can touch there to display the information window.

Scenes specify their information windows by listing them in the package contents object. For more information on items in the package contents object, see the Specifying a Package Content Object in the *Software Packages* chapter of this book.

Titled windows can have their own information windows.

Current Scene

The **current scene** is the scene being displayed on the screen. The current scene contains all the other viewables between the top and bottom bars, except windows. You can determine the current scene by reading the iCurrentScene indexical. When a scene becomes current, Magic Cap sets iCurrentScene to the new scene, then calls the scene's OpenScene operation. You can override OpenScene to have your scene perform some action when it becomes current. Similarly, you can override CloseScene to perform some special action when your scene is no longer current.

When the communicator is about to shut off, Magic Cap calls the current scene's ShutdownScene operation. You can override ShutdownScene to perform some action before power is shut off. However, if your scene isn't current, or an unexpected power loss occurs, your scene's ShutdownScene operation won't be called.

Cards, Stacks, and Forms

Some scenes, such as the controls scene, display a fixed set of viewables. However, many scenes vary their display depending on the user's actions. One of the most common ways to organize these variable displays within a scene is by containing viewables in cards, viewables which fill most of the scene's screen area. For example, each page in the notebook is a card. When the user looks at a page in the notebook, the card that represents that page becomes a subview of the notebook scene.

Often, many cards in a scene share some information. Magic Cap uses **forms** for the information that can be shared among cards. In some scenes, all the cards are collected together into a list called a stack of cards, also called a **stack**. For example, the notebook has one form for each of its stationery types, and all the cards in the

notebook are part of a single stack of cards. Most scenes that display a stack of cards are members of class `StackScene`, a subclass of `Scene`. See the Cards, Stacks, and Forms chapter of this book for more information on this subject.

If you're using a stack scene, you can get the card being displayed by calling `CurrentCard` or by reading `iCurrentCard`. You can get the scene's stack by calling `Stack`.

If you're using a stack scene, you can call various operations to display cards in the stack. You can display any card by calling `SetCurrentCard`. If the card isn't already in the stack, `SetCurrentCard` adds it. You can also call `GoToNext` or `GoToPrevious` to display the next or previous card in the stack, or `GoRelative` to move a specified number of cards forward or backward in the stack.

Class `Scene` defines `CurrentCard`, `Stack`, `GoToNext`, `GoToPrevious`, and `GoRelative`, but their implementations are empty, left to be defined by subclasses like `StackScene`. You can also create your own subclass of `Scene` that implements these operations however you wish. For example, you might create a subclass of Scene that uses something other than cards to contain its viewables. Such a subclass could use `GoToNext`, `GoToPrevious`, and `GoRelative` to move through the scene in the same way that stack scenes move from one card to another. For example, class `DatebookScene` overrides `GoRelative` to move through days, months, and years.

Scene Additions

When a scene becomes current, it can install its own commands, rules, and other objects in various system locations. These objects are called **scene additions**. Magic Cap provides support for the following kinds of scene additions:

kind of addition	where installed
`commands (usually buttons)`	commands area in Magic Lamp
`rules`	rules window in Magic Lamp
`tools`	tools window
`stamps (one drawer)`	bottom drawer of stamper
`banks of drawers of stamps`	stamper

Magic Cap contains examples of all these kinds of additions. Many scenes add commands, such as the *back up* command added by the storeroom scene. Many scenes add rules. For example, the in box scene adds a rule that allows the user to periodically collect electronic mail. Various built-in scenes, including the notebook, provide an example of added tools: they install the *arranging* tools. The datebook demonstrates adding a drawer of stamps: its stamps provide images for common kinds of appointments. The name cards scene add an entire bank of drawers, five drawers that provide labels for addresses and phone numbers.

Command Additions

A scene can choose to add different commands, rules, and tools depending on whether construction mode is on. For example, the desk scene adds the *tidy up* command only if construction mode is on.

To specify scene additions, include an object of class `SceneAdditions` in your instance definition file and set the scene object's *sceneAdditions* field to refer to it. If your scene has no additions, just set this field to `nilObject`.

The fields of the scene additions object specify the items to be added when the scene is current; if any of its fields are not `nilObject`, Magic Cap will automatically install the scene's additions when the scene becomes current.

You can also add additions to scenes outside your package by creating an install specifier with the addition as the object to be installed and the scene as the receiver. Magic Cap automatically installs the addition into the right place in that scene. Magic Cap has a subclass of `InstallSpecifier` specifically for providing more control over how scene additions are installed, `PerSceneInstallSpecifier`. A per scene install specifier allows you to specify whether a scene addition should be installed only when construction mode is on. You can also specify the position in a stack a card should be installed with a per scene install specifier. For more information about install specifiers, see the How Packages Install Objects section of the *Software Packages* chapter of this book.

The *commands* field of the scene additions object refers to an object list with two items. These items are object lists themselves: the first object list contains the viewables that should be displayed in the commands area if construction mode is off, and the second has the viewables to be displayed if construction mode is on. This relationship is shown in the following diagram:

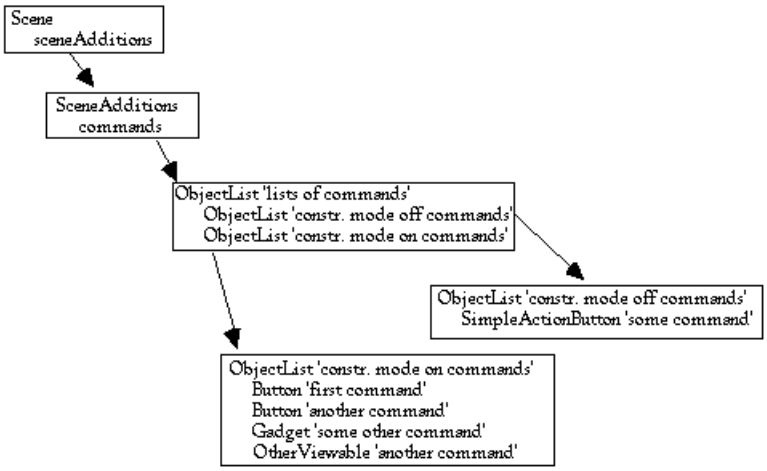


Figure 20 Command additions for a scene

When a scene adds viewables to the commands area of the Magic Lamp, Magic Cap automatically lays out the viewables in the window. The location of the viewables specified by their relative origins is ignored.

Rules Additions

In the same way, the *rules* field of the scene additions object refers to an object list with two items. These items are objects lists themselves: the first contains the rules to be displayed with construction mode off, and the second lists the rules to be shown with construction mode on.

Often, you'll want the same commands and rules to appear regardless of whether construction mode is turned on. If you do this, you don't have to declare two identical object lists of commands or rules. Simply set both entries of the main object list to refer to the same list of commands or rules.

Tools Additions

As with commands and rules, the *tools* field contains an object list with two items, one to be used with construction mode off and the other when construction mode is on. However, these object lists don't contain object lists themselves. Instead, they contain a single viewable object that will be installed in the tool window when the scene is current. To have your tools accompany a scene, declare a box that contains your tools as subviews, then make the tools object list refer to that box. This relationship is shown in the following diagram.

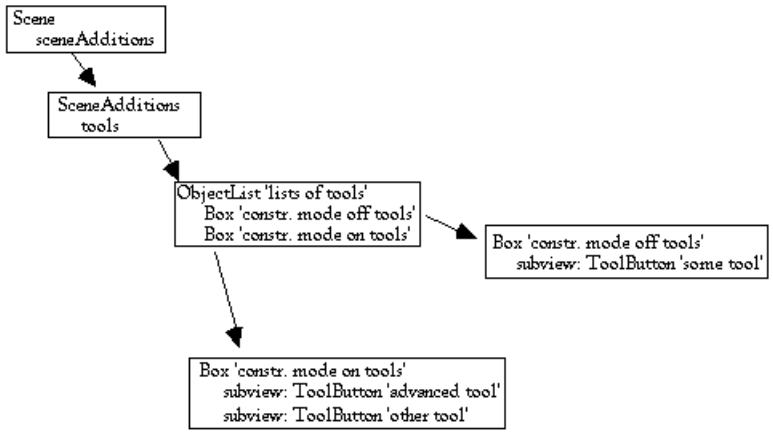


Figure 21 Tool additions for a scene

When a scene adds tool buttons in a box, Magic Cap installs the box as a subview in the tools window just as its relative origin specifies in the instance definition file. The tool buttons themselves inside the box also appear according to their relative origins. Magic Cap doesn't attempt any automatic layout.

As with commands and rules, you may want the same tools to appear regardless of whether construction mode is turned on. If you do this, you don't have to declare two identical boxes and tool buttons. Simply set both entries of the main object list to refer to the same box containing tool buttons.

Stamp Additions

Your scene can add specialized stamps to Magic Cap in either or both of these ways:

- Your scene can put any number of its own stamps into the bottom drawer of the stamper's *main* bank of drawers.
- Your scene can add one entire bank of drawers filled with stamps.

If your scene adds its own stamps, the scene additions object's *stamps* field refers to a object list of objects in a view chain. The name of the drawer is set to the name of the scene object. Unlike commands, rules, and tools, there's no way to vary the selection of stamps according to the construction mode setting.

The relationship among the pertinent objects is shown in the following diagram:

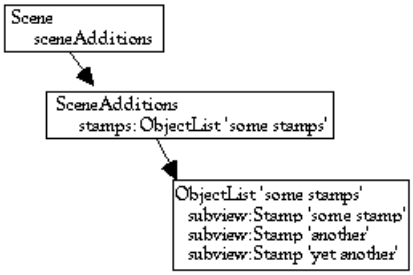


Figure 22 Stamp additions for a scene

If the scene adds its own bank of drawers, the scene addition object's *stampBankNames* field refers to a text object that contains the names of the added drawers, separated by the carriage return character (ASCII 0x0D). The number of drawers is determined by the number of names in the text object. Because of the limited area on the screen for the stamp drawers, you should have no more than 5 drawers in your bank.

To specify a carriage return character in your instance definition files, you can use a backslash followed by *n*. For example, the object compiler translates the following string in an instance definition file to the ASCII strings *one* and *two*, separated by a carriage return:

```
'one\ntwo'
```

The entire expression is enclosed in single quotation marks, which is the object compilers's standard syntax for ASCII text.

The scene addition object's *stampBankContents* field refers to an object list. This object list contains one entry for each drawer in the bank. Each entry in this object list refers to another object list that lists the view chain of objects that will appear in a drawer. The name of the bank of stamps is set to the name of the object list. Unlike

commands, rules, and tools, there's no way to vary the selection of stamps according to the construction mode setting. The relationship among these objects is shown in the following diagram:

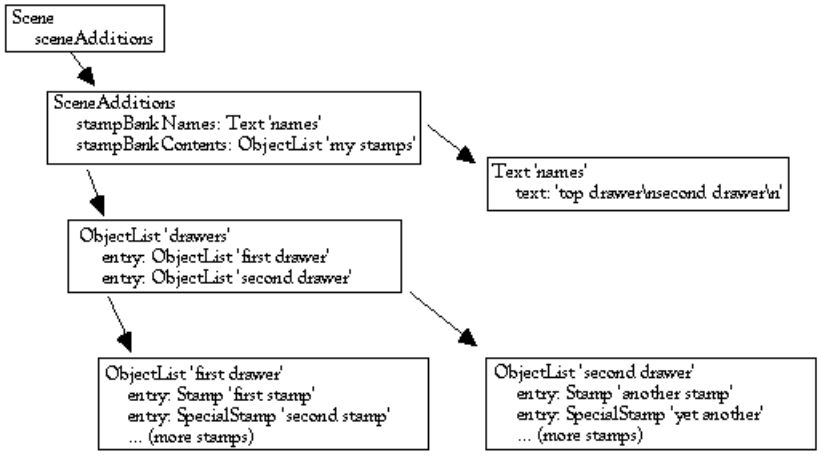


Figure 23 Stamp bank addition for a scene

When a scene adds its own stamps, Magic Cap installs the stamps as subviews in the drawer at the positions specified by their relative origins in the instance definition file. Magic Cap doesn't attempt any automatic layout of the stamps.

When you declare a scene in your instance definition files, you can use the scene's *sceneDrawer* and *sceneDrawerBank* fields to indicate which drawer should open when the user first opens the stamper while in your scene. You can specify the *general* drawer of the *main* bank by setting *sceneDrawer* to `false`, the bottom (custom) drawer of the *main* bank by setting *sceneDrawer* to `true`, or the top drawer of the scene's custom bank by setting *sceneDrawerBank* to `true`.

Class `Scene` defines attributes for `Additions`, `Commands`, `Rules`, `Tools`, `LocalStamps`, `StampBankNames`, and `StampBankContents`. However, you'll probably never use any of these directly. By specifying your scene additions in the instance definition file, Magic Cap will install and remove them automatically.

Many scenes, especially stack scenes, display a column of five buttons along the right side of the screen, between the top and bottom bars. Although many scenes have these buttons, Magic Cap doesn't provide any explicit support for this feature in its scene classes. Instead, scenes simply declare these buttons as subviews in instance definition files. For more information about setting up buttons like this, see the Stacks and Stack Scenes section of this book's Cards, Stacks, and Forms chapter.

Sending, Imaging, and Filing Scene Contents

The Magic Lamp includes buttons, available in every scene, that let the user perform various sending, imaging, and filing operations on the scene's contents. Specifically, the following buttons are available in the Magic Lamp:

command button	function
`file`	put objects or copies into a package or file cabinet
`print`	print objects to attached printer or via linked computer
`mail`	put objects on a new telecard
`fax`	send images of objects via fax
`beam`	send objects via infrared beam to another Magic Cap communicator

When the user touches one of these buttons, Magic Cap displays a window that includes a choice box listing the sets of objects that can be used when performing the command. This choice box is sometimes called the "what" box because it lists what can be sent, imaged, or filed.

The choices offered depend on the current scene. For example, touching *print* may offer the choice of printing the screen, the current card, or all the cards in the current scene. The following figure shows the *File* window and its choices when the user touches the *file* button while looking at the General Magic name card.

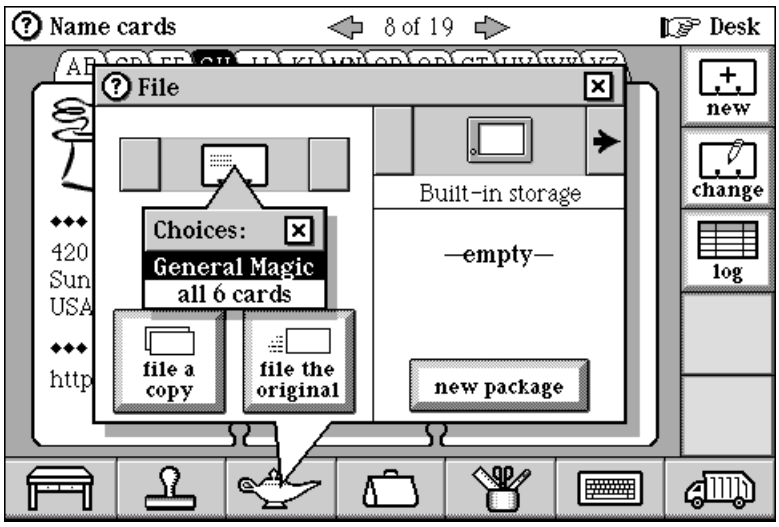


Figure 24 Choices for what to file

Content Proxies

Magic Cap supports these command buttons by using **content proxies**, place holder objects that refer to the objects being sent, imaged, or filed. When the user touches one of the buttons listed above, Magic Cap asks the current scene for a list of the sets of objects that should be presented in the choice box. Each choice is represented by a content proxy. For example, the preceding figure shows two content proxies, one

for the current name card and one for all the cards in the name card file. Magic Cap uses the content proxy to get the necessary information for performing the command.

When the user touches one of the command buttons, Magic Cap calls the scene's `MakeContentProxyChoices` operation to ask what sets of objects to list in the "what" choice box. Your scene can control what appears in the "what" box by overriding `MakeContentProxyChoices`. Magic Cap passes a parameter to `MakeContentProxyChoices` that indicates which button the user touched. This allows `MakeContentProxyChoices` to vary its response according to the action the user is performing.

When Magic Cap calls `MakeContentProxyChoices`, it passes as a parameter the list object that contains the "what" choices. To add to the choices, your scene should call `NewContentProxy` for each new choice, then call `ObjectList_AddLast` to add the new content proxy to the list. The calls to `NewContentProxy` and `AddLast` are typically combined in a single line, as in the following example:

```
AddLast(list, NewContentProxy(self, nilObject, 1, CurrentCard(self),
iPrintImage, nil));
```

The parameters to `NewContentProxy` provide the following information:

- the number of items in the newly added choice (for example, a single choice may include all the cards in a stack)
- the object that will be used to perform the command
- the image and text used to describe the choice.

`NewContentProxy` creates new content proxies based on a template object returned by the scene's `PrototypeContentProxy` attribute. Class `Scene` returns `iPrototypeContentProxy`. Override this attribute to return a different template proxy object.

For the imaging commands (*print* and *fax*), the object for which you create a proxy should be a viewable that contains all the objects to be imaged. To perform the command, Magic Cap will first make the connection to the printer or fax machine, then call `Viewable_Draw` for the container and all its contents.

For the sending commands (*mail* and *beam*), the object for which you create a proxy should be a member of class `Card` or an object list of cards. To perform the command, Magic Cap creates a new, blank telecard, then places a copy of the objects to be sent on the telecard, using an image supplied by the content proxy. Typically, all the objects will be on the current card, so you can call `NewContentProxy` to make a proxy for the current card, as in the example above. Magic Cap sends a copy of the original object, along with copies of objects that it refers to in its fields, except objects declared as `noSend` in the class's definition.

You can also send objects that aren't contained on cards, although you must do considerably more work. When Magic Cap prepares to send, it calls the content proxy's `MailOnTelecard` operation to create the telecard. `MailOnTelecard` in turn calls the `CreateContainer` operation of the object to be sent. `CreateContainer` copies the objects and places them on the telecard. To send objects that aren't cards,

you should create a subclass of `ContentProxy` and override `MailOnTelecard` and `CreateContainer`. For example, the datebook uses this approach in order to send tasks with the mail and beam commands.

For filing, the object for which you create a proxy must be a member of class `CanBeFiled` or an object list of such objects. When the object is filed with the *file the original* button, Magic Cap files the object along with objects that it refers to in its fields, except objects declared as `noSend` in the class's definition.

When the object is filed with the *file a copy* button, Magic Cap files a copy of the original object, along with copies of objects that it refers to in its fields, except objects declared as `noCopy` in the class's definition.

Scene Information in the Package Contents Object

The package contents object includes several fields related to scenes. The *sceneIndexicalsList* field of the software package is an object list that specifies indexicals that refer to all scenes in the package. Magic Cap will look for rule additions in these scenes and add them to the rules book in the library. The *helpOnObjects* field lists the scenes and windows in the package that have information windows and their corresponding information. The *creditsScene* field specifies a scene that shows author and publisher credits for the package. The user can see this scene by touching the *credits* button in the package storeroom scene in the storeroom. The *startupScene* field specifies the scene that Magic Cap goes to when the user taps the *go to* button in the package storeroom scene.

The installation list lets you create a way for the user to get to scenes in your package, such as a door in the hallway. To do this, include scenes and their corresponding locations in the installation list. For complete information on items in the package contents object, see the Software Packages chapter of this book.

Subclasses of Scene

Magic Cap provides various scene subclasses that you can use in your packages. The most important is class `StackScene`, the subclass used for displaying a stack of cards, such as the name cards. Magic Cap provides another scene subclass, `CardScene`, which is used to display a single card that is not part of a stack. Another useful subclass is `IndexScene`, which inherits from class `HasContentList`. Objects of this class are scenes that can show individual list elements or can show an index (a content list view) that summarizes all the elements in the list.

Class `ModalScene` is used for scenes that contain several screens of information, like stacks, but have little in common among the screens. Magic Cap's phone is an example of a modal scene. The various modes of a modal scene are usually implemented by having a card for each mode.

Several of the built-in packages in Magic Cap use scene subclasses to display information. The name card file uses the `NameCardsScene` subclass to display its cards. The datebook uses the DatebookScene subclass to display its tasks and appointments.

Scene Flags and Indexicals

Class `Scene` defines various flags that you can use to customize the appearance and behavior of its members. You'll use these flags primarily when you declare scenes in instance definition files.

Scene flags enable various features, including customizing the display at the top of the screen when your scene is current, instructing the *find* command to skip your scene when the user searches for something, and preventing the time and date from appearing in the name bar. There are other scene flags that provide more customization.

Magic Cap keeps track of the **current drawing tool**, the tool most recently used to draw or write. Scenes that often use drawing tools, such as the messages scene and the notebook scene, can set a flag that causes Magic Cap to choose the current drawing tool whenever the scene becomes current. You can call `DefaultTool` to determine the tool, if any, that will be set when the scene becomes current. You can also check `iLastDrawingTool` to determine the current drawing tool.

Magic Cap provides indexicals that contain useful scenes or information about scenes. Indexical `iSceneIndexicalList` is a list containing many of Magic Cap's built-in scenes, such as the desk, hallway, downtown, name cards, datebook, and so on. You can use the `GoTo` operation and an indexical from `iSceneIndexicalList` to take the user to a well-known scene.

6

Cards, Stacks, and Forms

This chapter discusses **cards**, viewables that conveniently collect and display other viewables in scenes. This chapter also discusses lists of cards called **stacks**, and **forms**, viewables that provide a means for sharing information among related cards.

Before reading this chapter, you should already be familiar with the concepts presented in the Viewables and Scenes chapters of this book. You should also be able to create simple packages with the Magic Cap development environment. For complete information on developing for Magic Cap, see *Guide to Magic Cap Development Tools*.

About Cards, Stacks, and Forms

Many package scenes collect their data into units of related information, with each unit containing roughly one screen of data. Magic Cap provides cards as a way to organize these units of information that fill most of a scene's screen area. For example, each page in the notebook is a card, as is each name card in the name card file.

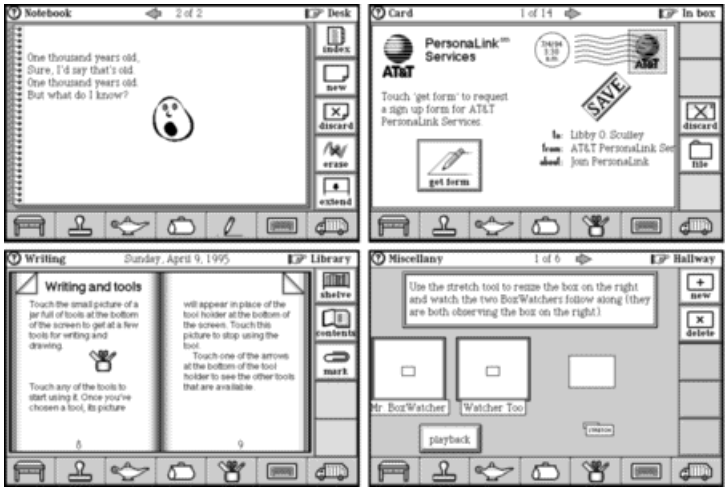


Figure 7 Various cards in their scenes

Some scenes contain only a single card, such as the message scene that is used when creating a new telecard. Scenes that contain multiple cards usually collect the cards together into a list called a **stack of cards**, also called a **stack**. Scenes that display stacks of cards are **stack scenes**. For example, the notebook and the name card file both collect their cards into stacks and display them using stack scenes.

Often, a group of cards in a stack shares some information or a common basic appearance. Magic Cap uses **forms** to represent this shared information or common appearance. Forms are viewables that contain the information shared by the group of cards. For example, all name cards in the name card file share a form, and all index cards in the name card file share a different form. The notebook uses several forms: one for plain paper, one for lined paper, one for graph paper, and one for list paper.

Each form includes items shared by all its cards. For example, in the name card file, the form for name cards includes boxes that contain addresses and telephone numbers and a viewable object used by group name cards to list their members. The form for index cards simply provides the visual background for those cards and contains no shared objects.

Similarly, the notebook forms provide shared objects and background appearance for notebook pages. The forms for plain paper and for lined paper include two text fields, one for the title of the page and one for the page's body text. The form for graph paper includes a text field for the page's title and a ruling object that draws the grid on the page. All the different notebook forms provide the spiral notebook background.

Each form includes a list of **form items**, the objects that are shared among all the cards that use the form, such as the two text fields provided by the plain paper form in the notebook. Although the text fields themselves are shared, the data that provides the objects' content (that is, the text objects themselves) are not shared - each card has different text in these objects.

To accommodate having different data for shared objects on each card, each card keeps track of the data that it places into the form items. When the card becomes current, it installs its own data into the form items. When the card is no longer current, it removes its data from the form items and stores the data in its extra data. For example, when a new plain paper notebook page becomes current, it installs its title and body text into the text fields provided by the form.

The following figure shows an example of a card in a stack (a notebook page) and indicates which parts are provided by which objects.

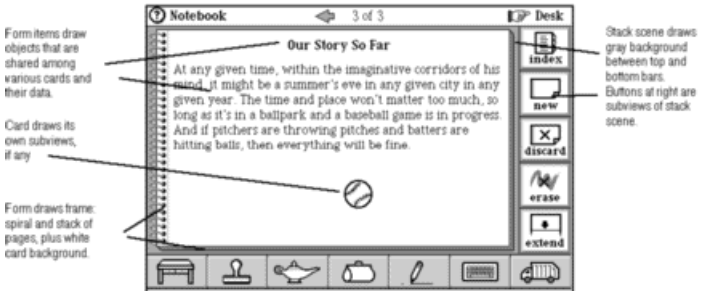


Figure 8 Card in its stack

When Magic Cap shows a scene with its stack, card, and form, the following view hierarchy is created (indented lines indicate subviews):

Screen
  Scene
    Card
      Form
        (items on the form, such as text fields)
      (items on the card, such as stamps added by user)
    (items in the scene, such as buttons at right)
  Name Bar
    (items in the name bar, such as the carousel)
  Control Bar
    (items on control bar: desk button, stamper, and so on)

You can use the card's `Stack` and `Form` attributes to get and set its stack and form. However, you'll usually indicate the card's stack and form when you declare the card in your instance definition file. You probably won't change these attributes directly at runtime.

You can create scenes that display a single card or a collection of cards. Many stacks of cards allow users to add new cards or remove existing cards. Stack scenes and cards provide navigation features that allow users to look through information and move from one card to another.

You may want to provide a way for users to work with cards that aren't on the screen. You can create tiny stand-ins for cards called **minicards** to represent cards that aren't being displayed.

When you display cards in a stack scene, you can control various settings of the stack, such as what happens when you add new cards, how to move cards to another stack, and how to remove existing cards entirely.

The following diagram illustrates the relationships among cards, stacks, forms, and stack scenes.

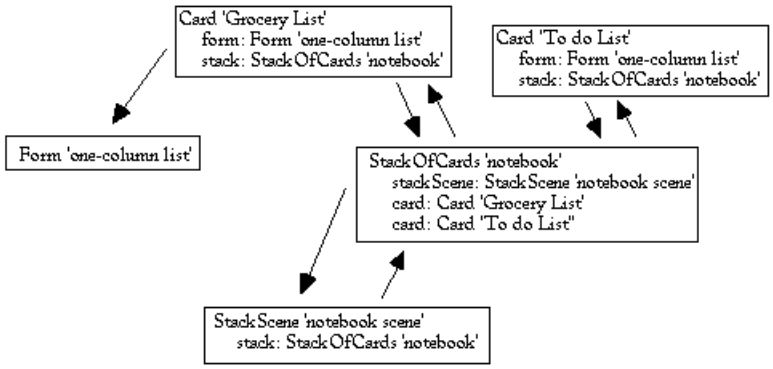


Figure 9 References among cards, stacks, forms, and stack scenes

In addition to cards, stacks, and forms, this chapter also discusses classes `StackScene`, `FormElement`, and `MiniCard`.

Navigation and Scenes

As with scenes, you can go directly to cards. Call `GoTo` to open any card in its scene. To record the spot on the screen where the user touched to go to the new card, call `GoToVia`. The new scene's step-back spot will be set to the viewable you pass when calling `GoToVia`. You can also call `GoToNext` or `GoToPrevious` to display the next or previous card in the stack, or `GoRelative` to move a specified number of cards forward or backward in the stack.

Every card has a **default scene**, the scene that will be used to display the card when opened outside its usual scene. For example, if an unsent telecard is on the desk and the user touches it, it's displayed by its default scene. You can override `DefaultScene` to return the default scene for your card subclass.

Magic Cap uses minicards as tiny representatives of cards. Minicards act as proxies in representing cards that aren't currently displayed. For example, when the user moves a telecard out of the in box and onto the desk, a minicard represents the unopened card on the desk. When the user slides a page out of the notebook index, a minicard represents the page.

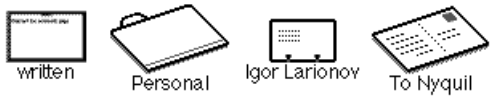


Figure 10 Minicards

You can call `InstallMiniCardNear` to create a minicard for a card and install it as a subview in any scene or viewable. `InstallMiniCardNear` calls `MiniCardPrototype` to get a template minicard. Override `MiniCardPrototype` to return a particular minicard subclass. You can display the card associated with a minicard by calling the minicard's `TapCenter` operation. The card zooms out and replaces the minicard and its scene.

You can specify the image to use for the card's minicard by overriding the card's `SmallCardImage` operation, which also specifies the image to use when displaying cards in a list view. Similarly, you can override `Object_TinyClassImage` to specify the image to be used for your card when Magic Cap displays it in the out of memory window or other similar content list.

To place a minicard in the out box, you can call the minicard's `HopToOutBox` operation. If the current scene is the desk or any scene that has an out box, the minicard will hop to the out box on the screen. If the current scene has no out box,

a window with an out box will appear, the minicard will hop to the out box in that window, and the window will disappear. The following figure shows the window and out box that appear when the current scene has no out box.

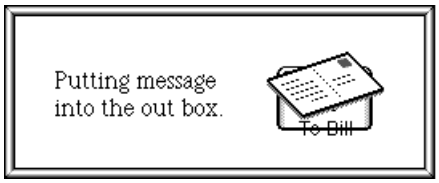


Figure 11 Out box appears in a window

Current Card

The **current card** is the card being displayed on the screen. The current card is a subview of the current scene. You can determine the current card by reading the `iCurrentCard` indexical. Magic Cap calls the current card's `AboutToShow` operation just before drawing it on the screen. You can override `AboutToShow` to have objects of your card subclass perform some action before they are drawn. Similarly, you can override `AboutToHide` to have your objects perform some action before they are removed from the screen.

You can use the stack scene's `CardNumber` attribute to determine the current card's relative position in the stack. If you want to display a particular card and you know its card number, you can call `GoToCardNum`. You can call `DeleteCurrentCardWithConfirmation` on the stack scene to delete the current card.

Magic Cap keeps track of whether a card has ever been displayed on the screen. You can find out if a card has ever been displayed by calling `WasSeen`. You can also call `SetWasSeen` to change the setting that determines whether the card has been displayed. The in box seen uses this information to display a check mark next to messages the user has read.

Forms

Each card can be associated with a form. The card's form provides two kinds of objects for the cards that use it.

- **static form objects**: viewable objects that are exactly the same on every card, such as a box or a decorative border.
- **dynamic form objects**: viewable objects containing some data that can vary from card to card, such as a text field containing text or a switch that has a setting.

For example, the meeting form in the datebook provides the hourly schedule bar and buttons, static objects that are the same for every meeting. The form also provides text fields containing the meeting information, dynamic objects that can have different contents for every meeting, and a time interval drawn on the schedule bar, another object that be different for every meeting.

The most common way to include static objects in your form is by building them in as subviews. When you declare a form in your instance definition file, you specify all the objects you want on the form, including static as well as dynamic objects, as a view chain that is a subview of the form. Magic Cap will draw the form and its subviews when the user goes to a card that uses the form.

To have your form provide a decorative background, specify the background object in the form's *border* field in your instance definition file. When the user goes to a card that uses the form, the background will be drawn along with the form.

Dynamic form objects appear on every card that uses the form, but each card stores its own data for the dynamic form objects. Most classes that are commonly used as dynamic form objects, including text fields, switches, sliders, choice boxes, and all other controls, are designed to retain their per-card data automatically by inheriting the interface of class `FormElement`. For information on creating your own kinds of form elements, see "Creating Your Own Form Elements" on page 103.

The form's *formItems* field refers to an object list that specifies the dynamic form objects that inherit from class `FormElement`. Note that the dynamic form items referred to by the *formItems* field are also listed as subviews of the form.

When the user is finished looking at a card and goes to another card or scene, the card calls its `StoreFormData` operation. `StoreFormData` checks the form's formItems list. If the list contains any objects, `StoreFormData` gets the data from the form objects, stores it as the card's extra data, then empties the form objects so that the next card to use the form won't display another card's data. When a new card is installed in the scene, the card calls `ExtractCardData` to get the values from the card's extra data and place them back in the form objects.

You can easily use dynamic form objects that inherit from class `FormElement`. In your instance definition file, include the dynamic form objects in the form's subview list. In the *formItems* field, specify an object list that consists of the dynamic form objects. If you set up the form this way, the dynamic objects will automatically save and restore their data as the user moves from card to card. The following diagram shows an example of a form.

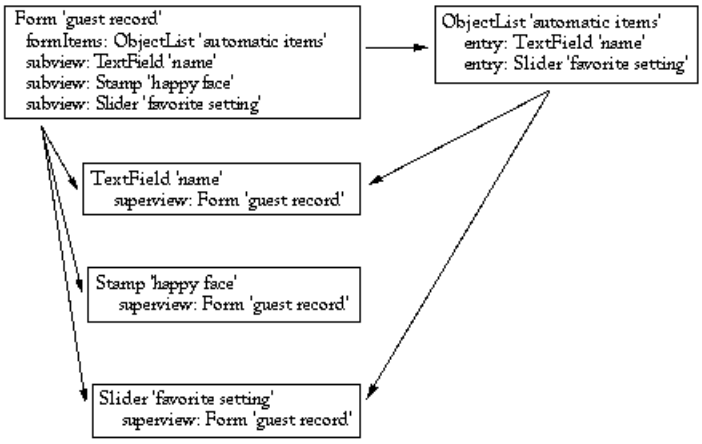


Figure 12 Forms with its subviews and form items

Note that you refer to each dynamic form object at least twice: once as a member of a view chain, referred to by its containing form, and once as an automatically updated dynamic form object, referred to by the form items list. Note also that the form can contain subviews that aren't in the form items list, such as the stamp in the example above. These subviews will be drawn on every card, but won't have card-specific data installed automatically.

You can add viewables to a form at runtime with the Magic Cap simulator. You can get many viewables, including switches, sliders, choice boxes, and text fields, from the Magic Hat. Text fields are available in the tool holder on the control bar. After creating one of these, you can use a *put in form* coupon from the *misc.* drawer of the Magic Hat's *extras* category to put the object in the form. The *extras* category is only available in the Magic Cap simulator, not in communicator versions of Magic Cap.

If the user adds objects that are members of class `FormElement`, Magic Cap adds the objects to the form items list automatically. When the user goes to another card, the card calls `StoreFormData`, which in turn calls `Form_UpdateFormItems`, which then examines all the objects on the form. Items that are members of `FormElement` are added to the form items list if not already in that list.

Forms include an `Image` attribute and associated *image* field. If you specify an image for a form in your instance definition file or at runtime, the image will be drawn in the center of the form whenever the form is installed in the scene.

You can get or change the form items at runtime by using the `FormItems` attribute.

Forms are shared objects, so they are never modified directly. When a card is made the current card, a copy of the form specified in the card's *prototypeForm* field is made and the copy is installed as a subview of a card. This allows the user to change the form without affecting other cards that use the same form. The form referenced from the *prototypeForm* field should always be an indexical.

Although a form is normally installed into the current card, you can use the form to programmatically get at the per-card data in cards other than the current card in a stack. You can install a form into any card by calling `BeginUsingForm`. If you pass `true` as the `needCardData` parameter, `ExtractCardData` will be called to put the per-card data into the form items. `BeginUsingForm` will return `true` if the form was installed, or `false` if a form is already installed in this card. When you are finished using the form with a card that is not current, you should call `EndUsingForm`. You can pass `true` as the `hadCardData` parameter to have the data in the form items stored back into the card. Here is how you might write code to get the data from all the cards in a stack:

```
Private void
GetAllCardData(Reference stack)
{
  ulong     index;
  ulong     count;

  count = Count(stack);
  for (index = 1; index <= count; index++) {
    Reference card;
    Boolean   formInstalled;
    Boolean   dataChanged;

    // Get each card in the stack
    card = At(stack, index);

    // Install this card's form and load up the form items with card data
    formInstalled = BeginUsingForm(card, true);

    // Do what you will with the data. The data is accessed by getting
    // the list of form items on the newly installed form.
    dataChanged = AnalyzeData(FormItems(InstalledForm(card)));

    // When you're done, return the form. There's no need to write the
    // data back to the card if it didn't change.
    if (formInstalled)
      EndUsingForm(card, dataChanged);
  }
}
```

Creating Your Own Form Elements

Magic Cap provides a large collection of classes, including all text fields and all controls, that inherit from class `FormElement` and so can be used as dynamic form objects just by specifying them in the form items list. You can also create your own class that acts as a dynamic form object, storing and retrieving its data as the user moves from card to card.

To create your own class of dynamic form object, inherit the interface of class `FormElement` in your new class's declaration. Then, override the four operations defined by `FormElement`. Override `IsFormElement` to return `true` if you want the object to be treated as a form element. Override `FormData` and `SetFormData` to move the object's contents to and from the form. Override `FormDataInfo` to provide information about the dynamic data.

You can also create a new class of dynamic form object by subclassing an existing dynamic form element class, such as `TextField` or `Control`. In this case, you probably won't have to override the `FormElement` operations to implement your new class.

Stacks and Stack Scenes

You can make a new card in the stack by calling the stack scene's `CreateNewCard` operation. `CreateNewCard` makes a new card in the stack by copying the stack's **prototype card**, which is available through the stack's `ProtoCard` attribute. The prototype card is a model that includes all objects that should be present on newly created cards. You can declare your own prototype card in your instance definition file, or you can use a supplied card, such as `iPrototypeCard`. After the new card is created, Magic Cap goes to the new card. When a new card is created in the stack, it is added after the last card in the stack.

You can remove a card from its stack and install it in another by calling `MoveToStack`. If you pass `nilObject` as the `newStack` parameter, the card will not be part of any stack. You can take the current card out of its stack and remove it from the screen view list by calling `DetachCard`. If you want to destroy the current card completely, not just remove it from the stack, call `DeleteCard`. To delete the current card after getting confirmation from the user, call `DeleteCurrentCardWithConfirmation` on the stack scene.

Many scenes, especially stack scenes, display a column of five buttons along the right side of the screen, between the top and bottom bars. Although many scenes have these buttons, Magic Cap doesn't provide any explicit support for this feature in its scene classes. Instead, scenes simply declare these buttons as subviews in instance definition files. If your scene has five buttons down the right side, you should set each button's *viewFlags* field to `0x10101000` and its *border* field to `iSquareButtonBorderUp`. If your scene uses fewer than five buttons, you can use an instance of class `ButtonBackdrop` which fills in the rest of the column with simulated disabled buttons.

You should use the following x-y coordinate pairs for the buttons' relative origins and content sizes:

button position from top	relative origin	content size
1	<211.0, -104.0>	<50.0, 41.0>
2	<211.0, -53.0>	<50.0, 43.0>
3	<211.0, -1.0>	<50.0, 43.0>
4	<211.0, -51.0>	<50.0, 43.0>
5	<211.0, 102.0>	<50.0, 43.0>

Note: Note that the content size of the first button is <50.0,41.0>, while the size of all others is <50.0,43.0>. You could make the button sizes more uniform while occupying the same total amount of space by setting the sizes of two of them to <50.0,42.0> and the other three to <50.0,43.0>. However, due to the way Magic

Cap draws dithered grays, setting the content size of one button to <50.0,42.0> produces an unattractive dithered line beneath the button. In addition, Magic Cap is optimized to draw buttons with border `iSquareButtonBorderUp` and content size <50.0,43.0> quickly.

When a stack scene is current, the name bar normally displays arrows that the user can touch to move to the next or previous card. When the user goes to the first or last card in the stack, one of the two arrows in the name bar vanishes to show that there are no more cards in that direction. However, you can set a stack to allow the user to treat the cards as if they were laid out in a circle, continuing past the last card to the first card, or past the first to the last. Continuing through the cards in this way is called **wrapping**, because the path through the cards seems to wrap around the edges of the stack, from one end to the other. To turn on wrapping, set the *canWrap* field of the stack's instance definition to `true`, or use the `CanWrap` attribute to set or clear this feature.

Stack scenes usually display arrows in the name bar along with the number of the current card and the total number of cards in the stack. You can suppress that display by setting the *blankTitle* scene field, or by overriding `ShowNameBarInfo`.

The stack of cards refers to its scene with the `StackScene` attribute. The scene refers back to its stack with the `Stack` attribute.

Stacks include a setting that indicates whether there are cards in the stack that haven't been opened by the user. You can get or change this setting by calling `ContainsNewItems` or `SetContainsNewItems`. This feature is used mainly by class `MailStack`, which shows stacks of telecards, as in the in box and out box. `MailStack` is the only Magic Cap class that changes this setting to `true`.

You must call `SetContainsNewItems` to change the flag to `true` in your stack subclass to indicate the presence of new cards if you want to support this feature. The stack scene changes this setting to false when the user goes to the stack. You can use this feature in conjunction with the `WasSeen` and `SetWasSeen` operations to indicate whether the user has looked at a card.

Large Cards and Scrolling

Some cards are too large to fit on the screen completely. You can make a card larger by calling its `ExtendBottom` operation. For example, the notebook calls `ExtendBottom` when the user touches the *extend* button. You can call `RevealTop` if you want to ensure that the top of the card has scrolled into view on the screen, or `RevealBottom` to make sure that the bottom of the card can be seen. If you want to ensure that a particular viewable on the card is visible, you can call `RevealViewable`.

These cards usually have scroll arrows that let the user see the entire card.

Electronic Mail

Your package can create a telecard, then present the telecard to the user in the message scene, which provides standard buttons for addressing and sending the telecard. In addition, you can add other electronic mail features to your package.

Creating Card, Stack, and Form Objects

You'll declare instances of `StackOfCards`, `StackScene`, and `Form` in your instance definition file. You'll probably never create any objects of these classes at runtime. You'll usually declare instances of class `Card` in your instance definition files, but your package may allow the creation of new cards at runtime. Instances of class `MiniCard` are usually created at runtime by copying a prototype object.

Flags and Indexicals

Class `Card` defines various flags that you can use to customize the appearance and behavior of its members. You'll use these flags primarily when you declare cards in instance definition files. These flags are implemented as boolean fields in card objects.

Card flags let you determine information about the card, such as whether it is a postcard, letter, or envelope, whether the card has been mailed, whether the user can change the card, and many other settings. For more information on card flags, see the Utilities.h file.

Class `StackOfCards` defines a few flags that you can use to customize the appearance and behavior of its members. Stack flags determine whether the stack will wrap when the user goes past the first or last card, whether the stack can be deleted, and other settings. You'll use these flags primarily when you declare stacks in instance definition files. For more information on stack flags, see the Utilities.h file.

Magic Cap provides many indexicals that contain useful cards and stacks or information about cards and stacks. Some indexicals, such as `iReplyCard` and `iForwardCard` specify prototype cards for various uses. The `iCurrentSceneStack` is the stack associated with the current scene. Several indexicals provide access to the most common stacks in Magic Cap, such as `iInBoxStack`, `iNameCardsStack`, and `iNoteCardStack`.

See the Indexicals.cdef file for more information on indexicals.

Card and Stack Information in the Package Content Object

When you declare a stack and its cards in your instance definition file, you should list the cards individually in the package content object's install list, with their stack as the receiver. In addition, you should define indexicals for all your package's stacks and list them in the *stackIndexicalsList* field of the package content object. By doing putting your cards in the install list and your stacks in the stack indexicals list, Magic

Cap can handle the cards properly if the user moves the cards out of the package and then packs up the package or removes the storage card containing the package, or if the cards somehow become disconnected from the package.

For more information on items in the package content object, see the Software Packages chapter of this book.

Subclasses of Card and Stack

Magic Cap includes many subclasses of `Card` that specialize its behavior in various ways. Many built-in scenes in Magic Cap uses card subclasses to display their information. For example, book pages in the library, folders in the file cabinet and elsewhere, and name cards are all card subclasses. Because these classes are specialized for their particular uses, you'll probably use class `Card` or create your own subclasses.

There are a few highly specialized subclasses of `StackOfCards` in Magic Cap, such as `MailStack`, used for a collection of telecards, and `NameCardStack`, used exclusively by the name card file. You might use class `SortedStackOfCards`, which mixes in `SortedList` to create a stack whose cards are always in order.