NSData Class Cluster

Class Description

NSData objects provide an object-oriented wrapper for byte buffers. This enables simple allocated buffers (that is, data with no embedded pointers) to take on the behavior of Foundation Kit objects. NSData is typically used for data storage. It is also useful in Distributed Objects applications, where data contained in NSData objects can be copied or moved between applications.

NSData can be used to wrap data of any size. When the data size is over a few pages, NSData uses virtual memory management. NSData can also be used to wrap pre-existing data, regardless of how the data was allocated. NSData contains no information about the data itself (such as its type); the responsibility for deciding how to use the data lies with the client. In particular, it will not handle byte-order swapping when distributed between big-endian and little-endian machines. For typed data, use NSValue.

NSData provides an operating system-independent way to benefit from copy-on-write memory. The copy-on-write technique means that when data is copied through a virtual memory copy (using **vm_copy()**), an actual copy of the data is not made until there is an attempt to modify it. NSData uses either **bcopy()** (byte copy) or **vm_copy()** to copy data, depending on the data's size. For large amounts of data, **vm_copy()** is used. For more information about **vm_copy()**, see Chapter 1 of *NEXTSTEP Operating System Software*.

The cluster's two public classes, NSData and NSMutableData, declare the programmatic interface for static and dynamic NSData objects, respectively.

The objects you create using these classes are referred to as *data objects*. Because of the nature of class clusters, data objects are not actual instances of the NSData or NSMutableData classes but of one of their private subclasses. Although a data object's class is private, its interface is public, as declared by these abstract superclasses, NSData and NSMutableData. (See "Class Clusters" in the introduction to the Foundation Kit for more information on class clusters and creating subclasses within a cluster.)

Generally, you instantiate a data object by sending one of the **data...** messages to either the NSData or NSMutableData class object. These methods return a data object containing or copying the bytes you pass in as arguments. Depending on the method used to instantiate the object, a copy of the bytes may be created and added to the receiver before the data object is instantiated. This means that when the data object is released, a copy of the bytes it contains continues to exist. Alternatively, you can instantiate a data object with a method whose name includes "NoCopy," such as **initWithBytesNoCopy:length:**. In that case, no copy of the bytes remains when the original bytes are freed along with the data object that contains them.

The NSData classes adopt the NSCopying and NSMutableCopying protocols, making it convenient to convert between efficient, read-only data objects and mutable data objects.

▶ NSData

Inherits From:	NSObject
Conforms To:	NSCopying
	NSMutableCopying
Declared In:	foundation/NSData.h

Class Description

The NSData class declares the programmatic interface to an object that contains data in the form of bytes. NSData's two primitive methods—**bytes** and **length**—provide the basis for all the other methods in its interface. The **bytes** method returns a pointer to the bytes contained in the data object. **length** returns the number of bytes contained in the data object.

NSData provides access methods for copying bytes from a data object into a specified buffer. **getBytes** copies all of the bytes into a buffer, whereas **getBytes:length:** copies bytes into a buffer of length *length*. **getBytes:range:** copies a range of bytes from a starting point within the bytes themselves. You can also return a data object that contains a subset of the bytes in another data object by using the **subdataWithRange:** method. Or, you can use the **description** method to return an NSString representation of the bytes in a data object.

For determining if two data objects are equal, NSData provides the **isEqualToData:** method, which does a byte-for-byte comparison.

The **writeToFile:atomically:** method enables you to write the contents of a data object to a file.

Instance Variables

None declared in this class.

Adopted Protocols

NSCopying	– copy
	– copyWithZone:

| NSMutableCopying | – mutableCopy |
| | – mutableCopyWithZone: |

Method Types

Allocating and initializing	+ allocWithZone:
	+ data
	+ dataWithBytes:length:
	+ dataWithBytesNoCopy:length:
	+ dataWithContentsOfFile:
	+ dataWithContentsOfMappedFile
	– initWithBytes:length:
	– initWithBytesNoCopy:length:
	– initWithContentsOfFile:
	– initWithContentsOfMappedFile:
	– initWithData:
Accessing data	– bytes
	– description
	– getBytes:
	– getBytes:length:
	– getBytes:range:
	– subdataWithRange:
Testing data	– isEqualToData:
	– length
Storing data	– writeToFile:atomically:

Class Methods

allocWithZone

+ **allocWithZone:**(NSZone *)*zone*

Creates and returns an uninitialized data object in the specified zone. If the receiver is the NSData class object, an instance of the appropriate immutable subclass is returned; otherwise, an object of the receiver's class is returned.

Typically, you create dictionary objects using the **data...** class methods, not the **alloc...** and **init...** methods. Note that it's your responsibility to release (with either **release** or **autorelease**) those objects created with the **alloc...** methods.

data

+ **data**

Creates and returns an empty data object. This method is declared primarily for the use of mutable subclasses of NSData.

dataWithBytes:length:

+ **dataWithBytes:**(const void *)*bytes* **length:**(unsigned)*length*

Creates and returns a data object containing *length* bytes copied from the buffer *bytes*. If page alignment is being used and if the data size is more than a few pages, this method performs an efficient virtual copy using **vm_copy()**.

See also: – **dataWithBytesNoCopy:length:**

dataWithBytesNoCopy:length:

+ **dataWithBytesNoCopy:**(void *)*bytes* **length:**(unsigned)*length*

Creates and returns a data object containing *length* bytes from the buffer *bytes*.

See also: – **dataWithBytes:length:**

dataWithContentsOfFile:

+ **dataWithContentsOfFile:**(NSString *)*path*

Creates and returns a data object by reading every byte from the file specified by *path*.

For example, this excerpt creates a data object *myData* initialized with the contents of **myFile.txt**. The path must be absolute.

```
NSString* thePath = @"/u/smith/myFile.txt";
NSData *myData;
myData = [NSData dataWithContentsOfFile:thePath];
```

See also: – **dataWithContentsOfMappedFile:**

dataWithContentsOfMappedFile:

    + **dataWithContentsOfMappedFile:**(NSString *)*path*

Creates and returns a data object from the mapped file specified by *path*. Because of file mapping restrictions, this method should only be used if the file is guaranteed to exist for the duration of the data object's existence. It is generally safer to use the **dataWithContentsOfFile:** method.

This methods assumes that mapped files are available on the underlying operating system. A mapped file uses virtual memory techniques to avoid copying pages of the file into memory until they are actually needed.

See also: – **dataWithContentsOfFile:**

Instance Methods

bytes

    – (const void *)**bytes**

Returns a pointer to the data object's contents. This method returns read-only access to the data.

See also: – **description**, – **getBytes:**, – **getBytes:length:**, – **getBytes:range:**

description

    – (NSString *)**description**

Returns an NSString object that contains a hexadecimal representation of the receiver's contents. This string can be read by the ASCII parser.

See also: – **bytes**

getBytes:

    – (void)**getBytes:**(void *)*buffer*

Copies a data object's contents into *buffer*.

For example, this excerpt initializes a data object *myData* with the NSString *myString*. It then copies the contents of *myData* into *aBuffer*.

```
NSData *myData;
unsigned char aBuffer[20];
```

```
NSString* myString = @"Test string.";
myData = [NSData
    dataWithBytes:[myString cString]
    length:[myString length]];

[myData getBytes:aBuffer];
```

See also: – **bytes:**, – **getBytes:length:**, – **getBytes:range:**

getBytes:length:

   – (void)**getBytes:**(void *)*buffer* **length:**(unsigned)*length*

Copies *length* bytes from a data object into *buffer*.

See also: – **bytes:**, – **getBytes:**, – **getBytes:range:**

getBytes:range:

   – (void)**getBytes:**(void *)*buffer* **range:**(NSRange)*range*

Copies the a data object's contents into *buffer*, from a range *range* that is within the bytes
in the object. If *range* isn't within the receiver's range of bytes, an NSRangeException error
is raised.

See also: – **bytes:**, – **getBytes:**, – **getBytes:length:**

hash

   @protocol NSObject
   – (unsigned int)**hash**

Returns an unsigned integer that can be used as a table address in a hash table structure. For
a data object, **hash** returns the length of the data object. If two data objects are equal (as
determined by the **isEqual:** method), they have the same hash value.

See also: – **isEqual:**

initWithBytes:length:

– **initWithBytes:**(const void *)*bytes* **length:**(unsigned)*length*

Initializes a newly allocated data object by adding to it *length* bytes of data copied from the buffer *bytes*.

See also: – **initWithBytesNoCopy:length:**

initWithBytesNoCopy:length:

– **initWithBytesNoCopy:**(void *)*bytes* **length:**(unsigned)*length*

Initializes a newly allocated data object by adding to it *length* bytes of data from the buffer *bytes*.

See also: – **initWithBytes:length:**

initWithContentsOfFile:

– **initWithContentsOfFile:**(NSString *)*path*

Initializes a newly allocated data object by reading into it the data from the file specified by *path*. This method invokes **initWithData:** as part of its implementation.

See also: – **initWithContentsOfMappedFile:**

initWithContentsOfMappedFile:

– **initWithContentsOfMappedFile:**(NSString *)*path*

Initializes a newly allocated data object by reading into it the mapped file specified by *path*. This method invokes **initWithData:** as part of its implementation.

See also: – **initWithContentsOfFile:**

initWithData:

– **initWithData:**(NSData *)*data*

Initializes a newly allocated data object by placing in it the contents of another data object, *data*.

isEqual:

@protocol NSObject
– (BOOL)**isEqual:***anObject*

Returns YES if the receiver and *anObject* are equal; otherwise returns NO. A YES return value indicates that the receiver and *anObject* both inherit from NSData and contain the same data (as determined by the **isEqualToData:** method).

See also: – **isEqualToData:**

isEqualToData:

– (BOOL)**isEqualToData:**(NSData *)*other*

Compares the receiving data object to *other*. If the contents of *other* are equal to the contents of the receiver, this method returns YES. If not, it returns NO. Two data objects are equal if they hold the same number of bytes, and if the bytes at the same position in the objects are the same.

length

– (unsigned)**length**

Returns the number of bytes contained in a data object.

subdataWithRange:

– (NSData *)**subdataWithRange:**(NSRange)*range*

Returns a data object containing a copy of the receiver's bytes that fall within the limits specified by *range*. If *range* isn't within the receiver's range of bytes, an NSRangeException error is raised.

For example, this excerpt initializes a data object, *data2*, to contain a sub-range of *data1*:

```
NSString* myString = @"ABCDEFG";
NSRange range = {2, 4};
NSData *data1, *data2;

data1 = [NSData
    dataWithBytes:[myString cString]
    length:[myString length]];

data2 = [data1 subdataWithRange:range];
```

The result of this excerpt is that *data2* contains CDEF.

writeToFile:atomically:

– (BOOL)**writeToFile:**(NSString *)*path* **atomically:**(BOOL)*useAuxiliaryFile*

Writes the bytes in a data object to the file specified by *path*.

If you provide a value of YES for **atomically:**, the data is written to a backup file and then, assuming no errors occur, the backup file is renamed to the intended file name.

A return value of YES indicates that **writeToFile:atomically:** succeeded. If NO is returned, the method failed.

▶ NSMutableData

Inherits From:	NSData : NSObject
Conforms To:	NSCopying
	NSMutableCopying
Declared In:	foundation/NSData.h

Class Description

The NSMutableData class declares the programmatic interface to an object that contains modifiable data in the form of bytes. NSMutableData's two primitive methods— **mutableBytes** and **setLength:**—provide the basis for all the other methods in its interface. The **mutableBytes** method returns a pointer for writing into the bytes contained in the mutable data object. **setLength:** allows you to truncate or extend the length of a mutable data object.

NSMutableData provides an additional method for changing the length of a mutable data object: **increaseLengthBy:**.

The **appendBytes:length:** and **appendData:** methods let you append bytes or the contents of another data object to a mutable data object. You can replace a range of bytes in a mutable data object with either zeroes (using the **resetBytesInRange:** method), or with different bytes (using the **replaceBytesInRange:withBytes:** method).

Instance Variables

None declared in this class.

Method Types

Allocating and initializing	+ allocWithZone:
	+ dataWithCapacity:
	+ dataWithLength:
	– initWithCapacity:
	– initWithLength:

Adjusting capacity	– increaseLengthBy:
	– setLength:
	– mutableBytes
Adding data	– appendBytes:length:
	– appendData:
Modifying data	– replaceBytesInRange:withBytes:
	– resetBytesInRange:

Class Methods

allocWithZone

+ **allocWithZone:**(NSZone *)*zone*

Creates and returns an uninitialized data object in the specified zone. If the receiver is the NSData class object, an instance of the appropriate immutable subclass is returned; otherwise, an object of the receiver's class is returned.

Typically, you create objects using the **data...** class methods, not the **alloc...** and **init...** methods. Note that it's your responsibility to release objects created with the **alloc...** methods.

dataWithCapacity:

+ **dataWithCapacity:**(unsigned)*aNumItems*

Creates and returns an NSMutableData object, initially allocating enough memory to hold *aNumItems* objects. Mutable data objects allocate additional memory as needed, so *aNumItems* simply establishes the object's initial capacity.

This method acts by invoking the **alloc** and **initWithCapacity:** methods.

See also: – **dataWithLength:**, – **initWithCapacity:**, – **initWithLength:**

dataWithLength:

+ **dataWithLength:**(unsigned)*length*

Creates an autoreleased, mutable data object of *length* bytes, zero-filled.

See also: – **dataWithCapacity:**, – **initWithCapacity:**, – **initWithLength:**

Instance Methods

appendBytes:length:

– (void)**appendBytes:**(const void *)*bytes* **length:**(unsigned)*length*

Appends *length* bytes to a mutable data object from the buffer *bytes*. If page alignment is being used and if the data size is more than a few pages, this method performs an efficient virtual copy using **vm_copy()**.

This excerpt copies the bytes in *data2* into *aBuffer*, and then appends *aBuffer* to *data1*.

```
NSMutableData *data1, *data2;
NSString* firstString =  @"ABCD";
NSString* secondString = @"EFGH";
unsigned char *aBuffer;
unsigned len;

data1 = [NSMutableData
    dataWithBytes:[firstString cString]
    length:[firstString length]];
data2 = [NSMutableData
    dataWithBytes:[secondString cString]
    length:[secondString length]];

len = [data2 length];
aBuffer = malloc(len);

[data2 getBytes:aBuffer];
[data1 appendBytes:aBuffer length:len];
```

The final string value of *data1* is `"ABCDEFGH"`.

See also: – **appendData:**

appendData:

– (void)**appendData:**(NSData *)*other*

Appends the contents of a data object other to a receiver data object.

See also: – **appendBytes:**

increaseLengthBy:

   – (void)**increaseLengthBy:**(unsigned)*extraLength*

Increases the length of a mutable data object by *extraLength*.

See also: – **setLength:**

initWithCapacity:

   – **initWithCapacity:**(unsigned)*capacity*

Initializes a newly allocated mutable data object, giving it enough memory to hold *capacity* bytes. Sets the length of the data object to 0.

See also: – **dataWithCapacity:, – initWithLength:**

initWithLength:

   – **initWithLength:**(unsigned)*length*

Initializes a newly allocated mutable data object, giving it enough memory to hold *length* bytes. Fills the object with zeroes up to *length*. This method acts by invoking the **initWithCapacity:** and **setLength:** methods.

See also: – **dataWithCapacity:**, – **dataWithLength:**, – **initWithCapacity:**

mutableBytes

   – (void *)**mutableBytes**

Returns a pointer to the bytes in a mutable data object that enables you to modify the bytes.

In this excerpt, **mutableBytes** is used to return a pointer to the bytes in *data2*. The bytes in *data2* are then overwritten with the contents of *data1*.

```
NSMutableData *data1, *data2;
NSString* myString = @"string for data1";
NSString* yourString = @"string for data2";
unsigned char *firstBuffer, *secondBuffer;

/* initialize data1, data2, firstBuffer, and secondBuffer...
 * ...
 */

[data2 getBytes:secondBuffer];
```

```
fprintf(stderr, "data2 before: \"%s\"\n", (char*)secondBuffer);
firstBuffer = [data2 mutableBytes];
[data1 getBytes:firstBuffer];
fprintf(stderr, "data1: \"%s\"\n", (char*)firstBuffer);
[data2 getBytes:secondBuffer];
fprintf(stderr, "data2 after: \"%s\"\n", (char*)secondBuffer);
```

This excerpt produces the output:

```
data2 before: "String for data2"
data1: "String for data1."
data2 after: "String for data1."
```

replaceBytesInRange:withBytes:

– (void)**replaceBytesInRange:**(NSRange)*range* **withBytes:**(const void *)*bytes*

Specifies a range within the contents of a mutable data object to be replaced by *bytes*. If *range* isn't within the receiver's range of bytes, an NSRangeException error is raised.

In this excerpt, a range of bytes in *data1* is replaced by the bytes in *data2*.

```
NSMutableData *data1, *data2;
NSString* myString = @"Liz and John";
NSString* yourString = @"Larry";
unsigned len;
unsigned char *aBuffer;
NSRange range = {8, [yourString length]};

data1 = [NSMutableData
    dataWithBytes:[myString cString]
    length:[myString length]];

data2 = [NSMutableData
    dataWithBytes:[yourString cString]
    length:[yourString length]];

len = [data2 length];
aBuffer = malloc(len);
[data2 getBytes:aBuffer];
[data1 replaceBytesInRange:range withBytes:aBuffer];
```

The contents of *data1* change from "Liz and John" to "Liz and Larry."

See also: – **resetBytesInRange:**

resetBytesInRange:

– (void)**resetBytesInRange:**(NSRange)*range*

Specifies a range within the contents of a mutable data object to be replaced by zeroes. If *range* isn't within the receiver's range of bytes, an NSRangeException error is raised.

See also: – **replaceBytesInRange:withBytes:**

setLength:

– (void)**setLength:**(unsigned)*length*

Extends or truncates the *length* of a mutable data object. If the mutable data object is extended, the additional bytes are zero-filled.

See also: – **increaseLengthBy:**