A Foundation for WebScript Programmers: Quick Guide to Useful Classes

This chapter gives an overview of the classes you use most commonly in WebScript, the WebObjects scripting language. For a description of WebScript language features and syntax, see the "Using WebScript" chapter.

Foundation Objects

Types

All the variables you create in WebScript are objects. Consequently, there is only one data type: **id**. For more information on the **id** type, see "The id Data Type" in the "Using WebScript" chapter.

Sending Messages

To get an object to invoke one of its methods, you send it a message. For example, the following statement:

```
[colorArray removeAllObjects];
```

tells the object **colorArray** to invoke its **removeAllObjects** method. Message expressions are enclosed in square brackets:

```
[receiver message];
```

The receiver is an object, and the message is the method you want to invoke and any arguments passed to it. The following are examples of messages with arguments:

```
[colorArray addObject:newColor];
[colorArray writeToFile:fileName atomically:YES];
```

In the first statement, **newColor** is an argument to the method **addObject:**. In the second statement, **fileName** and **YES** are both arguments to the method **writeToFile:atomically:**.

For more information on messages, see "Messaging in WebScript" in the "Using WebScript" chapter.

Representing Objects as Strings

You can get a human-readable string representation of any object by sending it a **description** message. This method is particularly useful for debugging. In some cases, the string returned from **description** only contains the name of the receiver's class, but most objects provide more information. For class specific details, see the **description** method descriptions later in this chapter.

Mutable and Immutable Objects

Some objects are immutable; once they are created, they can't be modified. Other objects are mutable. They can be modified at any time. When you create an object, you can often choose to create it as either immutable or mutable. Three kinds of objects discussed in this chapter—strings, arrays, and dictionaries—have both immutable and mutable versions.

For clarity, it's best to use immutable objects wherever possible. Only use a mutable object if you need to modify its contents after you create it.

Determining Equality

You can determine if two objects are equal using the **isEqual:** method. **isEqual:** returns YES if the receiver of the message and the specified object are equal, NO otherwise. Different types of objects determine equality in different ways. For example, array objects define two arrays as equal if they contain the same contents. For more information, see the **isEqual:** method descriptions later in this chapter.

Reading from and Writing to Files

Strings, arrays, and dictionaries—three of the classes discussed in this chapter—provide methods for writing to and reading from files. The method **writeToFile:atomically:** writes a textual description of the receiver's contents to a specified path name, and corresponding class-specific creation methods—**stringWithContentsOfFile:**, **arrayWithContentsOfFile:**, and **dictionaryWithContentsOfFile:**—create an object from the contents of a specified file.

For example, the following code excerpt:

```
id errorLog = [NSString stringWithContentsOfFile:errorPath];
id newErrorLog = [errorLog stringByAppendingFormat:@"%@: %@.\n",
        timeStamp, @"premature end of file."];
[newErrorLog writeToFile:errorPath atomically:YES];
```

reads the contents of an error log stored in a file, appends a new error to the log, and saves the updated log to the same file.

Writing to Files

The method **writeToFile:atomically:** uses the **description** method to obtain a human-readable string representation of the receiver. It then writes the string to the specified file. The resulting file is suitable for use with ***className*WithContentsOfFile:** methods. This method returns YES if the file is written successfully, and NO otherwise.

If the argument for **atomically:** is YES, the string representation is first written to an auxiliary file. Then the auxiliary file is renamed to the specified file name. If flag is NO, the object is written directly to the specified file. The YES option guarantees that the specified file, if it exists at all, won't be corrupted even if the system should crash during writing.

When **writeToFile:atomically:** fails, it returns NO. If this happens, check the permissions on the specified file and its directory. The most common cause of write failures is that the process owner doesn't have the necessary permissions to write to the file or its directory. If the argument for **atomically:** is NO, it's sufficient to grant write permissions only on the file.

Note: The configuration of your HTTP server determines the user who owns autostarted applications.

Reading from Files

The string, array, and dictionary classes provide methods of the form ***className*WithContentsOfFile:**. These methods create a new object and initialize it with the contents of a specified file, which can be specified with a full or relative pathname.

Working with Strings

NSString and NSMutableString objects represent static and dynamic character strings, respectively. They may be searched for substrings, compared against one another, combined into new strings, and so on.

The difference between NSStrings and NSMutableStrings is that you can't change an NSString's contents from its initial character string. While NSMutableStrings provide methods such as **appendString:** and **setString:** to add to or replace the string's contents, there are no such methods available for NSStrings. For clarity, it's best to use NSStrings wherever possible. Only use an NSMutableString if you need to modify its contents after you create it.

You can create NSStrings with WebScript's @ syntax for defining constant objects. For example, the following statement creates an NSString:

```
id msg = @"The option you chose is no longer available, please choose another.";
```

You can also create string objects with creation methods—methods whose names are preceded by a + and that return new objects. The strings created with the @ syntax are always NSStrings, so they can't be modified. If you use a

creation method instead, you can choose to create either an NSString or a NSMutableString. The following code excerpt illustrates the creation of both NSString and NSMutableString objects:

```
// Create an immutable NSString
id message = [NSString stringWithString:@“Hi”];

// Create a mutable NSMutableString
id message = [NSMutableString stringWithString:@“Hi”];
```

The methods provided by NSString and NSMutableString are described in more detail in the next section, “Commonly Used String Methods.”

Commonly Used String Methods

The following sections list the most commonly used NSString and NSMutableString methods.

Creating Strings

The methods in this section are class methods, as denoted by the plus sign (+). You use class methods to send messages to a class—in this case, NSString and NSMutableString. For more information on class methods, see “Messaging in WebScript” in the “Using WebScript” chapter.

+ string

Returns an empty string. Usually used to create NSMutableStrings. NSStrings created with this method are permanently empty.

```
// Most common use
id mutableString = [NSMutableString string];

// May not be what you want
id string = [NSString string];
```

+ stringWithFormat:

Returns a string created by substituting arguments into a specified format string in the manner that **printf()** does in the C programming language. In WebScript, only the "at sign" (@) conversion character is supported, and it expects a corresponding **id** argument.

```
// These are fine
id party = [NSString stringWithFormat:@"Party date: %@", partyDate];
id mailto = [NSString stringWithFormat:@"mailto: %@", [person email]];
id footer = [NSString stringWithFormat:
        @"Interaction %@ in session %@.",
        numberOfInteractions, sessionNumber];

// C users, NO! This won't work. Only %@ is supported.
id string = [NSString stringWithFormat:@"%d of %d %s", x, y, cString];
```

+ stringWithString:

Returns a string containing the same contents as a specified string. This method is usually used to create an NSMutableString from an NSString. For example, the following statement:

```
id mutableString = [NSMutableString stringWithString:@"Change me."];
```

creates an NSMutableString from a constant NSString object.

+ stringWithContentsOfFile:

Returns a string created by reading characters from a specified file. For example, the following statement creates an NSString containing the contents of the file specified in **path**.

```
id fileContents = [NSString stringWithContentsOfFile:path];
```

See also **writeToFile:atomically:**.

Combining And Dividing Strings

– stringByAppendingFormat:

Returns a string made by appending to the receiver a string constructed from a specified format string and following arguments in the manner of **stringWithFormat:**. For example, assume the variable **guestName** contains the string "Rena". Then the following code excerpt:

```
id string = @"Hi";
id message = [string stringByAppendingFormat:@", %@!", guestName];
```

produces the string **message** with contents "Hi, Rena!".

– stringByAppendingString:

Returns a string object made by appending a specified string to the receiver. This code excerpt, for example:

```
id errorTag = @"Error: ";
id errorString = @"premature end of file.";
id errorMessage = [errorTag stringByAppendingString:errorString];
```

produces the string “Error: premature end of file.”.

– componentsSeparatedByString:

Returns an NSArray containing substrings from the receiver that have been divided by a specified separator string. For example, the following statements:

```
id toolString = @"wrenches, hammers, saws";
id toolArray = [toolString componentsSeparatedByString:@", "];
```

produce an NSArray containing the strings “wrenches”, “hammers”, and “saws.”

See also **componentsJoinedByString:** (NSArray and NSMutableArray).

Comparing Strings

– compare:

Returns -1 if the receiver precedes a specified string in lexical ordering, 0 if it is equal, and 1 if it follows. For example, the following statements:

```
if ([@“hello” compare:@“Hello”] == -1) {
    result = [NSString stringWithFormat:
            @"‘%@’ precedes ‘%@’ lexicographically.",
            @“hello”, @“Hello”];
}
```

result in an NSString **result** with the contents “‘hello’ precedes ‘Hello’ lexicographically.”

– caseInsensitiveCompare:

Same as **compare:**, but case distinctions among characters are ignored.

– isEqual:

Returns YES if a specified object is equivalent to the receiver, NO otherwise. An object is equivalent to a string if the object is an NSString or an NSMutableString and **compare:** returns 0. For example, the following statements:

```
if ([string isEqual:newString) {
    result = @“Found a match”;
}
```

assign the contents “Found a match” to **result** if **string** and **newString** are lexicographically equal.

Converting String Contents

– floatValue

Returns the floating-point value of the receiver’s text as a float, skipping white space at the beginning of the string.

– intValue

Returns the integer value of the string’s text, assuming a decimal representation and skipping white space at the beginning of the string.

Modifying Strings

Warning: The following methods are not supported by NSString. They are only available to NSMutableString objects.

– appendFormat:

Appends a constructed string to the receiver. Creates the new string by using **stringWithFormat:** method with the arguments listed. For example, in the following code excerpt, assume the variable **guestName** contains the string “Rena”:

```
id message = [NSMutableString stringWithString:@“Hi”];
[message appendFormat:@“, %@!”, guestName];
```

message has the resulting contents “Hi, Rena!”.

– appendString:

Adds the characters of a specified string to the end of the receiver. For example, the following statements create an NSMutableString and append another string to its initial value:

```
id mutableString = [NSMutableString stringWithFormat:@“Hello ”];
[mutableString appendString:@“world!”];
```

mutableString has the resulting contents “Hello world!”.

– setString:

Replaces the characters of the receiver with those in a specified string. For example, the following statement replaces the contents of an NSMutableString with the empty string:

```
[mutableString setString:@""];
```

Storing Strings

– writeToFile:atomically:

Writes the string to a specified file, returning YES on success and NO on failure. If YES is specified for **atomically:**, this method attempts to write the file safely so that an existing file with the specified path is not overwritten, and the method does not create a new file at the specified path unless the write is successful. The resulting file is suitable for use with **stringWithContentsOfFile:**. For example, the following code excerpt:

```
id errorLog = [NSString stringWithContentsOfFile:errorPath];
id newErrorLog = [errorLog stringByAppendingFormat:@"%@: %@.\n",
       timeStamp, @"premature end of file."];
[newErrorLog writeToFile:errorPath atomically:YES];
```

reads the contents of an error log stored in a file, appends a new error to the log, and saves the updated log to the same file.

Working with Arrays

NSArray and NSMutableArray objects manage immutable and mutable collections of objects, respectively. Each has the following attributes:

- A count of the number of objects in the array
- The objects contained in the array

The difference between NSArray and NSMutableArray is that you can't add to or remove from an NSArray's initial collection of objects. Insertion and deletion methods provided for NSMutableArrays are not available for NSArrays. Although their use is limited to managing static collections of objects, it is best to use NSArrays wherever possible.

You can create NSArrays with WebScript's @ syntax for defining constant objects. For example, the following statements create NSArrays:

```
id availableQuantities = @(1, 6, 12, 48);
id shortWeekDays = @("Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat");
```

You can also create NSArrays with creation methods. If you want to create a static array that contains variables, you have to use a creation method, since you can't use variables in WebScript's @ syntax. The following statement creates an NSArray that contains variables:

```
id dinnerPreferences = [NSArray arrayWithObjects:firstChoice, secondChoice, nil];
```

The variable **dinnerPreferences** is an NSArray, so its initial collection of objects can't be added to or subtracted from. When you need to create an array that can be modified, use a creation method to create an NSMutableArray. For example, the following statement creates an empty NSMutableArray to which you can add objects:

```
id mutableArray = [NSMutableArray array];
```

The methods provided by NSArray and NSMutableArray are described in more detail in the next section, "Commonly Used Array Methods."

Commonly Used Array Methods

The following tables list the most commonly used NSArray and NSMutableArray methods.

Creating Arrays

The methods in this section are class methods, as denoted by the plus sign (+). You use class methods to send messages to a class—in this case, NSArray and NSMutableArray. For more information on class methods, see "Messaging in WebScript" in the "Using WebScript" chapter.

+ array

Returns an empty array. Usually used to create NSMutableArrays. NSArrays created with this method are permanently empty.

```
// Most common use
id mutableArray = [NSMutableArray array];

// May not be what you want
id array = [NSArray array];
```

+ arrayWithObject:

Returns an array containing the single specified object.

+ arrayWithObjects:

Returns an array containing the objects in the argument list. The argument list is a comma-separated list of objects ending with **nil**.

```
id array = [NSMutableArray arrayWithObjects:
       @“Plates”, @“Plasticware”, @“Napkins”, nil];
```

+ arrayWithArray:

Returns an array containing the contents of a specified array. Usually used to create an NSMutableArray from an immutable NSArray. For example, the following statement:

```
id mutableArray = [NSMutableArray arrayWithArray:@(“A”, “B”, “C”)];
```

creates an NSMutableArray from a constant NSArray object.

+ arrayWithContentsOfFile:

Returns an array initialized from the contents of a specified file. The specified file can be a full or relative pathname; the file that it names must contain a string representation of an array, such as that produced by the **writeToFile:atomically:** method. See also **description**.

Querying Arrays

– count

Returns the number of objects in the array.

– isEqual:

Returns YES if the specified object is an array and has contents equivalent to the receiver, NO otherwise. Two arrays have equal contents if they each hold the same number of objects and objects at a given index in each array satisfy the **isEqual:** test.

– objectAtIndex:

Returns the object located at a specified index. Arrays have a zero-based index. The first object in an array is at index 0, the second is at index 1, and so on. It is an error to specify an index that is out of bounds (greater than or equal to the array’s count).

– indexOfObject:

Returns the index of the first object in the array that is equivalent to a specified object. To determine equality, each element of the array is sent an **isEqual:** message.

– indexOfObjectIdenticalTo:

Returns the index of the first occurrence of the a specified object.

Sorting Arrays

– sortedArrayUsingSelector:

Returns an NSArray that lists the receiver's elements in ascending order, as determined by a specified method. This method is used to sort arrays containing strings and/or numbers. For example, the following code excerpt:

```
id guestArray = @("Suzy", "Alice", "John", "Peggy", "David");
id sortedArray = [guestArray sortedArrayUsingSelector:@"compare:"];
```

creates the NSArray **sortedArray** containing the string "Alice" at index 0, "David" at index 1, and so on.

Adding and Removing Objects

Warning: The following methods are not supported by NSArray. They are only available to NSMutableArray objects.

– addObject:

Adds a specified object at the end of the receiver. It is an error to specify **nil** as an argument to this method. You can not add **nil** to an array.

– insertObject:atIndex:

Inserts an object at a specified index. If the specified index is already occupied, the objects at that index and beyond are shifted down one slot to make room. The specified index can't be greater than the receiver's count, and the specified object man not be **nil**.

Array objects have a zero-based index.. The first object in an array is at index 0, the second is at index 1, and so on. You can only insert new objects in ascending order—with no gaps. Once you add two objects, the array's size is 2, so you can insert objects at indexes 0, 1, or 2. Index 3 is illegal and out of bounds.

It is an error to specify **nil** as an argument to this method. You can not add **nil** to an array. It is also an error to specify an index that is greater than the array's count.

– removeObject:

Removes all objects in the array equivalent to a specified object, and moves elements up as necessary to fill any gaps. Equivalency is determined using the **isEqual:** method.

– removeObjectIdenticalTo:

Removes all occurrences of a specified object, and moves elements up as necessary to fill any gaps.

– removeObjectAtIndex:

Removes the object at a specified index and moves all elements beyond the index up one slot to fill the gap. Arrays have a zero-based index. The first object in an array is at index 0, the second is at index 1, and so on.

It is an error to specify an index that is out of bounds (greater than or equal to the array's count).

– removeAllObjects

Empties the receiver of all of its elements.

– setArray:

Empties the receiver of all its elements, then adds the contents of a specified array.

Storing Arrays

– writeToFile:atomically:

Writes the array's string representation to a specified file using the **description** method. Returns YES on success and NO on failure. If YES is specified for **atomically:**, this method attempts to write the file safely so that an existing file with the specified path is not overwritten, and it does not create a new file at the specified path unless the write is successful. The resulting file is suitable for use with **arrayWithContentsOfFile:**. For example, the following code excerpt:

```
id guestArray = [NSMutableArray arrayWithContentsOfFile:path];
[guestArray addObject:newGuest];
[guestArray writeToFile:path atomically:YES];
```

creates **guestArray** with the contents of the specified file, adds a new guest, and saves the changes to the same file.

Representing Arrays as Strings

– description

Returns a string that represents the contents of the receiver. For example, the following code excerpt:

```
id array = [NSMutableArray arrayWithObjects:
       @“Plates”, @“Plasticware”, @“Napkins”, nil];
id description = [array description];
```

produces the string “(Plates, Plasticware, Napkins)”.

– componentsJoinedByString:

Returns an NSString created by interposing a specified string between the elements of the receiver’s objects. Each element of the array must be a string. If the receiver has no elements, an empty string is returned. See also **componentsSeparatedByString:** (NSString and NSMutableString). For example, the following code excerpt:

```
id commaString = @“A, B, C”;
id array = [string componentsSeparatedByString:@“,”];
id dash String = [array componentsJoinedByString:@“-”];
```

creates the NSString **dashString** with the contents “A-B-C”.

Working with Dictionaries

NSDictionary and NSMutableDictionary objects store collections of key-value pairs. The key-value pairs within a dictionary are called *entries*. Each entry consists of an NSString that represents the key and a second object which is that key’s value. Within a dictionary, the keys are unique. That is, no two keys in a single dictionary are equivalent. You use dictionaries to store objects that can be uniquely identified by strings.

The difference between NSDictionary and NSMutableDictionary is that you can’t add, modify, or remove entries from an NSDictionary’s initial collection of entries. Insertion and deletion methods provided for NSMutableDictionaries are not available for NSDictionaries. Although their use is limited to managing static collections of objects, it’s best to use NSDictionaries wherever possible.

You can create NSDictionaries with WebScript's @ syntax for defining constant objects. For example, the following statements create NSDictionaries:

```
id sizes = @{“S” = “Small”; “M” = “Medium”; “L” = “Large”; “X” = “Extra Large”};
id defaultPreferences = @{
    “seatAssignment” = “Window”;
    “smoking” = “Non-smoking”;
    “aircraft” = “747”};
```

You can also create dictionaries with creation methods. For example, if you want to create an NSDictionary that contains variables, you have to use a creation method. You can't use variables with WebScript's @ syntax. The following statement creates an NSDictionary that contains variables:

```
id customerPreferences = [NSDictionary dictionaryWithObjectsAndKeys:
    seatingPreference, @“seatAssignment”,
    smokingPreference, @“smoking”,
    aircraftPreference, @“aircraft”, nil];
```

The variable **customerPreferences** is an NSDictionary, so its initial collection of entries can't be modified. To create a dictionary that can be modified, use a creation method to create an NSMutableDictionary. For example, the following statement creates an empty NSMutableDictionary:

```
id dictionary = [NSMutableDictionary dictionary];
```

The methods provided by NSDictionary and NSMutableDictionary are described in more detail in the next section, “Commonly Used Dictionary Methods.”

Commonly Used Dictionary Methods

The following sections list some of the most commonly used methods of NSDictionary and NSMutableDictionary.

Creating Dictionaries

The methods in this section are class methods, as denoted by the plus sign (+). You use class methods to send messages to a class—in this case, NSDictionary

and NSMutableDictionary. For more information on class methods, see "Messaging in WebScript" in the "Using WebScript" chapter.

+ dictionary

Returns an empty dictionary. Usually used to create NSMutableDictionaries. NSDictionaries created with this method are permanently empty.

```
// Most common use
id mutableDictionary = [NSMutableDictionary dictionary];

// May not be what you want
id dictionary = [NSDictionary dictionary];
```

+ dictionaryWithObjects:forKeys:

Returns a dictionary containing entries constructed from the contents of a specified array of objects and a specified array of keys. The two arrays must have the same number of elements.

```
id preferences = [NSMutableDictionary
        dictionaryWithObjects:@("window", "non-smoking", "747")
        forKeys:@("seatAssignment", "smoking", "aircraft")];
```

+ dictionaryWithObjectsAndKeys:

Returns a dictionary containing entries constructed from a specified set of objects and keys. **dictionaryWithObjectsAndKeys:** takes a variable number of arguments: a list of alternating objects and keys ending with **nil**.

```
id customerPreferences = [NSDictionary dictionaryWithObjectsAndKeys:
        seatingPreference, @"seatAssignment",
        smokingPreference, @"smoking",
        aircraftPreference, @"aircraft", nil];
```

+ dictionaryWithDictionary:

Returns a dictionary containing the contents of a specified dictionary. Usually used to create an NSMutableDictionary from an immutable NSDictionary.

+ dictionaryWithContentsOfFile:

Returns a dictionary initialized from the contents of a specified file. The specified file can be a full or relative pathname; the file that it names must contain a string representation of a dictionary, such as that produced by the **writeToFile:atomically:** method.

See also **description**.

Querying Dictionaries

– allKeys

Returns an array containing the dictionary's keys or an empty array if the dictionary has no entries. This method is useful for accessing all the entries in a dictionary. For example, the following code excerpt:

```
id index;
id keys = [dictionary allKeys];
for (index = 0; index < [keys count]; index++) {
    value = [dictionary objectForKey:[keys objectAtIndex:index];
    // Use the value
}
```

creates the NSArray **keys** and uses it to access the value of each entry in the dictionary.

– allKeysForObject:

Returns an array containing all the keys corresponding to values equivalent to a specified object. Equivalency is determined using the **isEqual:** method. If the specified object isn't equivalent to any of the values in the receiver, this method returns **nil**.

– allValues:

Returns an array containing the dictionary's values, or an empty array if the dictionary has no entries.

Note that the array returned from **allValues** may have a different count than the array returned from **allKeys**. An object can be in a dictionary more than once if it corresponds to multiple keys.

– keysSortedByValueUsingSelector:

Returns an NSArray containing the dictionary's keys such that their corresponding values are sorted in ascending order, as determined by a specified method. For example, the following code excerpt:

```
id choices = @{“Steak” = 3; “Seafood” = 2; “Pasta” = 1};
id keys = [choices sortedByValueUsingSelector:@“compare:”];
```

creates the NSArray **keys** containing the string “Pasta” at index 0, “Seafood” at index 1, and “Steak” at index 2.

– count

Returns the number of entries currently in the dictionary.

– isEqual:

Returns YES if the specified object is a dictionary and has contents equivalent to the receiver, NO otherwise. Two dictionaries have equivalent contents if they each hold the same number of entries and, for a given key, the corresponding value objects in each dictionary satisfy the **isEqual:** test.

– objectForKey:

Returns the object that corresponds to a specified key. For example, the following code excerpt:

```
id preferences = [NSMutableDictionary
       dictionaryWithObjects:@(“window”, “non-smoking”, “747”)
       forKeys:@(“seatAssignment”, “section”, “aircraft”)];

id sectionPreference = [dictionary objectForKey:@“section”];
```

produces the NSString **sectionPreference** with the contents “non-smoking”.

Adding, Removing, and Modifying Entries

Warning: The following methods are not supported by NSDictionary. They are only available to NSMutableDictionary objects.

– setObject:forKey:

Adds an entry to the receiver, consisting of a specified key and its corresponding value object. If the receiver already has an entry for the specified key, the previous value for that key is replaced with the argument for **setObject:**. For example, the following code excerpt:

```
id dictionary = [NSMutableDictionary dictionaryWithDictioary:
       @{“seatAssignment” = “window”}];
[dictionary setObject:@“non-smoking” forKey:@“section”];
[dictionary setObject:@“aisle” forKey:@“seatAssignment”];
```

produces the NSMutableDictionary **dictionary** with the value “non-smoking” for the key “section” and the value “aisle” for the key “seatAssignment”. Notice that the original value for “seatAssignment” is replaced.

It is an error to specify **nil** as an argument for **setObject:** or **forKey:**. You can’t put **nil** in a dictionary as a key or as a value.

– addEntriesFromDictionary:

Adds the entries from a specified dictionary to the receiver. If both dictionaries contain the same key, the receiver’s previous value for that key is replaced with the new value.

– removeAllObjects

Empties the dictionary of its entries.

– removeObjectForKey:

Removes the entry for a specified key.

– removeObjectsForKeys:

Removes the entries for each key in a specified array.

– setDictionary:

Removes all the entries in the receiver, then adds the entries from a specified dictionary.

Representing Dictionaries as Strings

– description

Returns a string that represents the contents of the receiver. For example, the following code excerpt:

```
id preferences = [NSMutableDictionary
       dictionaryWithObjects:@(“window”, “non-smoking”, “747”)
       forKeys:@(“seatAssignment”, “section”, “aircraft”)];

id description = [preferences description];
```

produces the string “{“seatAssignment” = “Window”; “section” = “Non-smoking”; “aircraft” = “747”}”.

Storing Dictionaries

– writeToFile:atomically:

Writes the dictionary’s string representation to a specified file using the **description** method. Returns YES on success and NO on failure. If YES is specified for **atomically:**, this method attempts to write the file safely so that an existing file with the specified path is not overwritten, and it does not create a new file at the specified path unless the write is successful. The resulting file is suitable for use with **dictionaryWithContentsOfFile:**. For example, the following excerpt:

```
id defaults = [NSMutableDictionary
       dictionaryWithContentsOfFile:path];
[defaults setObject:newLanguagePreference forKey:@“Language”];
[defaults writeToFile:path atomically:YES];
```

creates an NSMutableDictionary from the contents of the file specified by **path**, updates the object for the key @“Language”, and saves the updated dictionary back to the same file.

See also **description**.

Working with NSCalendarDates

NSCalendarDate objects represent dates and times. These objects are especially suited for representing and manipulating dates according to western calendrical systems. NSCalendarDate performs date computations based on western calendrical systems, primarily the Gregorian.

The methods provided by NSCalendarDate are described in more detail in the section "Commonly Used NSCalendarDate Methods."

The Calendar Format

Each NSCalendarDate object has a calendar format associated with it. This format is a string that contains date-conversion specifiers that are very similar to those used in the standard C library function **strftime()**. NSCalendarDate interprets dates that are represented as strings conforming to this format. You can set the default format for an NSCalendarDate object at initialization time or using the **setCalendarFormat:** method. Several methods allow you to specify formats other than the one bound to the object.

The date conversion specifiers cover a range of date conventions:

Conversion Specifier	Argument Type
%%	a '%' character
%A, %a	full and abbreviated weekday name, respectively
%B, %b	full and abbreviated month name, respectively
%d	day of the month as a decimal number (01-31)
%F	milliseconds as a decimal number (000-999)
%H, %I	hour based on a 24-hour or 12-hour clock as a decimal number, respectively. (00-23 or 01-12)
%j	day of the year as a decimal number (001-366)
%M	minute as a decimal number (00-59)
%p	AM/PM designation for the locale
%S	second as a decimal number (00-59)
%w	weekday as a decimal number (0-6), where Sunday is 0
%Y, %y	year with century (such as 1990) and year without century (00-99), respectively.
%Z, %z	time zone abbreviation (such as PDT) and time zone offset in hours and minutes from GMT (HHMM), respectively.

Commonly Used NSCalendarDate Methods

The following sections list some of the most commonly used methods of NSCalendarDate.

Creating NSCalendarDates

The methods in this section are class methods, as denoted by the plus sign (+). You use class methods to send messages to a class—in this case, NSCalendarDate. For more information on class methods, see Messaging in WebScript in "Using WebScript."

+ calendarDate

Returns an NSCalendarDate initialized to the current date and time.

+ dateWithString:calendarFormat:

Returns an NSCalendarDate initialized to the date in a provided string, and sets the new NSCalendarDate's calendar format to the specified format. The date string must match the provided format exactly. See "**The Calendar Format**" for more detailed information on formats used by NSCalendarDate.

Adjusting an NSCalendarDate

– addYear:month:day:hour:minute:second:

Returns an NSCalendarDate derived from the receiver by adding a specified number of years, months, days, hours, minutes, and seconds.

Representing NSCalendarDates as Strings

– description

Returns a string representation of the NSCalendarDate formatted according to the NSCalendarDate's default calendar format.

– descriptionWithCalendarFormat:

Returns a string representation of the receiver formatted according to the provided format string.

– calendarFormat

Returns a string that indicates the receiver's default calendar format. See "**The Calendar Format**" for more detailed information on formats used by NSCalendarDate.

– setCalendarFormat:

Set the receiver's default calendar format to the provided string.

Retrieving NSCalendarDate Elements

– dayOfWeek

Returns a number that indicates the NSCalendarDate's day of the week (0-6).

– dayOfMonth

Returns the NSCalendarDate's day of the month (1-31).

– dayOfYear

Returns a number that indicates the NSCalendarDate's day of the year (1-366).

– dayOfCommonEra

Returns the NSCalendarDate's number of days since the beginning of the Common Era. The base year of the Common Era is 1 A.C.E. (which is the same as 1 A.D.).

– monthOfYear

Returns a number that indicates the NSCalendarDate's month of the year (1-12).

– yearOfCommonEra

Returns the NSCalendarDate's year value (including the century).

– hourOfDay

Returns the NSCalendarDate's hour value (0-23).

– minuteOfHour

Returns the NSCalendarDate's minutes value (0-59).

– secondOfMinute

Returns the NSCalendarDate's seconds value (0-59).