class_addMethods() → See class_getInstanceMethod()

class_createInstance(), class_createInstanceFromZone()

SUMMARY Create a new instance of a class

DECLARED IN objc/objc-class.h

SYNOPSIS id **class_createInstance(**Class *aClass*, unsigned int *indexedIvarBytes***)**
id **class_createInstanceFromZone(**Class *aClass*, unsigned int *indexedIvarBytes*, NXZone ***zone***)**

DESCRIPTION These functions provide an interface to the object allocators used by the run-time system. The default allocators, which can be changed by reassigning the **_alloc** and **_zoneAlloc** variables, create a new instance of *aClass* by dynamically allocating memory for it, initializing its **isa** instance variable to point to the class, and returning the new instance. All other instance variables are initialized to 0.

The two functions are identical, except that **class_createInstanceFromZone()** allocates memory for the new object from the region specified by *zone* and **class_createInstance()** allocates memory from the default zone returned by **NXDefaultMallocZone()**.

Object's **alloc** and **allocFromZone:** methods use **class_createInstanceFromZone()** to allocate memory for a new object, with **alloc** taking the memory from the default zone. The **new** method uses **class_createInstance()**.

The second argument to both functions, *indexedIvarBytes*, states the number of extra bytes required for indexed instance variables. Normally, it's 0.

Indexed instance variables are instance variables that are not declared or accounted for in the usual way, generally because they don't have a fixed size. Usually they're arrays whose length can't be computed at compile time. Since the components of a C structure can't be of uncertain size, indexed instance variables can't be declared in the class interface. The class must account for them outside the normal channels provided by the Objective C language.

All of the storage required for indexed instance variables must be allocated through one of these two functions. The following code shows how they might be used in an instance-creating class method:

```
+ new:(unsigned int)numBytes
{
    self = class_createInstance((Class)self, numBytes);
    length = numBytes;
    . . .
}

- (char *)getArray
{
    return(object_getIndexedIvars(self));
}
```

Indexed instance variables should be avoided if at all possible. It's a much better practice to store variable-length data outside the object and declare one real instance variable that points to it and perhaps another that records its length. For example:

```
+ new:(unsigned int)numBytes
{
```

```
        self = [super new];
        data = malloc(numBytes);
        length = numBytes;
        . . .
    }

    - (char *)getArray
    {
        return data;
    }
```

RETURN If successful, both **class_createInstance()** and **class_createInstanceFromZone()** return the new instance of *aClass*. If not successful, they generate an error message and call **abort()**.

class_createInstanceFromZone() → See class_createInstance()

class_getClassMethod() → See class_getInstanceMethod()

class_getInstanceMethod(), class_getClassMethod(), class_addMethods(), class_removeMethods()

SUMMARY Get, add, and remove methods

DECLARED IN objc/objc-class.h

SYNOPSIS Method **class_getInstanceMethod(**Class *aClass*, SEL *aSelector***)**
Method **class_getClassMethod(**Class *aClass*, SEL *aSelector***)**
void **class_addMethods(**Class *aClass*, struct objc_method_list **methodList***)**
void **class_removeMethods(**Class *aClass*, struct objc_method_list **methodList***)**

DESCRIPTION The first two functions, **class_getInstanceMethod()** and **class_getClassMethod()**, return a pointer to the class data structure that describes the *aSelector* method. For **class_getInstanceMethod()**, *aSelector* must identify an instance method; for **class_getClassMethod()**, it must identify a class method. Both functions return a NULL pointer if *aSelector* doesn't identify a method defined in or inherited by *aClass*.

The run-time system uses the next function, **class_addMethods()**, to implement Objective C categories. Each function adds the methods in *methodList* to the dictionary of methods defined for *aClass*. To add methods that can be used by instances of a class, *aClass* should be the class object. To add methods that can be used by a class object, *aClass* should be the metaclass object (the **isa** field of the Class structure). All the methods in *methodList* must be mapped to valid SEL selectors before they're added to the class. The **sel_registerName()** function can be used to accomplish this.

The last function, **class_removeMethods()**, removes methods that were previously added using **class_addMethods()**. The run-time system uses it to unload categories that were dynamically loaded at an earlier point in time. Its second argument, *methodList*, must be identical to a pointer previously passed to **class_addMethods()**. To remove instance methods, *aClass* should be the class object. To remove class methods, *aClass* should be the **isa** field of the Class structure.

RETURN **class_getInstanceMethod()** and **class_getClassMethod()** return a pointer to the data structure that describes the *aSelector* method as implemented for *aClass*. If *aSelector* isn't defined for *aClass*, they return NULL.

class_getInstanceVariable()

SUMMARY Get the class template for an instance variable

DECLARED IN objc/objc-class.h

SYNOPSIS Ivar **class_getInstanceVariable(**Class *aClass*, const char **variableName***)*

RETURN This function returns a pointer to the class data structure that describes the *variableName* instance variable. If *aClass* doesn't define or inherit the instance variable, a NULL pointer is returned.

class_getVersion() → See class_setVersion()

class_poseAs()

SUMMARY Pose as the superclass

DECLARED IN objc/objc-class.h

SYNOPSIS Class **class_poseAs(**Class *theImposter*, Class *theSuperclass*)

DESCRIPTION **class_poseAs()** causes one class, *theImposter*, to take the place of its own superclass, *theSuperclass*. Messages sent to *theSuperclass* will actually be received by *theImposter*. The posing class can't declare any new instance variables, but it can define new methods and override methods defined in the superclass.

Posing is usually done through Object's **poseAs:** method, which calls this function.

RETURN Normally, **class_poseAs()** returns its first argument, *theImposter*. However, if *theImposter* defines instance variables or is not a subclass of (or the same as) *theSuperclass*, it generates an error message and aborts.

class_removeMethods() → See class_getInstanceMethod()

class_setVersion(), class_getVersion()

SUMMARY Set and get the class version

DECLARED IN objc/objc-class.h

SYNOPSIS void **class_setVersion(**Class *aClass*, int *versionNumber*)
int **class_getVersion(**Class *aClass*)

DESCRIPTION These functions set and return the class version number. This number is used when archiving instances of the class.

Object's **setVersion:** and **version** methods do the same work as these functions.

RETURN **class_getVersion()** returns the version number for *aClass* last set by **class_setVersion()**, or 0 if no version has been set.

marg_getRef() → See marg_getValue()

marg_getValue(), marg_getRef(), marg_setValue()

SUMMARY Examine and alter method argument values

DECLARED IN objc/objc-class.h

SYNOPSIS *type-name* **marg_getValue(**marg_list *argFrame*, int *offset*, *type-name***)**
type-name ***marg_getRef(**marg_list *argFrame*, int *offset*, *type-name***)**
void **marg_setValue(**marg_list *argFrame*, int *offset*, *type-name*, *type-name value***)**

DESCRIPTION These three macros get and set the values of arguments passed in a message. They're designed to be used within implementations of the **forward::** method, which is described under the Object class in Chapter 1, ªRoot Class.º

The first argument to each macro, *argFrame*, is a pointer to the list of arguments passed in the message. The run-time system passes this pointer to the **forward::** method, making it available to be used in these macros. The next two argumentsÐan *offset* into the argument list and the type of the argument at that offsetÐcan be obtained by calling **method_getArgumentInfo()**.

The first macro, **marg_getValue**, returns the argument at *offset* in *argFrame*. The return value, like the argument, is of type *type-name*. The second macro, **marg_getRef**, returns a reference to the argument at *offset* in *argFrame*. The pointer returned is to an argument of the *type-name* type. The third macro, **marg_setValue**, alters the argument at *offset* in *argFrame* by assigning it *value*. The new value must be of the same type as the argument.

Because these are macros, the *type-name* must be written as types normally are in source code; it can't be passed as a variable. Therefore, if the type is obtained from **method_getArgumentInfo()**, a **switch** statement would be required to select the correct macro call from a list of predetermined choices. **method_getArgumentInfo()** encodes the argument type according to the conventions of the **@encode()** compiler directive.

RETURN **marg_getValue** returns a *type-name* argument value. **marg_getRef** returns a pointer to a *type-name* argument value.

marg_setValue() → See marg_getValue()

method_getArgumentInfo() → See method_getNumberOfArguments()

method_getNumberOfArguments(), method_getSizeOfArguments(), method_getArgumentInfo()

SUMMARY Get information about a method

DECLARED IN objc/objc-class.h

SYNOPSIS unsigned int **method_getNumberOfArguments(**Method *aMethod***)**
unsigned int **method_getSizeOfArguments(**Method *aMethod***)**
unsigned int **method_getArgumentInfo(**Method *aMethod*, int *index*, const char ****type*, int **offset***)**

DESCRIPTION The three functions described here all provide information about the argument structure of a particular method. They take as their first argument the method's data structure, *aMethod*, which can be obtained by calling **class_getInstanceMethod()** or **class_getClassMethod()**.

The first function, **method_getNumberOfArguments()**, returns the number of arguments that *aMethod* takes. This will be at least two, since it includes the ªhiddenº arguments, **self** and **_cmd**, which are the first two arguments passed to every method implementation.

The second function, **method_getSizeOfArguments()**, returns the number of bytes that all of *aMethod*'s arguments, taken together, would occupy on the stack. This information is required by **objc_msgSendv()**.

The third function, **method_getArgumentInfo()**, takes an *index* into *aMethod*'s argument list and returns, by reference, the type of the argument and the offset to the location of that argument in the list. Indices begin with 0. The ªhiddenº arguments **self** and **_cmd** are indexed at 0 and 1; method-specific arguments begin at index 2. If *index* is too large for the actual number of arguments, the *type* aand *offset* pointers are set to NULL. Otherwise, the offset is measured in bytes; it depends entirely on the size of arguments preceding the one at *index*. The type is encoded according to the conventions of the **@encode()** compiler directive.

The information obtained from **method_getArgumentInfo()** can be used in the **marg_getValue**, **marg_getRef**, and **marg_setValue** macros to examine and alter the values of an argument on the stack after *aMethod* has been called. The offset can be passed directly to these macros, but the type must first be decoded to a full type name.

RETURN **method_getNumberOfArguments()** returns how many arguments the implementation of *aMethod* takes, and **method_getSizeOfArguments()** returns how many bytes the arguments take up on the stack. **method_getArgumentInfo()** returns the *index* it is passed.

method_getSizeOfArguments() → See method_getNumberOfArguments()

objc_addClass() → See objc_getClass()

objc_getClass(), objc_lookUpClass(), objc_getMetaClass(), objc_getClasses(), objc_addClass(), objc_getModules()

SUMMARY Manage run-time structures

DECLARED IN objc/objc-runtime.h

SYNOPSIS id **objc_getClass(**const char ***aClassName**)
id **objc_lookUpClass(**const char ***aClassName**)
id **objc_getMetaClass(**const char ***aClassName**)
NXHashTable ***objc_getClasses(**void**)**
void **objc_addClass(**Class *aClass*)
Module ***objc_getModules(**void**)**

DESCRIPTION These functions return and modify the principal data structures used by the run-time system.

The first two functions, **objc_getClass()** and **objc_lookUpClass()**, both return the **id** of the class object for the *aClassName* class. However, if the *aClassName* class isn't known to the run-time system, **objc_getClass()** prints a message to the standard error stream and causes the process to abort, while **objc_lookUpClass()** merely returns **nil**.

The third function, **objc_getMetaClass()**, returns the **id** of the metaclass object for the *aClassName*

class. The metaclass object holds information used by the class object, just as the class object holds information used by instances of the class. Like **objc_getClass()**, it prints a message to the standard error stream and causes the process to abort if *aClassName* isn't a valid class.

objc_getClasses() returns a pointer to the hash table containing all the Objective C classes that are currently known to the run-time system. You can examine the table using the common hashing functions. In the following example, **NXNextHashState()** gets each class from the table in turn, and **object_getClassName()** asks for their names:

```
NXHashTable *classes = objc_getClasses();
NXHashState state = NXInitHashState(classes);
Class thisClass;

while ( NXNextHashState(classes, &state, (void **)&thisClass) )
    fprintf(stderr, "%s\n", object_getClassName((id)thisClass));
```

The NXHashTable type returned by **objc_getClasses()** is defined in the **objc/hashtable.h** header file and is documented in Chapter 3, ªCommon Classes and Functions.º This data structure can be read, as illustrated in the example above, but it should not be modified or freed.

objc_addClass() adds *aClass* to the list of classes known to the run-time system. (The class is added to the hash table that **objc_getClasses()** returns.)

The compiler creates a Module data structure for each file it compiles. The **objc_getModules()** function returns a pointer to the run-time system's list of all current modules, except those that were dynamically loaded. Module structures are described under ªSupporting Header Filesº later in this chapter.

RETURN **objc_lookUpClass()** returns the class object for *aClassName*, or **nil** if there is no such class. **objc_getClass()** and **objc_getMetaClass()** return the class and metaclass objects for *aClassName*, if such a class exists, and abort otherwise. **objc_getClasses()** returns a pointer to a hash table of all current classes, and **objc_getModules()** returns a pointer to all current modules.

objc_getClasses() → See objc_getClass()

objc_getMetaClass() → See objc_getClass()

objc_getModules() → See objc_getClass()

objc_loadModules(), objc_unloadModules()

SUMMARY Dynamically load and unload classes

DECLARED IN objc/objc-load.h

SYNOPSIS long **objc_loadModules(**char **files*[], NXStream **stream*, void (**callback*)(Class, Category), struct mach_header ***header*, char **debugFilename***)**
long **objc_unloadModules(**NXStream **stream*, void (**callback*)(Class, Category)**)**

DESCRIPTION **objc_loadModules()** dynamically loads object files containing Objective C class and category definitions into a running program. Its first argument, *files*, is a list of null-terminated pathnames for the object files containing the classes and categories that are to be loaded. They can be full paths or paths relative to the current working directory. The second argument, *stream*, is a pointer to an NXStream where any error messages produced by the loader will be written. It can be NULL, in which case no messages will be written.

The third argument, *callback*, allows you to specify a function that will be called immediately after each class or category is loaded. When a category is loaded, the function is passed both the **Category** structure and the **Class** structure for that category. When a class is loaded, it's passed only the **Class** structure. Like *stream*, *callback* can be NULL.

The fourth argument, *header*, is used to get a pointer to the **mach_header** structure for the loaded modules. It, too, can be NULL. All the modules in *files* are grouped under the same header.

The final argument, which also can be NULL, is the pathname for a file that the loader will create and initialize with a copy of the loaded modules. This file can be passed to the debugger and added to the list of files being debugged. For example:

```
(gdb) add-file debugFilename
```

obj_unloadModules() unloads all the modules loaded by **objc_loadModules()**, that is, all the modules from the *files* list. Each time it's called, it unloads another set of modules, working its way back from the modules loaded by the most recent call to **objc_loadModules()** to those loaded by the next most recent call, and so on.

The first argument to **obj_unloadModules()**, *stream*, is a pointer to an NXStream where error messages will be written. Its second argument, *callback*, allows you to specify a function that will be called immediately before each class or category is unloaded. Both arguments can be NULL.

Note: The NXBundle class, documented in Chapter 3, ªCommon Classes and Functions,º provides a simpler and preferred way to dynamically load classes. NXBundle integrates dynamic loading with localization (using language-specific resources such as strings, images, and sounds).

RETURN Both functions return 0 if the modules are successfully loaded or unloaded and 1 if they're not.

objc_lookUpClass() → See objc_getClass()

objc_msgSend(), objc_msgSendSuper(), objc_msgSendv()

SUMMARY Send messages at run time

DECLARED IN objc/objc-runtime.h

SYNOPSIS id **objc_msgSend(**id *theReceiver*, SEL *theSelector*, ...**)**
id **objc_msgSendSuper(**struct objc_super ***superContext*, SEL *theSelector*, ...**)**
id **objc_msgSendv(**id *theReceiver*, SEL *theSelector*, unsigned int *argSize*, marg_list *argFrame***)**

DESCRIPTION The compiler converts every message expression into a call on one of the first two of these three functions. Messages to **super** are converted to calls on **objc_msgSendSuper()**; all others are converted to calls on **objc_msgSend()**.

Both functions find the implementation of the *theSelector* method that's appropriate for the receiver of the message. For **objc_msgSend()**, *theReceiver* is passed explicitly as an argument. For **objc_msgSendSuper()**, *superContext* defines the context in which the message was sent, including who the receiver is.

Arguments that are included in the *aSelector* message are passed directly as additional arguments to both functions.

Calls to **objc_msgSend()** and **objc_msgSendSuper()** should be generated only by the compiler. You shouldn't call them directly in the Objective C code you write. You can however use the

Object instance method **performv::** to send an arbitrary message to an object..

The third function, **objc_msgSendv()**, is an alternative to **objc_msgSend()** that's designed to be used within class-specific implementations of the **forward::** method. Instead of being passed each of the arguments to the *aSelector* message, it takes a pointer to the arguments list, *argFrame*, and the size of the list in bytes, *argSize*. *argSize* can be obtained by calling **method_getSizeOfArguments()**; *argFrame* is passed as the second argument to the **forward::** method.

objc_msgSendv() parses the argument list based on information stored for *aSelector* and the class of the receiver. Because of this additional work, it's more expensive than **objc_msgSend()**.

RETURN Each method passes on the value returned by the *aSelector* method.

objc_msgSendSuper() → See objc_msgSend()

objc_msgSendv() → See objc_msgSend()

objc_setMultithreaded()

SUMMARY Make the run-time system thread safe

DECLARED IN objc/objc-runtime.h

SYNOPSIS void **objc_setMultithreaded(**BOOL *flag***)**

DESCRIPTION When *flag* is YES, this function ensures that two or more threads of the same task can safely use the run-time system for Objective C. To guarantee correct run-time behavior, it should be called immediately before starting up a new thread.

Because of the additional checking required to ensure thread-safe behavior, messaging will be slower than normal. Therefore, *flag* should be reset to the default NO when there is only one thread using Objective C.

This function cannot guarantee that all parts of the run-time system are absolutely thread-safe. In particular, if one thread is in the middle of dynamically loading or unloading a class (using **objc_loadModules()** or **objc_unloadModules()**) while another thread is using the class, the second thread might find the class in an inconsistent state. Similarly, a thread that gets the class hash table (using **objc_getClasses()**) cannot be sure that another thread won't be modifying it at the same time.

objc_unloadModules() → See objc_loadModules()

object_copy() → See object_dispose()

object_copyFromZone() → See object_dispose()

object_dispose(), object_copy(), object_copyFromZone(), object_realloc(), object_reallocFromZone()

SUMMARY Manage object memory

DECLARED IN objc/Object.h

SYNOPSIS id **object_dispose(**Object *_anObject_**)**
id **object_copy(**Object *_anObject_, unsigned int _indexedIvarBytes_**)**
id **object_copyFromZone(**Object *_anObject_, unsigned int _indexedIvarBytes_, NXZone *_zone_**)**
id **object_realloc(**Object *_anObject_, unsigned int _numBytes_**)**
id **object_reallocFromZone(**Object *_anObject_, unsigned int _numBytes_, NXZone *_zone_**)**

DESCRIPTION These five functions, along with **class_createInstance()** and **class_createInstanceFromZone()**, manage the dynamic allocation of memory for objects. Like those two functions, each of them is simply a ªcoverº forÐa way of callingÐanother, private function.

object_dispose() frees the memory occupied by *anObject* after setting its **isa** instance variable to **nil**, and returns **nil**. The function it calls to do this work can be changed by reassigning the **_dealloc** variable.

object_copy() and **object_copyFromZone()** create a new object that's an exact copy of *anObject* and return the new object. **object_copy()** allocates memory for the copy from the same zone as the original; **object_copyFromZone()** places the copy in *zone*. The second argument to both functions, *indexedIvarBytes*, specifies the number of additional bytes that should be allocated to accommodate indexed instance variables; it serves the same purpose as the second argument to **class_createInstance()**.

The functions that **object_copy()** and **object_copyFromZone()** call to do their work can be changed by reassigning the **_copy** and **_zoneCopy** variables.

object_realloc() and **object_reallocFromZone()** reallocate storage for *anObject*, adding *numBytes* if possible. The memory previously occupied by *anObject* is freed if it can't be reused, and a pointer to the new location of *anObject* is returned. **object_realloc()** allocates memory for the object from the same zone that it originally occupied; **object_reallocFromZone()** locates the object in *zone*.

The functions that **object_realloc() and object_reallocFromZone()** call to do their work can be changed by reassigning the **_realloc** and **_zoneRealloc** variables.

RETURN **object_dispose()** returns **nil**, **object_copy()** and **object_copyFromZone()** return the copy, and **object_realloc()** and **object_reallocFromZone()** return the reallocated object. If the attempt to copy or reallocate the object fails, an error message is generated and **abort()** is called.

object_getClassName()

SUMMARY Return the class name

DECLARED IN objc/objc.h

SYNOPSIS const char ***object_getClassName(**id _anObject_**)**

DESCRIPTION This function returns the name of *anObject*'s class, or the string ªnilº if *anObject* is **nil**. *anObject* can be either an instance or a class object.

object_getIndexedIvars()

SUMMARY Return a pointer to an object's extra memory

DECLARED IN objc/objc.h

SYNOPSIS void ***object_getIndexedIvars(**id *anObject***)**

This function returns a pointer to the first indexed instance variable of *anObject*, if *anObject* has indexed instance variables. If not, the pointer returned won't be valid and should not be used.

SEE ALSO **class_createInstance()**

object_getInstanceVariable() → See object_setInstanceVariable()

object_realloc() → See object_dispose()

object_reallocFromZone() → See object_dispose()

object_setInstanceVariable(), object_getInstanceVariable()

SUMMARY Set and get instance variables

DECLARED IN objc/Object.h

SYNOPSIS Ivar **object_setInstanceVariable(**id *anObject*, const char **variableName*, void **value***)*
Ivar **object_getInstanceVariable(**id *anObject*, const char **variableName*, void ***value***)**

DESCRIPTION **object_setInstanceVariable()** assigns a new value to the *variableName* instance variable belonging to *anObject*. The instance variable must be one that's declared as a pointer; typically it's an **id**. The new value of the pointer is passed in the third argument, *value*. (Note that the pointer value is passed directly, not by reference.)

object_getInstanceVariable() gets the value of the pointer stored as *anObject*'s *variableName* instance variable. The pointer is returned by reference through the third argument, *value*. For example:

```
int *i;
Ivar var = object_getInstanceVariable(anObject, "num", (void **)&i);
```

These functions provide a way of setting and getting instance variables that are declared as pointers, without having to implement methods for that purpose. For example, Interface Builder calls **object_setInstanceVariable()** to initialize programmer-defined ªoutletº instance variables.

These functions cannot reliably be used to set and get instance variables that are not pointers.

RETURN Both functions return a pointer to the class template that describes the *variableName* instance variable. A NULL pointer is returned if *anObject* has no instance variable with that name.

The returned template has a field describing the data type of the instance variable. You can check it to be sure that the value set is of the correct type.

sel_getName() → See sel_getUid()

sel_getUid(), sel_getName()

SUMMARY Match method names with method selectors

DECLARED IN objc/objc.h

SYNOPSIS SEL **sel_getUid(**const char ***aName**)
const char ***sel_getName(**SEL *aSelector***)**

DESCRIPTION The first function, **sel_getUid()**, returns the unique identifier that represents the *aName* method at run time. The identifier is a selector (type SEL) and is used in place of the method name in compiled code; methods with the same name have the same selector. Whenever possible, you should use the **@selector()** directive to ask the compiler to provide the selector for a method. This function asks the run-time system for the selector and should be used only if the name isn't known at compile time.

The second function, **sel_getName()**, is the inverse of the first. It returns the name that was mapped to *aSelector*.

RETURN **sel_getUid()** returns the selector for the *aName* method, or 0 if there is no known method with that name. **sel_getName()** returns a character string with the name of the method identified by the *aSelector* selector. If *aSelector* isn't a valid selector, a NULL pointer is returned.

sel_isMapped()

SUMMARY Determine whether a selector is valid

DECLARED IN objc/objc.h

SYNOPSIS BOOL **sel_isMapped(**SEL *aSelector***)**

RETURN **sel_isMapped()** returns YES if *aSelector* is a valid selector (is currently mapped to a method implementation) or could possibly be one (because it lies within the same range as valid selectors); otherwise it returns NO.

Because all of a program's selectors are guaranteed to be mapped at start-up, this function has little real use. It's included here for reasons of backward compatibility only.

sel_registerName()

SUMMARY Register a method name

DECLARED IN objc/objc.h

SYNOPSIS SEL **sel_registerName(**const char ***aName**)

DESCRIPTION This function registers *aName* as a method name and causes it to be mapped to a SEL selector, which it returns.

No check is made to see if *aName* is already a valid method name. If it is, the same name will be mapped to more than one selector. When the run-time system needs to match a selector to the name, it's indeterminant which one it will find.

RETURN **sel_registerName()** returns the selector it maps to the *aString* method.