Visual Studio 2003 JScript

The content in this document is retired and is no longer updated or supported. Some links might not work. Retired content represents the latest updated version of this content.

Visual Studio .NET

Microsoft Visual Studio.net	Microsoft® Visual Studio® .NET is the comprehensive tool set for rapidly building and integrating XML Web services, Microsoft Windows®–based applications, and Web solutions. Today, the world takes a little step into the future.

Check out the Visual Studio .NET Developer Center

You'll find in-depth content, community resources, downloads, product information, and more at this one-stop resource for Visual Studio .NET developers.

In This Library Section	Essentials
• **Documentation** • **Technical Articles** • **Introducing Visual Studio .NET** • **Developing with Visual Studio .NET** • **Visual Studio Walkthroughs** • **Code Samples**	• **Frequently Asked Questions About Visual Studio .NET** • **Inside the .NET Framework** • **Upgrading to .NET** • **Visual Studio .NET Support Center** • **Newsgroups** • **Communities**

JScript

JScript .NET is a modern scripting language with a wide variety of applications. It is a true object-oriented language, and yet it still keeps its "scripting" feel. JScript .NET maintains full backwards compatibility with previous versions of JScript, while incorporating great new features and providing access to the common language runtime and .NET Framework.

The following topics introduce the essential components of JScript .NET and provide information about how to use the language. As with any modern programming language, JScript supports a number of common programming constructs and language elements.

If you have programmed in other languages, much of the material covered in this section may seem familiar. While most of the constructs are the same as in previous versions of JScript, JScript .NET introduces powerful new constructs similar to those in other class-based, object-oriented languages.

If you are new to programming, the material in this section will serve as an introduction to the basic building blocks for writing code. Once you understand the basics, you will be able to create powerful scripts and applications using JScript.

In This Section

Getting Started With JScript .NET
Introduces what is new in JScript .NET.

Writing, Compiling, and Debugging JScript Code
Provides a collection of links to topics that explain how to write, edit, and debug code with JScript .NET.

Displaying Information with JScript .NET
Includes a list of links to topics that explain how to display information from a command program, from ASP.NET, and in a browser.

Introduction to Regular Expressions
Comprises a guide to the elements and procedures that encompass Regular Expressions in JScript .NET. Topics explain the concept of Regular Expressions, proper syntax, and appropriate use.

JScript Reference
Lists elements that comprise JScript Language Reference and links to topics that explain the details behind the proper use of language elements.

Related Sections

Devenv Command Line Switches
Lists language reference topics that explain how to launch Visual Studio and how to build from the command prompt.

Development Environment Reference
Provides a list of links to topics such as shortcut keys, regular expression syntax, wildcard syntax, and so on.

Language Equivalents
Compares keywords, data types, operators, and programmable objects (controls) for Visual Basic, C++, C#, JScript, and Visual FoxPro.

.NET Framework Class Library
Contains links to topics that explain the namespaces in the .NET Framework class library and explains how to use the class library documentation.

User Interface Reference
Contains topics that explain options that appear on various dialog boxes, windows, and other user interfaces in Visual Studio.

Visual Studio Commands
Lists language reference topics that explain how to use commands to interact with the IDE from the Command Window and Find/Command box.

Visual Studio Walkthroughs
Provides links to topics that discuss the steps involved in the development of specific applications or how to use major application features.

Getting Started With JScript .NET

JScript .NET represents a major advance of the JScript language in several ways. With tighter integration into the Visual Studio .NET development environment, multiple new features, and access to the .NET Framework classes, moving from JScript to JScript .NET may appear at first to be a daunting task.

In reality, almost all of the changes represent additional functionality, while the core JScript .NET functionality is the same as in previous versions. Practically all JScript scripts will run without modification under JScript .NET (with the fast mode turned off). To run in fast mode (the mode supported by ASP.NET), some scripts will need minor modifications.

It is easy to make the transition from JScript to JScript .NET since you can include the new features into your code gradually. You can upgrade your scripts at your own pace, adding new functionality as you learn more about JScript .NET.

The following documentation will help you to upgrade your applications and quickly understand the changes to JScript .NET.

In This Section

What Is JScript .NET?
Provides a general overview of JScript .NET, its relationship to ECMAScript, and its evolution from previous versions of JScript.

What's New in JScript .NET
Lists the new features in JScript .NET, which include features developed in conjunction with ECMAScript Edition 4 and additional features not specified in ECMAScript.

Introduction to JScript .NET for JScript Programmers
Describes differences between JScript and JScript .NET which provides a quick background for experienced JScript programmers.

The JScript Version of Hello World!
Illustrates how to write the familiar Hello World! program in JScript .NET.

Upgrading Applications Created in Previous Versions of JScript
Describes how to upgrade existing JScript applications to work with JScript .NET.

Running a JScript Application on a Previous Version of the Common Language Runtime
Explains how to configure a JScript .NET application built with one version of the runtime to run on a previous version of the runtime.

Additional Resources for JScript Programmers
Lists several sites and newsgroups that provide answers for common JScript programming issues.

Related Sections

Version Information
Lists all the features of JScript and the corresponding version in which each was introduced.

JScript Language Tour
Introduces the elements and procedures that developers use to write JScript code and links to specific areas that explain the details behind language elements and code syntax.

JScript Reference
Lists elements that comprise JScript Language Reference and links to topics that explain the details behind the proper use of language elements.

What Is JScript .NET?

JScript .NET is the next generation of an implementation by Microsoft of the ECMA 262 language. Combining the feature set of previous versions of JScript with the best features of class-based languages, JScript .NET includes the best of both worlds. Improvements in JScript .NET — which is being developed in conjunction with ECMAScript Edition 4 — include true compiled code, typed and typeless variables, late- and early-binding, classes (with inheritance, function overloading, property accessors, and more), packages, cross-language support, and full access to the .NET Framework.

JScript .NET is a true object-oriented scripting language. Although JScript .NET can now use classes, types, and other advanced language features for writing robust applications, it retains its "scripting" feel, with support for typeless programming, expando functions and classes, dynamic code execution (using **eval**), and more.

In addition to being a typeless language, JScript .NET can now be a strongly typed language. In previous versions, the loosely typed structure of JScript meant that variables assumed the type of the value assigned to them. In fact, you could not declare the data types of variables in previous versions. JScript .NET provides more flexibility than previous versions of JScript by allowing variables to be type annotated. This binds a variable to a particular data type, and the variable can store only data of that type.

There are many advantages of strong typing in a programming language. In addition to the benefit that occurs when you use a data type that properly fits the data you are using, you get several other benefits:

- Improved execution speed
- Run-time/compile-time type-checking
- Self-documenting code

Finally, it is important to remember that JScript .NET is not a condensed version of another programming language, nor is it a simplification of anything. It is a modern scripting language with a wide variety of applications.

> **Note** The code in many of the included JScript .NET examples is somewhat more explicit and less dense than in actual scripts. The intent is to clarify the concepts rather than to express optimal coding conciseness and style. In any case, there is no shame in writing code that you can read and easily understand six months after you write it.

See Also

JScript Reference

What's New in JScript .NET

JScript .NET, the next generation of the Microsoft JScript language, is designed to be a fast and easy way to access the Microsoft .NET platform using the language of the Web. The primary role of JScript .NET is construction of Web sites with ASP.NET and customization of applications with Script for the .NET Framework.

JScript .NET, which is compatible with the ECMAScript standard, has additional features not specified by ECMAScript, such as true compiled code, cross-language support through Common Language Specification (CLS) compliance, and access to the .NET Framework. While the version of JScript .NET included in Visual Studio .NET 2002 leveraged the inherent security of the .NET Framework, JScript .NET 2003 further enhances security by adding a restricted security context for the **eval** Method.

Several new features in JScript .NET take advantage of the CLS — a set of rules that standardizes such things as data types, how objects are exposed, and how objects interoperate. Any CLS-compliant language can use the classes, objects, and components that you create in JScript .NET. And you, as a JScript developer, can access classes, components, and objects from other CLS-compliant programming languages without considering language-specific differences such as data types. Some of the CLS features that JScript .NET programs use are namespaces, attributes, by-reference parameters, and native arrays.

Following are some of the new features in JScript .NET:

What's New in JScript .NET 2003

Restricted Security Context for eval Method

To enhance security, the built-in **eval** method now by default runs scripts in a restricted security context, regardless of the caller's permissions. Calling **eval** with "unsafe" as the optional second parameter causes the script to run with the permissions of the caller, which may allow access to the file system, the network, or the user interface. For more information, see eval Method.

What's New in JScript .NET 2002

Class-based Objects

JScript .NET (like JScript) supports inheritance through prototype-based objects. JScript .NET also supports class-based objects by allowing you to declare classes that define data and behavior for objects. Classes created in JScript .NET can be used and extended by any .NET language. Classes can inherit the properties and methods of a base class. There are several attributes you can apply to classes and class members that modify their behavior and visibility. For more information, see Class-based Objects.

JScript Data Types

In JScript .NET (like JScript), you can write programs without specifying data types for your variables. JScript .NET can also be used as a strongly typed language, in which all variables are bound to a specific data type, or you can use a mix of typed and untyped variables. JScript .NET provides many new data types. You can also use classes and .NET types as data types. For more information, see JScript Data Types.

Conditional Compilation

Directives control compilation of your JScript .NET program. The @debug directive, for instance, turns the emission of debugging information on or off for a particular part of your script. For more information, see @debug Directive. The @position directive sets the line number for the debugger for the current line. For more information, see @position Directive. Both of these directives are useful if you are writing code that will be incorporated in other scripts. For more information, see Conditional Compilation.

JScript Namespaces

Namespaces prevent naming conflicts by organizing classes, interfaces, and methods into hierarchies. In JScript .NET, you can define your own namespaces. You can also access any .NET Framework namespace with JScript .NET, including ones you define. The package statement enables packaging of related classes for easy deployment and to avoid naming collisions. For more information, see package Statement. The import statement makes a .NET Framework namespace available to a script so that the script can access the classes and interfaces in the namespace. For more information, see import Statement.

JScript Variables and Constants

JScript .NET introduces a const statement that defines an identifier that represents a constant value. For more information, see JScript Variables and Constants.

Enumerations

JScript .NET introduces the enum statement that allows you to construct enumerated data types. With an enumeration, you can specify helpful names for your data type values. For more information, see enum Statement.

See Also

What Is JScript .NET? | What's New in the Visual Basic Language | What's New in Visual Studio .NET | Modifiers | Data Types | Directives | Statements | JScript Reference

Introduction to JScript .NET for JScript Programmers

The information presented here is mainly for programmers who are already familiar with JScript and want to learn about the new features introduced in JScript .NET.

How to compile programs

The JScript .NET command-line compiler creates executables and assemblies from JScript programs. For more information, see Compiling JScript code from the Command Line.

How to write a "Hello World" program

It is easy to write the JScript .NET version of "Hello World". For more information, see The JScript Version of Hello World!.

How to use data types

In JScript .NET, a colon specifies the type in a variable declaration or function definition. The default type is **Object**, which can hold any of the other types. For more information, see JScript Variables and Constants and JScript Functions.

JScript .NET has several built-in data types (such as **int**, **long**, **double**, **String**, **Object**, and **Number**). For more information, see JScript Data Types. You can also use any .NET Framework data type after importing the appropriate namespace. For more information, see .NET Framework Class Library.

How to access a namespace

A namespace is accessed using either the **import** statement (when using the command-line compiler) or the **@import** directive (when using ASP.NET). For more information, see import Statement. The **/autoref** option (which is on by default) automatically attempts to reference the assemblies that correspond to namespaces used in a JScript .NET program. For more information, see /autoref.

How to create typed (native) arrays

A typed array data type is declared by placing square brackets (**[]**) after the data type name. You can still use JScript array objects, objects created with the **Array** constructor. For more information, see Arrays Overview.

How to create a class

In JScript .NET, you can define your own classes. Classes can include methods, fields, properties, static initializers, and sub-classes. You can write a completely new class, or you can inherit from an existing class or interface. Modifiers control the visibility of the class members, how members are inherited, and the overall behavior of a class. Custom attributes can also be used. For more information, see Class-based Objects and JScript Modifiers.

See Also

Getting Started With JScript .NET | Upgrading Applications Created in Previous Versions of JScript

The JScript Version of Hello World!

The following console program is the JScript version of the traditional "Hello World!" program, which displays the string `Hello World!`.

```
// A "Hello World!" program in JScript.
print("Hello World!");
```

The important points of this program are the following:

- Comments
- Output
- Compilation and execution

Comments

The first line of the example contains a comment. Since the compiler ignores the comment, you can write any text. This comment describes the purpose of the program.

```
// A "Hello World!" program in JScript.
```

The double forward slash (`//`) means that the rest of the line is a comment. You can make an entire line a comment, or you can append a comment to the end of another statement, as follows:

```
var area = Math.PI*r*r;  // Area of a circle with radius r.
```

You can also use multiline comments. A multiline JScript comment begins with a forward slash and asterisk (`/*`), and ends with the reverse (`*/`).

```
/*
Multiline comments allow you to write long comments.
They can also be used to "comment out" blocks of code.
*/
```

For more information, see JScript Comments.

Output

This example uses the **print** statement to display the string `Hello World!`:

```
print("Hello World!");
```

For more information, see print Statement.

There are other ways for a program to communicate with the user. The class **System.Console** exposes methods and properties that facilitate interaction with the person using the console. For more information, see Console Class. The class **System.Windows.Forms.MessageBox** exposes methods and properties that facilitate interaction with the user using Windows Forms. For more information, see System.Windows.Forms Namespace.

Compilation and Execution

You can compile the "Hello World!" program using the command line compiler.

To compile and run the program from the command line

1. Create the source file using any text editor and save it using a file name such as Hello.js.
2. To start the compiler, enter the following at the command prompt:

   ```
   jsc Hello.js
   ```

Note You should compile the program from the Visual Studio .NET Command Prompt. For more information, see Compiling JScript Code from the Command Line.

If your program does not contain compilation errors, the compiler creates a Hello.exe file.

3. To run the program, enter the following at the command prompt:

```
Hello
```

For more information, see Compiling JScript code from the Command Line.

See Also

Writing, Compiling, and Debugging JScript Code | Displaying Information with JScript .NET | JScript Reference

Upgrading Applications Created in Previous Versions of JScript

A majority of existing JScript code will work fine with the enhancements included in JScript .NET because JScript .NET is almost completely backwards compatible with previous versions. The new features of JScript .NET cover new areas.

By default, JScript .NET programs are compiled in *fast mode*. Since fast mode places some restrictions on the types of code that are allowed, programs can be more efficient and execute faster. However, some features available in previous versions are not available in fast mode. For the most part, these features were incompatible with multi-threaded applications and produced inefficient code. For programs compiled with the command-line compiler, you can turn fast mode off and have complete backwards compatibility. Note that code compiled in this way is slower and more susceptible to errors. Fast mode cannot be turned off in ASP.NET applications because of the stability issues it would present. For more information, see /fast.

In fast mode, the following JScript behaviors are triggered:

- All variables must be declared.
- Functions become constants.
- Intrinsic objects cannot have expando properties.
- Intrinsic objects cannot have properties listed or changed.
- The arguments object is not available.
- Cannot assign to a read-only variable, field, or method.
- eval method cannot define identifiers in the enclosing scope.
- eval method executes scripts in a restricted security context.

All variables must be declared

Previous versions of JScript did not require explicit declaration of variables. Although this feature saves keystrokes for programmers, it also makes it difficult to trace errors. For example, you could assign a value to a misspelled variable name, which would neither generate an error nor return the desired result. Furthermore, undeclared variables have global scope, which can cause additional confusion.

Fast mode requires explicit variable declarations. This helps prevent the types of errors mentioned above and produces code that runs faster.

JScript .NET also supports type-annotated variables. This binds each variable to a particular data type, and the variable can store only data of that type. Although type annotation is not required, using it helps prevent errors associated with accidentally storing the wrong data in a variable and can increase program execution speed.

For more information, see JScript Variables and Constants.

Functions become constants

In previous versions of JScript, functions declared with the **function** statement were treated the same as variables that held a **Function** object. In particular, any function identifier could be used as a variable to store any type of data.

In fast mode, functions become constants. Consequently, functions cannot have new values assigned to them or be redefined. This prevents accidentally changing the meaning of a function.

If your script requires that a function change, you can explicitly use a variable to hold an instance of the **Function** object. Note, however, that **Function** objects are slow. For more information, see Function Object.

Intrinsic objects cannot have expando properties

In previous versions of JScript, you could add expando properties to intrinsic objects. This behavior could be used to add a method to a **String** object to trim leading spaces for the string, for example.

In fast mode, this is not allowed. If your script relies on this feature, you must modify the script. You can define functions in the global scope instead of attaching those functions to objects as methods. Then, rewrite each instance in the script where an expando method is called from the object so that the object is passed to the appropriate function.

One notable exception to this rule is the **Global** object, which still can have expando properties. All identifiers in the global scope are actually properties of the **Global** object. Obviously, the **Global** object must be dynamically extensible to support the addition of new global variables.

Intrinsic objects cannot have properties listed or changed

In previous versions of JScript, you could delete, enumerate, or write to the predefined properties of intrinsic objects. This behavior could be used to change the default **toString** method of the **Date** object, for example.

In fast mode, this is not allowed. This feature is no longer necessary since intrinsic objects cannot have expando properties, and the properties for each object are listed in the reference section. For more information, see Objects.

The arguments object is not available

Previous versions of JScript provided an **arguments** object inside function definitions, which allowed functions to accept an arbitrary number of arguments. The arguments object also provided a reference to the current function as well as the calling function.

In fast mode, the **arguments** object is not available. However, JScript .NET allows function declarations to specify a *parameter array* in the function parameter list. This allows the function to accept an arbitrary number of arguments, thus replacing part of the functionality of the **arguments** object. For more information, see function Statement.

There is no way to directly access and reference the current function or calling function in fast mode.

Cannot assign to a read-only variable, field, or method

In previous versions of JScript, statements could appear to assign a value to a read-only identifier. The assignment would fail quietly, and the only way to discover the assignment failure would be to test if the value actually changed. Assignment to a read-only identifier usually is the result of a mistake, since it has no effect.

In fast mode, a compile-time error will be generated if you attempt to assign a value to a read-only identifier. You can either remove the assignment or try assigning to an identifier that is not read-only.

If you turn fast mode off, assignments to read-only identifiers will fail silently at run-time, but a compile-time warning will be generated.

eval method cannot define identifiers in the enclosing scope

In previous versions of JScript, functions and variables could be defined in the local or global scope by a call to the **eval** method.

In fast mode, functions and variables can be defined within a call to the **eval** method, but they are accessible from within that particular call. Once the **eval** is finished, the functions and variables defined within the **eval** are no longer accessible. The result of a calculation made within an **eval** can be assigned to any variable accessible in the current scope. Calls to the **eval** method are slow, and you should consider rewriting code that contains them.

The previous behavior of the **eval** method is restored when fast mode is turned off.

eval method executes scripts in a restricted security context

In previous versions of JScript, code passed to the **eval** method would run in the same security context as the calling code.

To protect users, code passed to the **eval** method executes in a restricted security context, unless the string "unsafe" is passed as the second parameter. The restricted security context forbids access to system resources, such as the file system, the network, or the user interface. A security exception is generated if the code attempts to access those resources.

When the second parameter of eval is the string "unsafe", the code passed to the **eval** method is executed in the same security context as the calling code. This restores previous behavior of the **eval** method.

> **Security Note** Use **eval** in unsafe mode only to execute code strings obtained from trustworthy sources.

See Also

Getting Started With JScript .NET | Introduction to JScript .NET for JScript Programmers | /fast

Running a JScript Application on a Previous Version of the Common Language Runtime

Unless otherwise specified, a JScript .NET application is built to run with the common language runtime version that the compiler uses to build the application. However, it is possible for an .exe or ASP.NET Web application built with one version of the runtime to run on any version of the runtime.

To accomplish this, an .exe application needs an app.config file containing runtime version information (with the supportedRuntime tag). Other Visual Studio languages offer Integrated Development Environment (IDE) support for modifying the app.config file via their project's property pages dialog box. For example, modify the **SupportedRuntimes** property of a Visual C# windows application and use that updated app.config file in your JScript application. For more information, see Common Properties Property Pages Dialog Box.

At runtime, the name of the app.config file must be *filename.ext*.config (where *filename.ext* is the name of the executable that started the application) and the file must be in the same directory as the executable. For example, if your application is called TestApp.exe, the app.config file would be named TestApp.exe.config.

If you specify more than one runtime version and the application runs on a computer with more than one installed runtime version, the application uses the first version specified in the config file that matches an installed runtime that is available on the system.

For more information, see Targeting a .NET Framework Version.

Since JScript ASP.NET Web pages are single-file Web Form pages, they are not precompiled to a .dll with a dependency on the .NET Framework assembly associated with the compiler. Consequently, pages are compiled at runtime and no runtime version information is required in a web.config file.

See Also

SupportedRuntimes Property | Web Forms Code Model

Additional Resources for JScript Programmers

The following sites and newsgroups provide answers for common JScript programming issues.

Microsoft Resources

On the Web

Microsoft Product Support Services (http://support.microsoft.com/)
Provides access to Knowledge Base articles, downloads and updates, support Webcasts, and other services.

About JSscript .NET on GotDotNet (http://www.gotdotnet.com/team/jscript/)
Contains articles, samples, and other information of interest to JScript .NET developers.

MSDN Newsgroups (http://msdn.microsoft.com/newsgroups/)
Provides a way to connect as a community with experts from around the world.

Microsoft ASP.NET (http://www.asp.net/)
Provides articles, demos, tool previews, and other information for Web development in JScript .NET.

Microsoft Windows Script (http://msdn.microsoft.com/scripting/)
Contains articles, samples, and other information of interest to JScript 5.6 developers.

In Newsgroups

microsoft.public.dotnet.languages.jscript
Provides a forum for questions and general discussion about JScript .NET.

microsoft.public.dotnet.scripting
Provides a forum for questions and general discussion about script with the .NET Framework.

microsoft.public.vsnet.documentation
Provides a forum for questions and issues on the JScript .NET documentation.

microsoft.public.scripting.jscript
Provides a forum for questions and general discussion about JScript 5.*x*.

microsoft.public.scripting.wsh
Provides a forum for questions and general discussion about using JScript 5.*x* in Microsoft Windows Script Host (WSH).

Other Resources

On the Web

4GuysFromRolla.com (http://www.4GuysFromRolla.com/)
Provides articles, demos, tool previews, and other information for Web development in JScript .NET.

DevX (http://www.devx.com/)
Provides articles, demos, tool previews, and other information for development in JScript .NET.

Writing, Compiling, and Debugging JScript Code

The Visual Studio Integrated Development Environment (IDE), which is the common development environment for all languages, provides tools and validation schemes that help you write reliable code. The IDE also provides debugging features that help you reconcile inconsistencies and resolve coding mistakes.

In This Section

Writing JScript Code with Visual Studio .NET
: Explains how to use the Visual Studio .NET Integrated Development Environment (IDE) to write and edit JScript code.

Compiling JScript Code from the Command Line
: Describes how to use the command-line compiler to produce compiled JScript .NET programs.

Conditional Compilation
: Describes how and when to use conditional compilation, which enables various sections of code to be included at compile time for debugging purposes, and facilitates the use of new JScript features without sacrificing backward compatibility.

Detecting Browser Capabilities
: Describes how to determine what versions of JScript a Web browser engine supports by using script engine functions and conditional compilation.

Copying, Passing, and Comparing Data
: Illustrates the difference between storing data by reference or by value and how these alternatives depend on the type of data.

Debugging JScript with Visual Studio .NET
: Lists procedures that enable debugging for a command-line program or for an ASP.NET program.

Debugging JScript with the Common Language Runtime Debugger
: Lists procedures that enable use of the common language runtime compiler debugger for a command-line program or for an ASP.NET program.

Troubleshooting Your Scripts
: Provides hints and tips to avoid common script mistakes in specific areas, such as syntax, order of script interpretation, automatic type coercion, and so on.

Related Sections

Building from the Command Line
: Explains how to invoke the compiler from the command line and provides examples of command-line syntax that return specific results.

Debugging and Profiling Applications
: Describes the process of debugging a .NET Framework application and how to optimize applications using the .NET Framework if Visual Studio is not installed.

Compiling JScript Code from the Command Line

To produce an executable JScript program, you must use the command-line compiler, jsc.exe. There are several ways to start the compiler.

The Visual Studio .NET Command Prompt

If Visual Studio .NET is installed, you can use the Visual Studio .NET Command Prompt to access the compiler from any directory on your machine. The Visual Studio .NET Command Prompt is in the Visual Studio .NET Tools program folder within the Microsoft Visual Studio .NET program group.

As an alternative, you can start the compiler from a Windows command prompt, which is the typical procedure if Visual Studio .NET is not installed.

The Windows Command Prompt

To start the compiler from a Windows command prompt, you must run the application from within its directory or type the fully qualified path to the executable at the command prompt. To override this default behavior, you must modify the PATH environment variable, which enables you to run the compiler from any directory by simply typing the compiler name.

To modify the PATH Environment Variable

1. Use the Windows Search feature to find jsc.exe on your local drive. The exact name of the directory where jsc.exe is located depends on the name and location of the Windows directory and the version of the .NET Framework installed. If you have more than one version of the .NET Framework installed, you must determine which version to use (typically the latest version).

   For example, the compiler may be located in `C:\WINNT\Microsoft.NET\Framework\v1.0.2914`.

2. Right-click the **My Computer** icon on your Desktop (Windows 2000), and select **Properties** from the shortcut menu.
3. Select the **Advanced** tab and click the **Environment Variables** button.
4. In the **System variables** pane, select "Path" from the list and click **Edit**.
5. In the **Edit System Variable** dialog box, move the cursor to the end of the string in the **Variable Value** field and type a semicolon (`;`) followed by the full directory name found in Step 1.

   Continuing with the example in Step 1, you would type:

   `;C:\WINNT\Microsoft.NET\Framework\v1.0.2914`

6. Click **OK** to confirm your edits and close the dialog boxes.

After you change the PATH environment variable, you can run the JScript .NET compiler at the Windows command prompt from any directory on the machine.

Using the Compiler

The command-line compiler has some built-in help. A help screen is displayed by using the **/help** or **/?** command-line option or by using the compiler without any options. For example:

```
jsc /help
```

There are two ways to use JScript .NET. You can write programs to be compiled from the command line, or you can write programs to be run in ASP.NET.

To compile using jsc

- At the command prompt, type `jsc file.js`

  The command compiles the program `named file.js` to produce the executable file named `file.exe`.

To produce a .dll file using jsc

- At the command prompt, type `jsc /target:library file.js`

  The command compiles the program named `file.js` with the **/target:library** option to produce the library file named `file.dll`.

To produce an executable with a different name using jsc

- At the command prompt, type `jsc /out:newname.exe file.js`

  The command compiles the program named `file.js` with the **/out:** option to produce the executable named `newname.exe`.

To compile with debugging information using jsc

- At the command prompt, type `jsc /debug file.js`

  The command compiles the program named `file.js` with the **/debug** option to produce the executable named `file.exe` and a file named `file.pdb` that contains debugging information.

There are many more command-line options available for the JScript command-line compiler. For more information, see JScript Compiler Options.

See Also

Writing, Compiling, and Debugging JScript Code | JScript Compiler Options | Conditional Compilation

Writing JScript Code with Visual Studio .NET

The Visual Studio Integrated Development Environment (IDE) provides some of the same tools for JScript development that are available for other languages. The **New File** dialog box includes several templates that provide a framework for various JScript files.

To write new JScript code using Visual Studio .NET

1. Start Microsoft Visual Studio .NET.
2. From the **File** Menu, click **New**, and then click **File**.
3. In the **Categories** dialog box, click the **Script** folder.
4. In the **Templates** dialog box, choose **JScript File** or **JScript .NET Web Form** and click **Open**.

To edit JScript code using Visual Studio .NET

1. Start Microsoft Visual Studio .NET.
2. From the **File** Menu, click **Open**, and then click **File**.
3. In the **Open File** dialog box, browse to your source file, select it and click **Open**.

Keywords are colorized according to the language used in each file. As you edit code, context-applicable help will appear in the **Dynamic Help** window.

To save JScript code in Visual Studio .NET

- From the **File** Menu, click **Save <*Filename*>** or **Save <*Filename*> As...**.

> **Note** You cannot compile JScript .NET code in the Visual Studio .NET IDE. This step must be performed from the command line or by the ASP.NET page.

See Also

Writing, Compiling, and Debugging JScript Code | Developing with Visual Studio .NET | Using IntelliSense |
Building from the Command Line | Debugging JScript with Visual Studio .NET |
Debugging JScript with the Common Language Runtime Debugger

Conditional Compilation

Conditional compilation enables JScript to use new language features without sacrificing compatibility with older versions that do not support the features. Some typical uses for conditional compilation include using new features in JScript, embedding debugging support into a script, and tracing code execution.

In This Section

Conditional Compilation Directives
Lists the directives that control how a script is compiled and links to information that explains the proper syntax for each directive.

Conditional Compilation Statements
Lists and discusses the statements that control compilation of a script depending on the values of the conditional compilation variables.

Conditional Compilation Variables
Lists the predefined variables available for conditional compilation.

Related Sections

Writing, Compiling, and Debugging JScript Code
Links to topics that explain how to use the Integrated Development Environment (IDE) to write JScript code.

Compiling JScript Code from the Command Line
Describes how to use the command-line compiler to produce compiled JScript programs.

Detecting Browser Capabilities
Describes how to determine what versions of JScript a Web browser engine supports by using script engine functions and conditional compilation.

JScript Compiler Options
Links to information that lists the compiler options available for the JScript command-line compiler.

Conditional Compilation Directives

The following directives modify the default behavior when compiling code for debugging.

Directive	Description
@debug	Turns on or off the emission of debug symbols.
@position	Provides meaningful position information in error messages .

These directives are provided for developers designing code that is automatically included into JScript programs by host environments (such as ASP.NET.). The code included is generally of no interest to authors who write scripts to be run in that environment. When these authors debug their code, they only want to see the parts of the program that they wrote rather than their code and additional parts that their development tool wrote.

These conditional compilation directives can "hide" code by turning off the emission of debug symbols and resetting the line position. This allows script authors to see only code they write when debugging a script.

See Also

Conditional Compilation | Conditional Compilation Variables | Conditional Compilation Statements | /debug

Conditional Compilation Statements

The following statements enable JScript to control the compilation of a script depending on the values of the conditional compilation variables. You can use the variables provided by JScript, or you can define your own with the **@set** directive or the **/define** command-line option.

Statement	Description
@cc_on	Activates conditional compilation support.
@if	Conditionally executes a group of statements, depending on the value of an expression .
@set	Creates variables used with conditional compilation statements.

The **@cc_on**, **@if**, or **@set** statements activate conditional compilation. Some typical uses for conditional compilation include using new features in JScript, embedding debugging support into a script, and tracing code execution.

When writing scripts to be run by Web browsers, always place conditional compilation code in comments. Consequently, hosts that do not support conditional compilation can ignore it. Here is an example.

```
/*@cc_on @*/
/*@if (@_jscript_version >= 5)
document.write("JScript Version 5.0 or better.<BR>");
@else @*/
document.write("You need a more recent script engine.<BR>");
/*@end @*/
```

This example uses special comment delimiters that are only used if the **@cc_on** statement has activated conditional compilation. Scripting engines that do not support conditional compilation display a message advising of the need for a new scripting engine without generating errors. Engines that support conditional compilation compile either the first or second `document.write`, depending on the version of the engine. Note that version 7.*x* represents JScript .NET. For more information, see Detecting Browser Capabilities.

Conditional compilation is also useful for server-side scripts and command-line programs. In those applications, conditional compilation can be used to compile additional functions into a program to help with profiling when in debug mode.

See Also

Conditional Compilation | Conditional Compilation Variables | Conditional Compilation Directives | /define |
Detecting Browser Capabilities

Conditional Compilation Variables

The following predefined variables are available for conditional compilation.

Variable	Description
@_win32	**true** if running on a Win32 system, otherwise **NaN**.
@_win16	**true** if running on a Win16 system, otherwise **NaN**.
@_mac	**true** if running on an Apple Macintosh system, otherwise **NaN**.
@_alpha	**true** if running on a DEC Alpha processor, otherwise **NaN**.
@_x86	**true** if running on an Intel processor, otherwise **NaN**.
@_mc680x0	**true** if running on a Motorola 680x0 processor, otherwise **NaN**.
@_PowerPC	**true** if running on a Motorola PowerPC processor, otherwise **NaN**.
@_jscript	Always **true**.
@_jscript_build	The build number of the JScript scripting engine.
@_jscript_version	A number representing the JScript version number in major.minor format.
@_debug	**true** if compiled in debug mode, otherwise **false**.
@_fast	**true** if compiled in fast mode, otherwise **false**.

> **Note** The version number reported for JScript .NET is 7.*x*.

Before using a conditional compilation variable, conditional compilation must be turned on. The **@cc_on** statement can turn on conditional compilation. Conditional compilation variables are often used in scripts written for Web browsers. It is not so common to use conditional compilation variables in scripts written for ASP or ASP.NET pages or command-line programs since the capabilities of the compilers can be determined using other methods.

When writing a script for a Web page, always place conditional compilation code in comments. This allows hosts that do not support conditional compilation to ignore it. Here is an example.

```
/*@cc_on
  document.write("JScript version: " + @_jscript_version + ".<BR>");
  @if (@_win32)
     document.write("Running on 32-bit Windows.<BR>");
  @elif (@_win16)
     document.write("Running on 16-bit Windows.<BR>");
  @else
     document.write("Running on a different platform.<BR>");
  @end
@*/
```

Conditional compilation variables can be used to determine the version information of the engine interpreting a script. This allows a script to take advantage of the features available in the latest versions of JScript while maintaining backwards compatibility. For more information, see Detecting Browser Capabilities.

See Also

Conditional Compilation | Conditional Compilation Directives | Conditional Compilation Statements | Detecting Browser Capabilities

Detecting Browser Capabilities

Although browsers support most JScript .NET features, the new features that target the .NET Framework, class-based objects, data types, enumerations, conditional compilation directives, and the **const** statement, are supported only on the server-side. Consequently, you should use these features exclusively in server-side scripts. For more information, see Version Information.

A JScript .NET script can detect the capabilities of the engine that interprets or compiles it. This is unnecessary if you are writing code for a server-side application (to be run in ASP or ASP.NET) or a command-line program since you can easily discover the supported JScript version and code accordingly. However, when running client-side scripts in a browser, this detection is important to ensure that the script is compatible with the JScript engine in the browser.

There are two ways to check for JScript compatibility, either by using the script engine functions or by using conditional compilation. There are advantages to using both approaches.

Script Engine Functions

The script engine functions (**ScriptEngine**, **ScriptEngineBuildVersion**, **ScriptEngineMajorVersion**, **ScriptEngineMinorVersion**) return information about the current version of the script engine. For more information, see Functions.

For the most compatibility, only features found in JScript Version 1 should be used in a page that checks supported JScript versions. If an engine supports a version of JScript higher than 1.0, you can redirect to another page that includes the advanced features. This means that you must have a separate version of each Web page corresponding to each version of JScript you want to support. In most situations, the most efficient solution is to have only two pages, one designed for a particular version of JScript, and the other designed to work without JScript.

> **Note** JScript code that uses advanced features must be placed in a separate page that is not run by browsers with incompatible engines. This is mandatory because the script engine of a browser interprets all the JScript code contained in a page. Using an **if...else** statement to switch between a block of code that uses the latest version of JScript and a block of JScript version 1 code will not work for older engines.

The following example illustrates the use of the script engine functions. Since these functions were introduced in JScript Version 2.0, you must first determine if the engine supports the functions before attempting to use them. If the engine supports only JScript Version 1.0 or does not recognize JScript, the **typeof** operator will return the string "undefined" for each function name.

```
if("undefined" == typeof ScriptEngine) {
   // This code is run if the script engine does not support
   // the script engine functions.
   var version = 1;
} else {
   var version = ScriptEngineMajorVersion();
}
// Display the version of the script engine.
alert("Engine supports JScript version " + version);
// Use the version information to choose a page.
if(version >= 5) {
   // Send engines compatible with JScript 5.0 and better to one page.
   var newPage = "webpageV5.htm";
} else {
   // Send engines that do not interpret JScript 5.0 to another page.
   var newPage = "webpagePre5.htm";
}
location.replace(newPage);
```

Conditional Compilation

Conditional compilation variables and statements can hide JScript code from engines that do not support conditional compilation. This approach is useful if you want to include a small amount of alternate code directly in the Web page.

> **Note** Do not use multiline comments within the conditional compilation blocks since engines that do not support conditional compilation may misinterpret them.

```
<script>
/*@cc_on
```

```
@if(@_jscript_version >= 5 )
// Can use JScript Version 5 features such as the for...in statement.
// Initialize an object with an object literal.
var obj = {"a" : "Athens" , "b" : "Belgrade", "c" : "Cairo"};
var key;
// Iterate the properties.
for (key in obj) {
   document.write("The "+key+" property has value "+obj[key]+".<BR>");
}
@else
@*/
alert("Engine cannot interpret JScript Version 5 code.");
//@end
</script>
```

If the conditional **@if** block includes a lot of code, it may be easier to use the approach outlined above for using the script engine functions.

See Also

Writing, Compiling, and Debugging JScript Code | Version Information | Functions | Conditional Compilation

Copying, Passing, and Comparing Data

The way that JScript copies, passes, and compares data depends on how the data is stored, which in turn depends on the type of the data. JScript stores data either by value or by reference.

By Value vs. By Reference

JScript copies, passes, and compares numbers and Boolean values (**true** and **false**) *by value*. This process allocates a space in computer memory and copies the value of the original into it. Changes to the original do not affect the copy (and vice versa) because the two are separate entities. Two numbers or Boolean values are considered equal if they have the same value.

JScript copies, passes, and compares objects, arrays, and functions *by reference*. This process essentially creates a reference to the original item and uses the reference as if it were a copy. Changes to the original change both the original and the copy (and vice versa). There is really only one entity; the copy is just another reference to the data.

To successfully compare by reference, the two variables must refer to exactly the same entity. For example, two distinct **Array** objects will always compare as unequal, even if they contain the same elements. One of the variables must be a reference to the other one for the comparison to succeed. To check if two Arrays hold the same elements, compare the results of the **toString()** method.

Finally, JScript copies and passes strings by reference. Whether or not the strings are objects determines how the strings are compared. Two **String** objects (created with **new String**("something")) are compared by reference. If one (or both) of the strings is a literal or primitive string value, they are compared by value. For more information, see JScript Assignments and Equality.

> **Note** The ASCII and ANSI character sets are constructed so that capital letters precede lowercase ones in sequence order. For example, "Zoo" compares as *less* than "aardvark." You can call **toUpperCase()** or **toLowerCase()** on both strings if you want to perform a case-insensitive match.

Function Parameters

When JScript passes a parameter to a function by value, it makes a separate copy of that parameter that exists only inside the function. Even though objects and arrays are passed by reference, if a new value in the function directly overwrites them, the new value is not reflected outside the function. Only changes to properties of objects, or elements of arrays, are visible outside the function.

For example, the following program has two functions. The first function overwrites the input parameter, which prevents further the changes to the parameter from affecting the original input argument. The second function changes the property of the object without overwriting the object.

```
function clobber(param) {
   // Overwrite the parameter; this will not be seen in the calling code
   param = new Object();
   param.message = "This will not work.";
}

function update(param) {
    // Modify the property of the object; this will be seen in the calling code.
    param.message = "I was changed.";
}

// Create an object, and give it a property.
var obj = new Object();
obj.message = "This is the original.";

// Call clobber, and print obj.message.
clobber(obj);
print(obj.message);

// Call update, and print obj.message.
update(obj);
print(obj.message);
```

The output of this code is:

```
This is the original.
```

```
I was changed.
```

Data Comparison

JScript can compare data either by value or by reference. To perform a test by value, JScript compares two distinct items to determine whether they are equal to each other. Usually, this comparison is performed on a byte-by-byte basis. When it tests by reference, it checks to see whether two items refer to the same item. If they do, then they compare as equal; if not, although they may contain the exact same values, byte-for-byte, they compare as unequal.

Strings may be compared by value or by reference, depending on whether or not the strings are objects. If both strings are **String** objects, the strings are compared by reference; otherwise, they are compared by value. This allows two strings to compare as equal if each was created separately from the other but each has the same content. To compare the values of two **String** objects, first convert the objects to non-object strings with the **toString** or **valueOf** methods, and then compare the resulting strings. For more information, see JScript Assignments and Equality.

See Also

JScript Language Tour | JScript Functions | JScript Assignments and Equality | Data Type Summary

Debugging JScript with Visual Studio .NET

Some JScript programs are designed to run from the command line, while others are designed to run in an ASP.NET page. The type of program influences the debugging method.

To set up debugging for a command line program

1. Compile the program that you intend to debug using the **/debug** flag. For more information, see /debug.
2. Start Microsoft Visual Studio .NET.
3. From the **File** Menu, click **Open**, and then click **Project**.
4. In the **Open Project** dialog box, browse to your compiled program (the file with the .exe extension), select it and click **Open**.
5. From the **File** Menu, click **Open**, and then click **File**.
6. In the **Open File** dialog box, browse to your source code (the file with the .js extension), select it and click **Open**.
7. From the **File** Menu, click **Save All**.
8. Choose a name and location to save your new project.

After this setup is complete, you can proceed to the Debugging using Visual Studio .NET section.

To set up debugging for an ASP.NET program

1. Start Microsoft Visual Studio .NET.
2. Open the ASP.NET file that you intend to debug.
3. Set the debug flag to true in the **@page** directive. For example:

   ```
   <%@page Language=jscript debug=true %>
   ```

4. Open the page in a browser to compile the page.
5. From the Visual Studio **Tools** Menu, click **Debug Processes**.
6. In the **Processes** dialog box, select the **Show system processes** and **Show processes in all sessions** options.
7. In the **Available Processes** pane of the **Processes** dialog box, select the ASP.NET worker process that runs that Web application, and click **Attach**.

   By default, the worker process is aspnet_wp.exe for IIS 5.*x* (on Windows 2000 and Windows XP), and w3wp.exe for IIS 6.0 (on Windows Server 2003).

8. In the **Attach to Process** dialog box, select **Common Language Runtime** and click **Ok**.
9. In the **Processes** dialog box, click **Close**.

After this setup is complete, you can proceed to the Debugging using Visual Studio .NET section.

To debug using Visual Studio .NET

1. In the Visual Studio .NET IDE, open the file that you intend to debug, as described in either of the above setup sections.
2. Move the cursor to the location in the file where you want to set a breakpoint, and press **F9**.
3. Repeat the previous step to add more breakpoints.
4. From the **Debug** Menu, click **Start**.

   The program will run until it encounters a breakpoint or produces a runtime error.

5. At this point, several windows will open, allowing you to perform further debugging tasks.For more information, see Using the Debugger.
6. To stop debugging but leave the program running, from the **Debug** Menu, choose **Detach All**.

   Otherwise, the program will be terminated when you stop debugging.

Remarks

When debugging program compiled from the command line, Visual Studio .NET rereads the compiled program each time you start debugging. Consequently, you can modify your JScript code and (after recompiling the code) check the effect of those changes.

See Also

Using the Debugger | Building from the Command Line | Writing JScript Code with Visual Studio .NET | Building from the Command Line | Debugging JScript with the Common Language Runtime Debugger | Debugging Script and Web Applications

Debugging JScript with the Common Language Runtime Debugger

Some JScript programs are designed to run from the command line, while others are designed to run in an ASP.NET page. The type of program influences the debugging method.

The common language runtime debugger, dbgclr.exe, is located in the GuiDebug directory of the .NET Framework installation.

To use dbgclr.exe, you must either qualify the program name with the path name or add the path to the search path.

To set up debugging for a command line program

1. Write your program in any editor and save it as text.
2. Compile the program using the /debug flag. For more information, see /debug.
3. Start dbgclr, the common language runtime debugger.
4. From the **File** Menu of dgbclr, click **Open**, and then click **File**.
5. In the **Open File** dialog box, open the source file (the file with the .js extension) that you want to debug.
6. From the **Debug** Menu, click **Program to Debug**.
7. In the **Program To Debug** dialog box, click the ellipses (...) adjacent to the **Program** pane.
8. In the **Find Program to Debug** window, browse to your compiled program (the file with the .exe extension), select it and click **Open**.
9. In the **Program To Debug** dialog box, click **OK**.

After this setup is complete, you can proceed to the To debug using the Common Language Runtime Debugger section.

To set up debugging for an ASP.NET program

1. Write your program in any editor and save it as text.
2. Write the HTML wrapper for the ASP.NET. Be sure to specify that you want to debug the JScript code by including this line in your code:

   ```
   <%@page Language=jscript debug=true %>
   ```

3. Open the page in a browser to compile the page.
4. Start dbgclr, the common language runtime debugger.
5. From the **Tools** Menu of dgbclr, click **Debug Processes**.
6. In the **Processes** window, select both **Show system processes** and **Show processes in all sessions**.
7. In the **Available Processes** dialog box, select the ASP.NET worker process that runs that Web application, click **Attach**, and click **Close**.

   By default, the worker process is aspnet_wp.exe for IIS 5.*x* (on Windows 2000 and Windows XP), and w3wp.exe for IIS 6.0 (on Windows Server 2003).

8. From the **File** Menu, click **Open**, and then click **File**.
9. In the **Open File** window, browse to your source code, select it, and click **Open**.

After this setup is complete, you can proceed to the To debug using the Common Language Runtime Debugger section.

To debug using the Common Language Runtime Debugger

1. Move the cursor to the location in the file where you want to set a breakpoint, and press **F9**.
2. Repeat the previous step to add more breakpoints.
3. From the **Debug** Menu, click **Start**.

   The program will run until it finds a breakpoint or a run-time error. At this point, several windows will open, allowing you to perform further debugging tasks.

4. To stop debugging but leave the program running, from the **Debug** Menu, choose **Detach All**.

   Otherwise, the program will be terminated when you stop debugging.

Remarks

When debugging program compiled from the command line, dgbclr rereads the compiled program each time you start debugging. Consequently, you can modify your JScript code and (after recompiling the code) check the effect of those changes.

See Also

Using the Debugger | Writing JScript code with Visual Studio .NET | Building from the Command Line | Debugging JScript with Visual Studio .NET | CLR Debugger

Troubleshooting Your Scripts

All programming languages include potential pitfalls and surprises for novice and experienced users. Here are some potential trouble areas that you may encounter as you write JScript scripts.

Syntax Errors

Because syntax is much more rigid in programming languages than in natural languages, it is important to pay strict attention to detail when you write scripts. If, for example, you intend that a particular parameter be a string, you will encounter trouble if you forget to enclose it in quotation marks.

Order of Script Interpretation

In a Web page, JScript interpretation depends on each browser's HTML parsing process. A script inside the <HEAD> tag is interpreted before text within the <BODY> tag. Consequently, objects that are created in the <BODY> tag do not exist when the browser parses the <HEAD> element and cannot be manipulated by the script.

> **Note** This behavior is specific to Internet Explorer. ASP and WSH have different execution models (as would other hosts).

Automatic Type Coercion

JScript is a loosely typed language with automatic coercion. Consequently, despite the fact that values having different types are not strictly equal, the expressions in the following example evaluate to **true**.

```
"100" == 100;
false == 0;
```

To check that both the type and value are the same, use the strict equality operator, ===. The following both evaluate to false:

```
"100" === 100;
false === 0;
```

Operator Precedence

The order of operation execution during the evaluation of an expression depends more on operator precedence than on the order of operators in the expression. Thus, in the following example, multiplication is performed before subtraction even though the subtraction operator appears before the multiplication operator in the expression.

```
theRadius = aPerimeterPoint - theCenterpoint * theCorrectionFactor;
```

For more information, see Operator Precedence.

Using for...in Loops with Objects

When a script steps through the properties of an object with a **for...in** loop, the order in which the fields of the object are assigned to the loop counter variable are not necessarily predictable or controllable. Moreover, the order may be different in different implementations of the language. For more information, see for...in Statement.

with Keyword

Although the **with** keyword is convenient for addressing properties that already exist in a specified object, it cannot be used to add properties to an object. To create new properties in an object, you must refer to the object specifically. For more information, see with Statement.

this Keyword

Although the **this** keyword exists inside the definition of an object, you cannot ordinarily use **this** or similar keywords to refer to the currently executing function if the function is not an object definition. If the function is to be assigned to an object as a method, a script can use the **this** keyword within the function to refer to the object. For more information, see this Statement.

Writing a Script That Writes a Script in Internet Explorer or ASP.NET

The </SCRIPT> tag terminates the current script if the interpreter encounters it. To display "</SCRIPT>" itself, write this as two or more strings, for example, "</SCR" and "IPT>", which the script can then concatenate in the statement that writes them.

Implicit Window References in Internet Explorer

Because more than one window can be simultaneously open, any window reference that is implicit points to the current window. For other windows, you must use an explicit reference.

See Also

Writing, Compiling, and Debugging JScript Code | Writing JScript Code with Visual Studio .NET |
Debugging JScript with Visual Studio .NET

Displaying Information with JScript .NET

Programs typically display information to a user. The most basic method is to display a text string. A JScript program may display information through different procedures depending on the use of the program. There are three common ways to use JScript: in an ASP.NET page, in a client-side Web page, and from the command line. Some methods for displaying from each environment are discussed here.

In This Section

Displaying from a Command-Line Program
: Explains how to use the JScript **print** statement or the .NET Framework **System.Windows.Forms.MessageBox.Show** method to display data from a command-line program.

Displaying from ASP.NET
: Demonstrates how to use the **<%= %>** construct to display data from an ASP.NET program.

Displaying Information in the Browser
: Explains how to use the **write** or **writeln** methods to display data directly in a browser.

Using Message Boxes
: Describes how to use the **alert**, **confirm**, and **prompt** methods of the window object to create prompt message boxes, which typically solicit input from users.

Related Sections

JScript Reference
: Lists elements that comprise the JScript Language Reference and provides an example of the correct syntax for each element.

MessageBox.Show Method
: Illustrates the syntax of the **MessageBox.Show** method and the options that return various results.

Displaying from ASP.NET

There are several ways to display information from an ASP.NET program. One approach is to use the **<%= %>** construction. Another approach is to use the **Response.Write** statement.

Using <%= %>

The simplest way to display information from an ASP.NET program is to use the **<%= %>** construct. The value that is entered after the equals sign is written into the current page. The following code displays the value of the variable `name`.

```
Hello <%= name %>!
```

If the value of name were "Frank", the code would write the following string in the current page:

```
Hello Frank!
```

The **<%= %>** construct is most useful for displaying single pieces of information.

The Response.Write Statement

Another way to display text is to use the **Response.Write** statement. It can be enclosed within a **<% %>** block.

```
<% Response.Write("Hello, World!") %>
```

The **Response.Write** statement can also be used in a function or method within a script block. The following example shows a function that includes a **Response.Write** statement.

> **Note** In ASP.NET pages, functions and variables should be defined within **<script>** blocks, while executable code must be enclosed within **<% %>** blocks.

```
<script runat="server" language="JScript">
   function output(str) {
      Response.Write(str);
   }
   var today = new Date();
</script>
Today's date is <% output(today); %>. <BR>
```

The output of the **Response.Write** statement is incorporated into the page being processed. This allows the output of **Response.Write** to write code that in turn displays text. For example, the following code writes a script block that displays the current date (on the server) in an alert window of the browser accessing the page. The "<script>" tag is split so the server will not process the tag.

```
<script runat="server" language="JScript">
   function popup(str) {
      Response.Write("<scr"+"ipt> alert('"+str+"') </scr"+"ipt>");
   }
   var today = new Date();
</script>
<% popup(today); %>
```

For more information, see Page.Response Property.

See Also

Displaying Information with JScript .NET | Introduction to ASP.NET | Page.Response Property

Displaying from a Command-Line Program

There are three ways that JScript displays data from a command-line program. The Microsoft JScript command-line compiler provides the **print** statement. The class **System.Console** provides additional methods that facilitate interaction with the user using the console.

The **System.Windows.Forms.MessageBox.Show** method displays information and receives input from pop-up boxes.

The print Statement

The most common way to display information is the **print** statement. It takes one argument, a string, which it displays followed by a newline character in the command-line window.

Either single or double quotation marks can enclose strings, which permits quotes that contain quote marks or apostrophes.

```
print("Pi is approximately equal to " + Math.PI);
print();
```

> **Note** The **print** statement is only available for programs compiled with the JScript command-line compiler. Using **print** in an ASP.NET page causes a compiler error.

The Console Class

The class **System.Console** exposes methods and properties that facilitate interaction with users of the console. The **WriteLine** method of the **Console** class provides functionality similar to the **print** statement. The **Write** method displays a string without appending a newline character. Another useful method of the **Console** class is the **ReadLine** method, which reads a line of text that is entered from the console.

To use classes and methods from the .NET Framework, first use the **import** statement to import the namespace to which the class belongs. To call the method, use either the fully qualified name or just the name if there is no method in the current scope with the same name.

```
import System;
System.Console.WriteLine("What is your name: ");
var name : String = Console.ReadLine();
Console.Write("Hello ");
Console.Write(name);
Console.WriteLine("!");
```

The program requests that a name be entered from the console. After entering the name, Pete, the program displays:

```
What is your name:
Pete
Hello Pete!
```

For more information, see Console Class.

The Show Method

The **System.Windows.Forms.MessageBox.Show** method is versatile because it is overloaded. The simplest overload has one argument, the string of text you want to display. The message box is modal.

> **Note** A window or form is modal if it retains the focus until you explicitly close it. Dialog boxes and messages are usually modal. For example, in a modal dialog box, you cannot access another window until you choose **OK** in the dialog box.

```
import System.Windows.Forms;
System.Windows.Forms.MessageBox.Show("Welcome! Press OK to continue.");
MessageBox.Show("Great! Now press OK again.");
```

You can use other overloads of the **Show** method to include a caption, other buttons, an icon, or default button. For more information, see MessageBox.Show Method.

See Also

Displaying Information with JScript .NET | print Statement | import Statement

Displaying Information in the Browser

Although browsers support most JScript .NET features, the new features that target the .NET Framework, class-based objects, data types, enumerations, conditional compilation directives, and the **const** statement, are supported only on the server-side. Consequently, you should use these features exclusively in server-side scripts. For more information, see Version Information.

Whenever a script is intended to run in a browser (client-side), experienced developers include code that detects the version of the script engine. After the script detects the engine version, it can redirect the browser to a page with script that is compatible with the browser's script engine. For more information, see Detecting Browser Capabilities.

JScript displays information in a browser through the **write** and **writeln** methods of the browser's document object. It can also display information in forms within a browser and in **alert**, **prompt**, and **confirm** message boxes. For more information, see Using Message Boxes.

Using document.write and document.writeln

The most common way to display information is the **write** method of the **document** object. It takes one argument, a string, which it displays in the browser. The string can be either plain text or HTML.

Since strings can be enclosed in either single or double quotation marks, you can quote something that contains quote marks or apostrophes.

```
document.write("Pi is approximately equal to " + Math.PI);
document.write();
```

Note The following simple function enables you to avoid typing `document.write` every time you want text to appear in the browser window. This function does not inform you if something that you attempt to write is undefined, but it does let you issue the command `w();`, which displays a blank line.

```
function w(m) { // Write function.
   m = String(m); //  Make sure that the m variable is a string.
   if ("undefined" != m) { // Test for empty write or other undefined item.
      document.write(m);
   }
   document.write("<br>");
}

w('<IMG SRC="horse.gif">');
w();
w("This is an engraving of a horse.");
w();
```

The **writeln** method, which is almost identical to the **write** method, appends a newline character to the provided string. In HTML this ordinarily results only in a space after an item; within <PRE> and <XMP> tags, the newline character is interpreted literally and the browser displays it.

The **write** method opens and clears the document if the document is not in the process of being opened and parsed when the **write** method is called. This poses potentially unexpected results. The following example, which shows a script that is intended to display the time once a minute, fails to do so after the first time because it clears itself in the process.

```
<HTML>
<HEAD>
<SCRIPT LANGUAGE="JScript">
function singOut()  {
   var theMoment = new Date();
   var theHour = theMoment.getHours();
   var theMinute = theMoment.getMinutes();
   var theDisplacement = (theMoment.getTimezoneOffset() / 60);
   theHour -= theDisplacement;
   if (theHour > 23)  {
      theHour -= 24
   }
   // The following line clears the script the second time it is run.
```

```
    document.write(theHour + " hours, " + theMinute + " minutes, Coordinated Universal Time.")
;
    window.setTimeout("singOut();", 60000);
}
</SCRIPT>
</HEAD>
<BODY>
<SCRIPT>
singOut();
</SCRIPT>
</BODY>
</HTML>
```

If you use the **alert** method of the window object instead of **document.write**, the script works.

```
    // This line produced the intended result.
    window.alert(theHour + " hours, " + theMinute + " minutes, Coordinated Universal Time.");
```

The **element.innerText** or **element.innerHTML** is preferred in Internet Explorer Version 5 and greater.

Clearing the Current Document

The **clear** method of the **document** object empties the current document. This method also clears your script (along with the rest of the document), so be very careful how and when you use it.

```
document.clear();
```

See Also

Displaying Information with JScript .NET | Using Message Boxes | Detecting Browser Capabilities

Using Message Boxes

Although browsers support most JScript .NET features, the new features that target the .NET Framework, class-based objects, data types, enumerations, conditional compilation directives, and the **const** statement, are supported only on the server-side. Consequently, you should use these features exclusively in server-side scripts. For more information, see Version Information.

Whenever a script is intended to run in a browser (client-side), experienced developers include code that detects the version of the script engine. After the script detects the engine version, it can redirect the browser to a page with script that is compatible with the browser's script engine. For more information, see Detecting Browser Capabilities.

JScript uses the **alert**, **confirm**, and **prompt** message boxes of the browser to obtain input from your user. The boxes are methods of the **window** object. Because the **window** object is at the top of the object hierarchy, you do not actually need to use the full name (for example, `window.alert()`) of any of these message boxes, but it is a good idea to do so because it helps you remember to which object they belong.

Alert Message Box

The **alert** method has one argument, the string of text that you want to display in the alert message box. The string is not HTML. The message box provides an OK button to close the message box and is modal, that is, the user must close the message box before continuing.

```
window.alert("Welcome! Press OK to continue.");
```

Confirm Message Box

The confirm message box, which includes OK and Cancel buttons, poses a question with two possible outcomes. The **confirm** method returns either **true** or **false**. This message box is also modal: the user must respond to it (click a button), and thereby close it, before proceeding.

```
var truthBeTold = window.confirm("Click OK to continue. Click Cancel to stop.");
if (truthBeTold)
   window.alert("Welcome to our Web page!");
else
   window.alert("Bye for now!");
```

Prompt Message Box

The prompt message box, which includes OK and Cancel buttons, provides a text field that accepts text in response to a prompt. If you provide a second string argument, the prompt message box displays the second string in the text field as the default response. Otherwise, the default text is "undefined".

Like the **alert** and **confirm** methods, **prompt** displays a modal message box. The user must close it before continuing.

```
var theResponse = window.prompt("Welcome?","Enter your name here.");
document.write("Welcome "+theResponse+".<BR>");
```

See Also

Displaying Information with JScript .NET | Displaying Information in the Browser | Detecting Browser Capabilities

Introduction to Regular Expressions

These thirteen sections introduce the concept of regular expressions and explain how to create and use them in JScript.

While each topic stands on its own, it is recommended that you peruse these topics sequentially to develop the best understanding of the material. Many topics rely upon the understanding of a feature or concept that was introduced in a previous topic in the sequence.

In This Section

Regular Expressions
: Explains the concept of regular expressions by comparing with concepts that are already familiar to most readers.

Uses for Regular Expressions
: Indicates how regular expressions extend conventional search criteria through practical examples.

Regular Expression Syntax
: Explains the characters that comprise a regular expression, the characters that comprise metacharacters, and the behavior of metacharacters.

Build a Regular Expression
: Describes the components of a regular expression and the relationship between the components and delimiters.

Order of Precedence
: Explains how regular expressions are evaluated and how the sequence and syntax of the regular expression effects the result.

Ordinary Characters
: Distinguishes ordinary characters from metacharacters and illustrates how to combine single-character regular expressions to create larger expressions.

Special Characters
: Explains the concept of escaping characters and how to create a regular expression that matches metacharacters.

Non-Printable Characters
: Lists the escape sequences that are used to represent non-printing characters in a regular expression.

Character Matching
: Illustrates how regular expressions use periods, escape characters, and brackets to create sequences that return specific results.

Quantifiers
: Explains how to create regular expressions when you cannot specify how many characters comprise a match.

Anchors
: Indicates how to fix a regular expression to either the beginning of a line or the end of a line and how to create regular expressions that occur within a word, at the beginning of a word, or at the end of a word.

Alternation and Grouping
: Illustrates how alternation uses the '|' character to allow a choice between two or more alternatives and how grouping works in conjunction with alternation to further refine the result.

Backreferences
: Explains how to create regular expressions that can access part of a stored matched pattern without recreating the regular expression that found the matched pattern.

Related Sections

.NET Framework Regular Expressions
: Clarifies how pattern-matching notation of regular expressions allows developers to quickly parse large amounts of text to find specific character patterns; to extract, edit, replace, or delete text substrings; or to add the extracted strings to a collection in order to generate a report.

Regular Expression Examples
: Provides a list of links to code examples that illustrate the use of regular expressions in common applications.

Regular Expressions

Unless you have previously used regular expressions, the term may be unfamiliar to you. However, you have undoubtedly used some regular expression concepts outside the area of scripting.

For example, you most likely use the ? and * wildcard characters to find files on your hard disk. The ? wildcard character matches a single character in a file name, while the * wildcard character matches zero or more characters. A pattern such as data?.dat would find the following files:

data1.dat

data2.dat

datax.dat

dataN.dat

Using the * character instead of the ? character expands the number of found files. data*.dat matches all of the following:

data.dat

data1.dat

data2.dat

data12.dat

datax.dat

dataXYZ.dat

While this method of searching is useful, it is also limited. The ability of the ? and * wildcard characters introduces the concept behind regular expressions, but regular expressions are more powerful and flexible.

See Also

Introduction to Regular Expressions

Uses for Regular Expressions

A typical search and replace operation requires you to provide the exact text that matches the intended search result. Although this technique may be adequate for simple search and replace tasks in static text, it lacks flexibility and makes searching dynamic text at least difficult if not impossible.

With regular expressions, you can:

- Test for a pattern within a string.

  For example, you can test an input string to see if a telephone number pattern or a credit card number pattern occurs within the string. This is called data validation.

- Replace text.

  You can use a regular expression to identify specific text in a document and either remove it completely or replace it with other text.

- Extract a substring from a string based upon a pattern match.

  You can find specific text within a document or input field.

For example, you may need to search an entire Web site, remove outdated material, and replace some HTML formatting tags. In this case, you can use a regular expression to determine if the material or the HTML formatting tags appears in each file. This process reduces the affected files list to those that contain material targeted for removal or change. You can then use a regular expression to remove the outdated material. Finally, you can use a regular expression to search for and replace the tags.

A regular expression is also useful in a language, such as JScript or C that is not known for its string-handling ability.

See Also

Introduction to Regular Expressions

Regular Expression Syntax

A regular expression is a pattern of text that consists of ordinary characters (for example, letters a through z) and special characters, known as *metacharacters*. The pattern describes one or more strings to match when searching text.

Here are some examples of regular expressions:

Expression	**Matches**
/^\s*$/	Match a blank line.
/\d{2}-\d{5}/	Validate an ID number consisting of 2 digits, a hyphen, and an additional 5 digits.
/<\s*(\S+)(\s[^>]*)?>[\s\S]*<\s*\/\1\s*>/	Match an HTML tag.

The following table contains the complete list of metacharacters and their behavior in the context of regular expressions:

Character	**Description**
\	Marks the next character as a special character, a literal, a backreference, or an octal escape. For example, 'n' matches the character "n". '\n' matches a newline character. The sequence '\\' matches "\" and "\(" matches "(".
^	Matches the position at the beginning of the input string. If the **RegExp** object's **Multiline** property is set, ^ also matches the position following '\n' or '\r'.
$	Matches the position at the end of the input string. If the **RegExp** object's **Multiline** property is set, $ also matches the position preceding '\n' or '\r'.
*	Matches the preceding character or subexpression zero or more times. For example, zo* matches "z" and "zoo". * is equivalent to {0,}.
+	Matches the preceding character or subexpression one or more times. For example, 'zo+' matches "zo" and "zoo", but not "z". + is equivalent to {1,}.
?	Matches the preceding character or subexpression zero or one time. For example, "do(es)?" matches the "do" in "do" or "does". ? is equivalent to {0,1}
{*n*}	*n* is a nonnegative integer. Matches exactly *n* times. For example, 'o{2}' does not match the 'o' in "Bob," but matches the two o's in "food".
{*n*,}	*n* is a nonnegative integer. Matches at least *n* times. For example, 'o{2,}' does not match the "o" in "Bob" and matches all the o's in "foooood". 'o{1,}' is equivalent to 'o+'. 'o{0,}' is equivalent to 'o*'.
{*n*,*m*}	*M* and *n* are nonnegative integers, where *n* <= *m*. Matches at least *n* and at most *m* times. For example, "o{1,3}" matches the first three o's in "fooooood". 'o{0,1}' is equivalent to 'o?'. Note that you cannot put a space between the comma and the numbers.
?	When this character immediately follows any of the other quantifiers (*, +, ?, {*n*}, {*n*,}, {*n*,*m*}), the matching pattern is non-greedy. A non-greedy pattern matches as little of the searched string as possible, whereas the default greedy pattern matches as much of the searched string as possible. For example, in the string "oooo", 'o+?' matches a single "o", while 'o+' matches all 'o's.
.	Matches any single character except "\n". To match any character including the '\n', use a pattern such as '[\s\S]'.
(*pattern*)	A subexpression that matches *pattern* and captures the match. The captured match can be retrieved from the resulting Matches collection using the **$0…$9** properties. To match parentheses characters ( ), use '\(' or '\)'.
(?:*pattern*)	A subexpression that matches *pattern* but does not capture the match, that is, it is a non-capturing match that is not stored for possible later use. This is useful for combining parts of a pattern with the "or" character (|). For example, 'industr(?:y|ies) is a more economical expression than 'industry|industries'.
(?=*pattern*)	A subexpression that performs a positive lookahead search, which matches the string at any point where a string matching *pattern* begins. This is a non-capturing match, that is, the match is not captured for possible later use. For example 'Windows (?=95|98|NT|2000)' matches "Windows" in "Windows 2000" but not "Windows" in "Windows 3.1". Lookaheads do not consume characters, that is, after a match occurs, the search for the next match begins immediately following the last match, not after the characters that comprised the lookahead.

(?!*pattern*)	A subexpression that performs a negative lookahead search, which matches the search string at any point where a string not matching *pattern* begins. This is a non-capturing match, that is, the match is not captured for possible later use. For example 'Windows (?!95|98|NT|2000)' matches "Windows" in "Windows 3.1" but does not match "Windows" in "Windows 2000". Lookaheads do not consume characters, that is, after a match occurs, the search for the next match begins immediately following the last match, not after the characters that comprised the lookahead.
x|*y*	Matches either *x* or *y*. For example, 'z|food' matches "z" or "food". '(z|f)ood' matches "zood" or "food".
[*xyz*]	A character set. Matches any one of the enclosed characters. For example, '[abc]' matches the 'a' in "plain".
[^*xyz*]	A negative character set. Matches any character not enclosed. For example, '[^abc]' matches the 'p' in "plain".
[*a-z*]	A range of characters. Matches any character in the specified range. For example, '[a-z]' matches any lowercase alphabetic character in the range 'a' through 'z'.
[^*a-z*]	A negative range characters. Matches any character not in the specified range. For example, '[^a-z]' matches any character not in the range 'a' through 'z'.
\b	Matches a word boundary, that is, the position between a word and a space. For example, 'er\b' matches the 'er' in "never" but not the 'er' in "verb".
\B	Matches a nonword boundary. 'er\B' matches the 'er' in "verb" but not the 'er' in "never".
\c*x*	Matches the control character indicated by *x*. For example, \cM matches a Control-M or carriage return character. The value of *x* must be in the range of A-Z or a-z. If not, c is assumed to be a literal 'c' character.
\d	Matches a digit character. Equivalent to [0-9].
\D	Matches a nondigit character. Equivalent to [^0-9].
\f	Matches a form-feed character. Equivalent to \x0c and \cL.
\n	Matches a newline character. Equivalent to \x0a and \cJ.
\r	Matches a carriage return character. Equivalent to \x0d and \cM.
\s	Matches any white space character including space, tab, form-feed, and so on. Equivalent to [\f\n\r\t\v].
\S	Matches any non-white space character. Equivalent to [^ \f\n\r\t\v].
\t	Matches a tab character. Equivalent to \x09 and \cI.
\v	Matches a vertical tab character. Equivalent to \x0b and \cK.
\w	Matches any word character including underscore. Equivalent to '[A-Za-z0-9_]'.
\W	Matches any nonword character. Equivalent to '[^A-Za-z0-9_]'.
\x*n*	Matches *n*, where *n* is a hexadecimal escape value. Hexadecimal escape values must be exactly two digits long. For example, '\x41' matches "A". '\x041' is equivalent to '\x04' & "1". Allows ASCII codes to be used in regular expressions.
\num	Matches *num*, where *num* is a positive integer. A reference back to captured matches. For example, '(.)\1' matches two consecutive identical characters.
\n	Identifies either an octal escape value or a backreference. If \n is preceded by at least *n* captured subexpressions, *n* is a backreference. Otherwise, *n* is an octal escape value if *n* is an octal digit (0-7).
\nm	Identifies either an octal escape value or a backreference. If \nm is preceded by at least *nm* captured subexpressions, *nm* is a backreference. If \nm is preceded by at least *n* captures, *n* is a backreference followed by literal *m*. If neither of the preceding conditions exists, \nm matches octal escape value *nm* when *n* and *m* are octal digits (0-7).
\nml	Matches octal escape value *nml* when *n* is an octal digit (0-3) and *m* and *l* are octal digits (0-7).
\u*n*	Matches *n*, where *n* is a Unicode character expressed as four hexadecimal digits. For example, \u00A9 matches the copyright symbol (©).

See Also

Introduction to Regular Expressions

Build a Regular Expression

The construction of regular expressions is analogous to the construction of arithmetic expressions. That is, a variety of metacharacters and operators combine small expressions to create large expressions.

You construct a regular expression by putting the various components of the expression pattern between a pair of delimiters. For JScript, the delimiters are forward slash (/) characters. For example:

```
/expression/
```

In the above example, the regular expression pattern (*expression*) is stored in the **Pattern** property of the **RegExp** object.

The components of a regular expression can be individual characters, sets of characters, ranges of characters, choices between characters, or any combination of all of these components.

See Also

Introduction to Regular Expressions

Order of Precedence

A regular expression is evaluated from left to right and follows an order of precedence, much like an arithmetic expression.

The following table illustrates, from highest to lowest, the order of precedence of the various regular expression operators:

Operator(s)	Description
\	Escape
(), (?:), (?=), []	Parentheses and Brackets
*, +, ?, {n}, {n,}, {n,m}	Quantifiers
^, $, \anymetacharacter, *anycharacter*	Anchors and Sequences
|	Alternation

Characters have higher precedence than the alternation operator, which allows 'm|food' to match "m" or "food". To match "mood" or "food", use parentheses to create a subexpression, which results in '(m|f)ood'.

See Also

Introduction to Regular Expressions

Ordinary Characters

Ordinary characters consist of all printable and non-printable characters that are not explicitly designated as metacharacters. This includes all uppercase and lowercase alphabetic characters, all digits, all punctuation marks, and some symbols.

The simplest form of a regular expression is a single, ordinary character that matches itself in a searched string. For example, a single-character pattern, such as A, matches the letter *A* wherever it appears in the searched string. Here are some examples of single-character regular expression patterns:

```
/a/
/7/
/M/
```

You can combine a number of single characters to form a large expression. For example, the following regular expression combines the single-character expressions: a, 7, and M.

```
/a7M/
```

Notice that there is no concatenation operator. You just type one character after another.

See Also

Introduction to Regular Expressions

Special Characters

A number of metacharacters require special treatment when trying to match them. To match these special characters, you must first *escape* the characters, that is, precede them with a backslash character (\). The following table lists special characters and their meanings:

Special Character	**Comment**
$	Matches the position at the end of an input string. If the **RegExp** object's **Multiline** property is set, $ also matches the position preceding \n or \r. To match the $ character itself, use \$.
( )	Marks the beginning and end of a subexpression. Subexpressions may be captured for later use. To match these characters, use \(and \).
*****	Matches the preceding character or subexpression zero or more times. To match the * character, use *.
+	Matches the preceding character or subexpression one or more times. To match the + character, use \+.
.	Matches any single character except the newline character \n. To match ., use \.
[ ]	Marks the beginning of a bracket expression. To match these characters, use \[and \].
?	Matches the preceding character or subexpression zero or one time, or indicates a non-greedy quantifier. To match the ? character, use \?.
\	Marks the next character as a special character, a literal, a backreference, or an octal escape. For example, the character n matches the character n. \n matches a newline character. The sequence \\ matches \ and \(matches (.
/	Denotes the start or end of a literal regular expression. To match the / character, use \/.
^	Matches the position at the beginning of an input string except when used in a bracket expression where it negates the character set. To match the ^ character itself, use \^.
{ }	Marks the beginning of a quantifier expression. To match these characters, use \{ and \}.
|	Indicates a choice between two items. To match |, use \\|.

See Also

Introduction to Regular Expressions

Non-Printable Characters

Non-printing characters may also be part of a regular expression. The following table lists the escape sequences that represent non-printing characters:

Character	Meaning
\c*x*	Matches the control character indicated by *x*. For example, \cM matches a Control-M or carriage return character. The value of *x* must be in the range of A-Z or a-z. If not, c is assumed to be a literal c character.
\f	Matches a form-feed character. Equivalent to \x0c and \cL.
\n	Matches a newline character. Equivalent to \x0a and \cJ.
\r	Matches a carriage return character. Equivalent to \x0d and \cM.
\s	Matches any white space character including space, tab, form-feed, and so on. Equivalent to [\f\n\r\t\v].
\S	Matches any non-white space character. Equivalent to [^ \f\n\r\t\v].
\t	Matches a tab character. Equivalent to \x09 and \cI.
\v	Matches a vertical tab character. Equivalent to \x0b and \cK.

See Also

Introduction to Regular Expressions

Character Matching

The period (.) matches all but one single printing or non-printing character in a string. The exception is a newline character (\n). The following regular expression matches aac, abc, acc, adc, and so on, as well as a1c, a2c, a-c, and a#c:

```
/a.c/
```

To match a string containing a file name in which a period (.) is part of the input string, precede the period in the regular expression with a backslash (\) character. To illustrate, the following regular expression matches filename.ext:

```
/filename\.ext/
```

These expressions only let you match *any* single character. You may want to match specific characters from a list. For example, you might want to find chapter headings that are expressed numerically (Chapter 1, Chapter 2, and so on).

Bracket Expressions

To create a list of matching characters, place one or more individual characters within square brackets ([and]). When characters are enclosed in brackets, the list is called a *bracket expression*. Within brackets, as anywhere else, an ordinary character represents itself, that is, it matches an occurrence of itself in the input text. Most special characters lose their meaning when they occur inside a bracket expression. Here are some exceptions:

- The] character ends a list if it is not the first item. To match the] character in a list, place it first, immediately following the opening [.
- The \ character continues to be the escape character. To match the \ character, use \\.

Characters enclosed in a bracket expression match only a single character for the position in the regular expression. The following regular expression matches Chapter 1, Chapter 2, Chapter 3, Chapter 4, and Chapter 5:

```
/Chapter [12345]/
```

Notice that the word *Chapter* and the space that follows are fixed in position relative to the characters within brackets. The bracket expression is used to specify only the set of characters that matches the single character position immediately following the word *Chapter* and a space. That is the ninth character position.

To express the matching characters using a range instead of the characters themselves, use the hyphen (-) character to separate the beginning and ending characters in the range. The character value of the individual characters determines the relative order within a range. The following regular expression contains a range expression that is equivalent to the bracketed list shown above.

```
/Chapter [1-5]/
```

When a range is specified in this manner, both the starting and ending values are included in the range. It is important to note that the starting value must precede the ending value in Unicode sort order.

To include the hyphen character in a bracket expression, do one of the following:

- Escape it with a backslash:

  ```
  [\-]
  ```

- Put the hyphen character at the beginning or the end of the bracketed list. The following expressions match all lowercase letters and the hyphen:

  ```
  [-a-z]
  [a-z-]
  ```

- Create a range in which the beginning character value is lower than the hyphen character and the ending character value is equal to or greater than the hyphen. Both of the following regular expressions satisfy this requirement:

  ```
  [!--]
  ```

```
[!-~]
```

To find all characters not in the list or range, place the caret (^) character at the beginning of the list. If the caret character appears in any other position within the list, it matches itself. The following regular expression matches chapter headings with numbers greater than 5:

```
/Chapter [^12345]/
```

In the examples above, the expression matches any digit character in the ninth position except 1, 2, 3, 4, or 5. So, for example, Chapter 7 is a match and so is Chapter 9.

The above expressions can be represented using the hyphen character (-):

```
/Chapter [^1-5]/
```

A typical use of a bracket expression is to specify matches of any upper- or lowercase alphabetic characters or any digits. The following expression specifies such a match:

```
/[A-Za-z0-9]/
```

See Also

Introduction to Regular Expressions

Quantifiers

If you cannot specify the number of characters that comprise a match, regular expressions support the concept of quantifiers. These quantifiers let you specify how many times a given component of a regular expression must occur for a match to be true.

The following table illustrates the various quantifiers and their meanings:

Character	Description
*	Matches the preceding character or subexpression zero or more times. For example, zo* matches z and zoo. * is equivalent to {0,}.
+	Matches the preceding character or subexpression one or more times. For example, zo+ matches zo and zoo, but not z. + is equivalent to {1,}.
?	Matches the preceding character or subexpression zero or one time. For example, do(es)? matches the do in do or does. ? is equivalent to {0,1}
{*n*}	*n* is a nonnegative integer. Matches exactly *n* times. For example, o{2} does not match the o in Bob but matches the two o's in food.
{*n*,}	*n* is a nonnegative integer. Matches at least *n* times. For example, o{2,} does not match the o in Bob and matches all the o's in foooood. o{1,} is equivalent to o+. o{0,} is equivalent to o*.
{*n*,*m*}	*m* and *n* are nonnegative integers, where *n* <= *m*. Matches at least *n* and at most *m* times. For example, o{1,3} matches the first three o's in fooooood. o{0,1} is equivalent to o?. Note that you cannot put a space between the comma and the numbers.

Since chapter numbers could easily exceed nine in a large input document, you need a way to handle two or three digit chapter numbers. Quantifiers give you that capability. The following regular expression matches chapter headings with any number of digits:

```
/Chapter [1-9][0-9]*/
```

Notice that the quantifier appears after the range expression. Therefore, it applies to the entire range expression that, in this case, specifies only digits from 0 through 9, inclusive.

The + quantifier is not used here because there does not necessarily need to be a digit in the second or subsequent position. The ? character also is not used because it limits the chapter numbers to only two digits. You want to match at least one digit following Chapter and a space character.

If you know that chapter numbers are limited to only 99 chapters, you can use the following expression to specify at least one but not more than two digits.

```
/Chapter [0-9]{1,2}/
```

The disadvantage of the above expression is that a chapter number greater than 99 will still only match the first two digits. Another disadvantage is that Chapter 0 would match. Better expressions for matching only two digits are the following:

```
/Chapter [1-9][0-9]?/
```

or

```
/Chapter [1-9][0-9]{0,1}/
```

The *, +, and ? quantifiers are all referred to as *greedy* because they match as much text as possible. However, sometimes you just want a minimal match.

For example, you may be searching an HTML document for an occurrence of a chapter title enclosed in an H1 tag. That text appears in your document as:

```
<H1>Chapter 1 – Introduction to Regular Expressions</H1>
```

The following expression matches everything from the opening less than symbol (<) to the greater than symbol (>) that closes the H1 tag.

```
/<.*>/
```

If you only want to match the opening H1 tag, the following, non-greedy expression matches only <H1>.

```
/<.*?>/
```

By placing the ? after a *, +, or ? quantifier, the expression is transformed from a greedy to a non-greedy, or minimal, match.

See Also

Introduction to Regular Expressions

Anchors

Examples in previous topics in this section have only been concerned with finding chapter headings. Any occurrence of the string Chapter followed by a space and a number could be an actual chapter heading, or it could also be a cross-reference to another chapter. Since true chapter headings always appear at the beginning of a line, it may be useful to devise a way to find only the headings and not the cross-references.

Anchors provide that capability. Anchors allow you to fix a regular expression to either the beginning or end of a line. They also allow you to create regular expressions that occur within a word, at the beginning of a word, or at the end of a word. The following table contains the list of regular expression anchors and their meanings:

Character	Description
^	Matches the position at the beginning of the input string. If the **RegExp** object's **Multiline** property is set, ^ also matches the position following \n or \r.
$	Matches the position at the end of the input string. If the **RegExp** object's **Multiline** property is set, $ also matches the position preceding \n or \r.
\b	Matches a word boundary, that is, the position between a word and a space.
\B	Matches a nonword boundary.

You cannot use a quantifier with an anchor. Since you cannot have more than one position immediately before or after a newline or word boundary, expressions such as ^* are not permitted.

To match text at the beginning of a line of text, use the ^ character at the beginning of the regular expression. Do not confuse this use of the ^ with the use within a bracket expression.

To match text at the end of a line of text, use the $ character at the end of the regular expression.

To use anchors when searching for chapter headings, the following regular expression matches a chapter heading that contains no more than two following digits and that occurs at the beginning of a line:

```
/^Chapter [1-9][0-9]{0,1}/
```

Not only does a true chapter heading occur at the beginning of a line, it is also the only text on the line. It occurs at beginning of the line and also at the end of the same line. The following expression ensures that the specified match only matches chapters and not cross-references. It does so by creating a regular expression that matches only at the beginning and end of a line of text.

```
/^Chapter [1-9][0-9]{0,1}$/
```

Matching word boundaries is a little different but adds a very important capability to regular expressions. A word boundary is the position between a word and a space. A nonword boundary is any other position. The following expression matches the first three characters of the word *Chapter* because the characters appear following a word boundary:

```
/\bCha/
```

The position of the \b operator is critical. If it is at the beginning of a string to be matched, it looks for the match at the beginning of the word. If it is at the end of the string, it looks for the match at the end of the word. For example, the following expression matches the string *ter* in the word Chapter because it appears before a word boundary:

```
/ter\b/
```

The following expression matches the string *apt* as it occurs in Chapter but not as it occurs in aptitude:

```
/\Bapt/
```

The string *apt* occurs on a nonword boundary in the word *Chapter* but on a word boundary in the word *aptitude*. For the \B nonword boundary operator, position is not important because the match is not relative to the beginning or end of a word.

See Also

Introduction to Regular Expressions

Alternation and Grouping

Alternation uses the | character to allow a choice between two or more alternatives. For example, you can expand the chapter heading regular expression to return more than just chapter headings. However, it is not as straightforward as you might think. Alternation matches the largest possible expression on either side of the | character. You might think that the following expression matches either Chapter or Section followed by one or two digits occurring at the beginning and ending of a line:

```
/^Chapter|Section [1-9][0-9]{0,1}$/
```

Unfortunately, the above regular expression matches either the word *Chapter* at the beginning of a line, or the word *Section* and whatever numbers follow Section at the end of the line. If the input string is Chapter 22, the above expression only matches the word *Chapter*. If the input string is Section 22, the expression matches Section 22.

To make the regular expressions more responsive, you can use parentheses to limit the scope of the alternation, that is, to make sure that it applies only to the two words *Chapter* and *Section*. However, parentheses are also used to create subexpressions and possibly capture them for later use, something that is covered in the section on backreferences. By adding parentheses in the appropriate places of the above regular expression, you can make the regular expression match either Chapter 1 or Section 3.

The following regular expression uses parentheses to group Chapter and Section so the expression works properly:

```
/^(Chapter|Section) [1-9][0-9]{0,1}$/
```

Although these expressions work properly, the parentheses around Chapter|Section also cause either of the two matching words to be captured for future use. Since there is only one set of parentheses in the above expression, there is only one captured *submatch*. This submatch can be referred to by using the **$1-$9** properties of the **RegExp** object.

In the above example, you merely want to use the parentheses to group a choice between the words *Chapter* and *Section*. To prevent the match from being saved for possible later use, place ?: before the regular expression pattern inside the parentheses. The following modification provides the same capability without saving the submatch:

```
/^(?:Chapter|Section) [1-9][0-9]{0,1}$/
```

In addition to the ?: metacharacters, two other non-capturing metacharacters create something called *lookahead* matches. A positive lookahead, which is specified using ?=, matches the search string at any point where a matching regular expression pattern in parentheses begins. A negative lookahead, which is specified using ?!, matches the search string at any point where a string not matching the regular expression pattern begins.

For example, suppose you have a document that contains references to Windows 3.1, Windows 95, Windows 98, and Windows NT. Suppose further that you need to update the document by changing all references to Windows 95, Windows 98, and Windows NT to Windows 2000. The following regular expression, which is an example of a positive lookahead, matches Windows 95, Windows 98, and Windows NT:

```
/Windows(?=95 |98 |NT )/
```

Once a match is found, the search for the next match begins immediately following the matched text without including the characters in the look-ahead. For example, if the above expression matched Windows 98, the search resumes after Windows not after 98.

See Also

Introduction to Regular Expressions | Backreferences

Backreferences

One of the most important features of regular expressions is the ability to store part of a matched pattern for later reuse. As you may recall, placing parentheses around a regular expression pattern or part of a pattern causes that part of the expression to be stored into a temporary buffer. You can override the capture by using the non-capturing metacharacters ?:, ?=, or ?!.

Each captured submatch is stored as it is encountered from left to right in a regular expressions pattern. The buffer numbers begin at one and continue up to a maximum of 99 captured subexpressions. Each buffer can be accessed using *n* where *n* is one or two decimal digits identifying a specific buffer.

One of the simplest, most useful applications of back references provides the ability to locate the occurrence of two identical, adjacent words in text. Take the following sentence:

```
Is is the cost of of gasoline going up up?
```

The above sentence clearly has several duplicated words. It would be nice to devise a way to fix that sentence without looking for duplicates of every single word. The following regular expression uses a single subexpression to do that:

```
/\b([a-z]+) \1\b/gi
```

The captured expression, as specified by [a-z]+, includes one or more alphabetic characters. The second part of the regular expression is the reference to the previously captured submatch, that is, the second occurrence of the word just matched by the parenthetical expression. \1 specifies the first submatch. The word boundary metacharacters ensure that only whole words are detected. Otherwise, a phrase such as "is issued" or "this is" would be incorrectly identified by this expression.

The global flag (g) following the regular expression indicates that the expression is applied to as many matches as it can find in the input string. The case insensitivity (i) flag at the end of the expression specifies case insensitivity. The multiline flag specifies that potential matches may occur on either side of a newline character.

Using the above regular expression, the following code can use the submatch information to replace an occurrence of two consecutive identical words in a string of text with a single occurrence of the same word:

```
var ss = "Is is the cost of of gasoline going up up?.\n";
var re = /\b([a-z]+) \1\b/gim;       //Create regular expression pattern.
var rv = ss.replace(re,"$1");   //Replace two occurrences with one.
```

The use of the **$1** within the **replace** method refers to the first saved submatch. If you had more than one submatch, you would refer to them consecutively by using **$2**, **$3**, and so on.

Back references can also break down a Universal Resource Indicator (URI) into its component parts. Assume that you want to break down the following URI to the protocol (ftp, http, and so on), the domain address, and the page/path:

```
http://msdn.microsoft.com:80/scripting/default.htm
```

The following regular expressions provide that functionality:

```
/(\w+):\/\/([^/:]+)(:\d*)?([^# ]*)/
```

The first parenthetical subexpression captures the protocol part of the Web address. That subexpression matches any word that precedes a colon and two forward slashes. The second parenthetical subexpression captures the domain address part of the address. That subexpression matches any sequence of characters that does not include / or : characters. The third parenthetical subexpression captures a port number if one is specified. That subexpression matches zero or more digits following a colon. Finally, the fourth parenthetical subexpression captures the path and/or page information specified by the Web address. That subexpression matches one or more characters other than # or the space character.

Applying the regular expression to the above URI, the submatches contain the following:

- **RegExp.$1** contains "http"
- **RegExp.$2** contains "msdn.microsoft.com"
- **RegExp.$3** contains ":80"

- **RegExp.$4** contains "/scripting/default.htm"

See Also

Introduction to Regular Expressions

JScript Reference

The JScript .NET programming language includes an assortment of properties, methods, objects, functions, and so on. In addition, JScript .NET can utilize many corresponding features from the .NET Framework class library. The following sections explain the proper use of these features and the proper syntax within JScript .NET.

In This Section

Feature Information
  Provides version information about the language features in JScript .NET, and compares the features with those specified by the ECMAScript standard.

JScript Language Tour
  Introduces the elements and procedures that developers use to write JScript code and links to specific areas that explain the details behind language elements and code syntax.

JScript .NET Language Reference
  Lists the essential components of the JScript .NET object-oriented programming language and links to topics that explain how to use the language.

JScript Compiler Options
  Lists options available for the command line compiler and links to topics that organize the options either alphabetically or by category.

Related Sections

.NET Framework Reference
  Lists links to topics that explain the syntax and structure of the .NET Framework class library and other essential elements.

Feature Information

The JScript .NET programming language includes an assortment of properties, methods, objects, functions, and so on. In addition, JScript .NET can utilize many corresponding features from the .NET Framework class library. The following sections explain the proper use of these features and the proper syntax within JScript .NET.

In This Section

Microsoft JScript Features - ECMA
  Identifies the language features in JScript .NET that are part of the ECMAScript Language Specification.

Microsoft JScript Features - Non-ECMA
  Identifies the language features in JScript .NET that are not included in the ECMAScript Language Specification.

Version Information
  Provides comparisons between the different versions of JScript.

Related Sections

JScript Reference
  Lists elements that comprise JScript Language Reference and links to topics that explain the details behind the proper use of language elements.

Microsoft JScript Features - ECMA

JScript .NET incorporates almost all of the features of the ECMAScript Edition 3 Language Specification. In addition, JScript .NET is being developed in conjunction with ECMAScript Edition 4 and incorporates many of the proposed features of that language as well. The tables below list the features of ECMAScript 3 and the proposed ECMAScript 4 features that JScript .NET supports.

Category	ECMAScript 3 Feature/Keyword
Array Handling	Array concat, join, length, reverse, slice, sort
Assignments	Assign (=), Addition Assignment (+=), Bitwise AND Assignment (&=), Bitwise OR Assignment (|=), Bitwise XOR Assignment (^=), Division Assignment (/=), Left Shift Assignment (<<=), Modulus Assignment (%=), Multiplication Assignment (*=), Right Shift Assignment (>>=), Subtraction Assignment (-=), Unsigned Right Shift Assignment (>>>=)
Booleans	Boolean, true, false
Comments	/*...*/ or //
Constants/Literals	NaN null, Infinity undefined

Control flow	break
	continue
	do...while
	for
	for...in
	if...else
	Labeled
	return
	switch
	while

Dates and Time	Date getDate, getDay, getFullYear, getHours, getMilliseconds, getMinutes, getMonth, getSeconds, getTime, getTimezoneOffset, getYear, getUTCDate, getUTCDay, getUTCFullYear, getUTCHours, getUTCMilliseconds, getUTCMinutes, getUTCMonth, getUTCSeconds, setDate, setFullYear, setHours, setMilliseconds, setMinutes, setMonth, setSeconds, setTime, setYear, setUTCDate, setUTCFullYear, setUTCHours, setUTCMilliseconds, setUTCMinutes, setUTCMonth, setUTCSeconds, toGMTString, toLocaleString, toUTCString, parse, UTC
Declarations	Function new this var with
Error Handling	Error, description, number, throw, try...catch
Function Creation	caller, Function arguments, length

Global Methods	Global escape, unescape eval isFinite, isNaN parseInt, parseFloat
Math	Math abs, acos, asin, atan, atan2, ceil, cos, exp, floor, log, max, min, pow, random, round, sin, sqrt, tan, E, LN2, LN10, LOG2E, LOG10E, PI, SQRT1_2, SQRT2
Numbers	Number MAX_VALUE, MIN_VALUE NaN NEGATIVE_INFINITY, POSITIVE_INFINITY
Object Creation	Object new constructor, instanceof, prototype, toString, valueOf
Operators	Addition (+), Subtraction (-) Modulus arithmetic (%) Multiplication (*), Division (/) Negation (-) Equality (==), Inequality (!=) Less Than (<), Less Than or Equal To (<=) Greater Than (>) Greater Than or Equal To (>=) Logical And(&&), Or (||), Not (!) Bitwise And (&), Or (|), Not (~), Xor (^) Bitwise Left Shift (<<), Shift Right (>>) Unsigned Shift Right (>>>) Conditional (?:) Comma (,) delete, typeof, void Decrement (--), Increment (++), Strict Equality (===), Strict Inequality (!==)
Objects	Array Boolean Date Function Global Math Number Object RegExp Regular Expression String
Regular Expressions and Pattern Matching	RegExp index, input, lastIndex, $1...$9, source, compile, exec, test Regular Expression Syntax
Strings	String charAt, charCodeAt, fromCharCode indexOf, lastIndexOf split toLowerCase, toUpperCase length concat, slice match, replace, search anchor, big, blink, bold, fixed, fontcolor, fontsize, italics, link, small, strike, sub, sup

Category	**Proposed ECMAScript 4 Feature/Keyword**

Class-Based Objects	class, extends, implements, interface, function get, function set, static, public, private, protected, internal, abstract, final, hide, override, static
Declarations	const
Enumerations	enum

See Also

Microsoft JScript Features - Non-ECMA | JScript Reference

Microsoft JScript Features - Non-ECMA

JScript .NET incorporates almost all of the features in ECMAScript Edition 3 and many of the proposed features proposed for ECMAScript Edition 4. In addition, JScript also has many unique features that are not provided by the ECMAScript languages. These JScript .NET specific features are listed in the table below.

Category	Feature/Keyword
Array Handling	VBArray dimensions, getItem, lbound, toArray, ubound
Class-Based Objects	expando, super
Conditional Compilation	@cc_on, @if Statement, @set Statement, @debug, @position, Conditional Compilation Variables
Data Types	boolean, byte, char, decimal, double, float, int, long, Number, sbyte, short, String, uint, ulong, ushort
Dates and Time	getVarDate
Displaying Information	print
Enumeration	Enumerator atEnd, item, moveFirst, moveNext
Namespaces	package, import
Objects	Enumerator VBArray ActiveXObject GetObject
Script Engine Identification	ScriptEngine ScriptEngineBuildVersion ScriptEngineMajorVersion ScriptEngineMinorVersion

See Also

Microsoft JScript Features - ECMA | JScript Reference

Version Information

JScript is a language that continues to evolve and each new version of the language introduces new features. To take advantage of all the features provided by a particular version of the language, a compatible version of compiler or script engine is required.

When writing code for a server-side application or a command-line program, the version of the compiler and the versions of JScript that it supports are usually known. However, when writing client-side scripts that run in the script engine of a browser, the running script detects the engine version. Once the engine version is known, a script written in a compatible version of JScript can be run. For more information, see Detecting Browser Capabilities.

The following table lists the version of Microsoft JScript implemented by host applications.

Host Application	**1.0**	**2.0**	**3.0**	**4.0**	**5.0**	**5.1**	**5.5**	**5.6**	**.NET**
Microsoft Internet Explorer 3.0	**x**								
Microsoft Internet Information Server 3.0		**x**							
Microsoft Internet Explorer 4.0			**x**						
Microsoft Internet Information Server 4.0			**x**						
Microsoft Internet Explorer 5.0					**x**				
Microsoft Internet Explorer 5.01						**x**			
Microsoft Windows 2000						**x**			
Microsoft Internet Explorer 5.5							**x**		
Microsoft Windows Millennium Edition							**x**		
Microsoft Internet Explorer 6.0								**x**	
Microsoft Windows XP								**x**	
Microsoft Windows Server 2003								**x**	
Microsoft .NET Framework 1.0									**x**

> **Note** The version number reported by the **ScriptEngineMajorVersion** function and the **@_jscript_version** conditional compilation variable is always numeric. This allows numeric comparisons to be made with the version number. For version .NET applications, the version reported is 7.*x*, not .NET. This means that engines that report a version number of 7.*x* or higher can compile JScript .NET code.

The following table lists JScript language features and the version when first introduced.

Language Element	**1.0**	**2.0**	**3.0**	**4.0**	**5.0**	**5.5**	**.NET**
0...n Property						**x**	
$1...$9 Properties			**x**				
abs Method	**x**						
abstract Modifier							**x**
acos Method	**x**						
ActiveXObject Object			**x**				
Addition Operator (+)	**x**						
Addition Assignment Operator (+=)	**x**						
anchor Method	**x**						
apply Method						**x**	
arguments Object	**x**						
arguments Property		**x**					
Array Object		**x**					
asin Method	**x**						
Assignment Operator(=)	**x**						
atan Method	**x**						
atan2 Method	**x**						
atEnd Method			**x**				
big Method	**x**						
Bitwise AND Operator (&)	**x**						
Bitwise AND Assignment Operator (&=)	**x**						
Bitwise Left Shift Operator (<<)	**x**						

Bitwise NOT Operator (~)	x						
Bitwise OR Operator (|)	x						
Bitwise OR Assignment Operator (|=)	x						
Bitwise Right Shift Operator (>>)	x						
Bitwise XOR Operator (^)	x						
Bitwise XOR Assignment Operator (^=)	x						
blink Method	x						
bold Method	x						
boolean Data Type							x
Boolean Object		x					
break Statement	x						
byte Data Type							x
call Method						x	
callee Property						x	
caller Property		x					
catch Statement					x		
@cc_on Statement			x				
ceil Method	x						
char Data Type							x
charAt Method	x						
charCodeAt Method						x	
class Statement							x
Comma Operator (,)	x						
// (Single-line Comment Statement)	x						
/*..*/ (Multiline Comment Statement)	x						
Comparison Operators	x						
compile Method			x				
concat Method (Array)			x				
concat Method (String)			x				
Conditional Compilation			x				
Conditional Compilation Variables			x				
Conditional (ternary) Operator (?:)	x						
const Statement							x
constructor Property		x					
continue Statement	x						
cos Method	x						
Data Type Conversion			x				
Date Object	x						
@debug Directive							x
debugger Statement			x				
decimal Data Type							x
decodeURI Method						x	
decodeURIComponent Method						x	
Decrement Operator (--)	x						
delete Operator			x				
description Property					x		
dimensions Method			x				
Division Operator (/)	x						
Division Assignment Operator (/=)	x						
do...while Statement			x				
double Data Type							x
E Property	x						
encodeURI Method						x	
encodeURIComponent Method						x	

enum Statement							x
Enumerator Object			x				
Equality Operator (==)	x						
Error Object					x		
escape Method	x						
eval Method	x						
exec Method			x				
exp Method	x						
expando Modifier							x
false Literal	x						
final Modifier							x
fixed Method	x						
float Data Type							x
floor Method	x						
fontcolor Method	x						
fontsize Method	x						
for Statement	x						
for...in Statement					x		
fromCharCode Method			x				
function get Statement							x
Function Object		x					
function set Statement							x
function Statement	x						
getDate Method	x						
getDay Method	x						
getFullYear Method			x				
getHours Method	x						
getItem Method			x				
getMilliseconds Method			x				
getMinutes Method	x						
getMonth Method	x						
GetObject Function			x				
getSeconds Method	x						
getTime Method	x						
getTimezoneOffset Method	x						
getUTCDate Method			x				
getUTCDay Method			x				
getUTCFullYear Method			x				
getUTCHours Method			x				
getUTCMilliseconds Method			x				
getUTCMinutes Method			x				
getUTCMonth Method			x				
getUTCSeconds Method			x				
getVarDate Method			x				
getYear Method	x						
Global Object			x				
global Property						x	
Greater than Operator (>)	x						
Greater than or equal to Operator (>=)	x						
hasOwnProperty Method						x	
hide Modifier							x
@if Statement			x				
if...else Statement	x						
ignoreCase Property						x	

import Statement							x
in Operator	x						
Increment Operator (++)	x						
index Property			x				
indexOf Method	x						
Inequality Operator (!=)	x						
Infinity Property			x				
input Property ($_)			x				
instanceof Operator					x		
int Data Type							x
interface Statement							x
internal Modifier							x
isFinite Method			x				
isNaN Method	x						
isPrototypeOf Method						x	
italics Method	x						
item Method			x				
JScript Data Types							x
join Method		x					
Labeled Statement			x				
lastIndex Property			x				
lastIndexOf Method	x						
lastMatch Property($&)						x	
lastParen Property ($+)						x	
lbound Method			x				
leftContext Property ($`)						x	
Left Shift Assignment Operator (<<=)	x						
length Property (arguments)						x	
length Property (Array)		x					
length Property (Function)		x					
length Property (String)	x						
Less than Operator (<)	x						
Less than or equal to Operator (<=)	x						
link Method	x						
LN2 Property	x						
LN10 Property	x						
localeCompare Method						x	
log Method	x						
LOG2E Property	x						
LOG10E Property	x						
Logical AND Operator (&&)	x						
Logical NOT Operator (!)	x						
Logical OR Operator (||)	x						
long Data Type							x
match Method			x				
Math Object	x						
max Method	x						
MAX_VALUE Property		x					
message Property						x	
min Method	x						
MIN_VALUE Property		x					
Modulus Operator (%)	x						
Modulus Assignment Operator (%=)	x						
moveFirst Method			x				

moveNext Method			x				
multiline Property						x	
Multiplication Operator (*)	x						
Multiplication Assignment Operator (*=)	x						
name Property						x	
NaN Property (Global)			x				
NaN Property (Number)		x					
NEGATIVE_INFINITY Property		x					
new Operator	x						
Nonidentity Operator (!==)	x						
null Literal	x						
Number Data Type							x
Number Object		x					
number Property					x		
Object Object			x				
Operator Precedence	x						
override Modifier							x
package Statement							x
parse Method	x						
parseFloat Method	x						
parseInt Method	x						
PI Property	x						
pop Method						x	
@position Directive							x
POSITIVE_INFINITY Property		x					
pow Method	x						
print Statement							x
private Modifier							x
propertyIsEnumerable Property						x	
protected Modifier							x
prototype Property		x					
public Modifier							x
push Method						x	
random Method	x						
RegExp Object			x				
Regular Expression Object			x				
Regular Expression Syntax			x				
replace Method	x						
return Statement	x						
reverse Method		x					
rightContext Property ($')						x	
Right Shift Assignment Operator (>>=)	x						
round Method	x						
sbyte Data Type							x
ScriptEngine Function		x					
ScriptEngineBuildVersion Function		x					
ScriptEngineMajorVersion Function		x					
ScriptEngineMinorVersion Function		x					
search Method			x				
@set Statement			x				
setDate Method	x						
setFullYear Method			x				
setHours Method	x						
setMilliseconds Method			x				

setMinutes Method	x						
setMonth Method	x						
setSeconds Method	x						
setTime Method	x						
setUTCDate Method			x				
setUTCFullYear Method			x				
setUTCHours Method			x				
setUTCMilliseconds Method			x				
setUTCMinutes Method			x				
setUTCMonth Method			x				
setUTCSeconds Method			x				
setYear Method	x						
shift Method						x	
short Data Type							x
sin Method	x						
slice Method (Array)			x				
slice Method (String)			x				
small Method	x						
sort Method		x					
source Property			x				
splice Method						x	
split Method			x				
sqrt Method	x						
SQRT1_2 Property	x						
SQRT2 Property	x						
static Modifier							x
static Statement							x
Strict Equality Operator (===)	x						
strike Method	x						
String Data Type							x
String Object	x						
sub Method	x						
substr Method			x				
substring Method	x						
Subtraction Operator (-)	x						
Subtraction Assignment Operator (-=)	x						
sup Method	x						
super Statement							x
switch Statement			x				
tan Method	x						
test Method			x				
this Statement	x						
throw Statement					x		
toArray Method			x				
toDateString Method						x	
toExponential Method						x	
toFixed Method						x	
toGMTString Method	x						
toLocaleDateString Method						x	
toLocaleLowerCase Method						x	
toLocaleString Method	x						
toLocaleTimeString Method						x	
toLocaleUpperCase Method						x	
toLowerCase Method	x						

toPrecision Method						x	
toString Method		x					
toTimeString Method						x	
toUpperCase Method	x						
toUTCString Method			x				
true Literal	x						
try...catch...finally Statement					x		
Type Annotation							x
Type Conversion							x
typeof Operator	x						
ubound Method			x				
uint Data Type							x
ulong Data Type							x
Unary Negation Operator (-)	x						
undefined Property						x	
unescape Method	x						
unshift Method						x	
Unsigned Right Shift Operator (>>>)	x						
Unsigned Right Shift Assignment Operator (>>>=)	x						
ushort Data Type							x
UTC Method	x						
valueOf Method		x					
var Statement	x						
VBArray Object			x				
void Operator		x					
while Statement	x						
with Statement	x						

See Also

JScript Reference | What's New in JScript .NET

JScript Language Tour

Like many other programming languages, Microsoft JScript scripts or programs are written in text format. Typically, a script or program is comprised of many statements and comments. Within a statement, you can use variables, expressions, and literal data such as strings and numbers.

In This Section

JScript Arrays
: Explains types of arrays and how to use them in JScript.

JScript Assignments and Equality
: Explains how JScript assigns values to variables, array elements, and property elements and explains the equality syntax used by JScript.

JScript Comments
: Illustrates how to use the proper JScript syntax to include comments in code.

JScript Expressions
: Explains how to combine keywords, operators, variables, and literals to yield new values.

JScript Identifiers
: Explains how to create valid names for identifiers in JScript.

JScript Statements
: Provides an overview of the two basic units of instruction in JScript, declaration statements and executable statements.

JScript Data Types
: Includes links to topics that explain how to use primitive data types, reference data types, and .NET Framework data types in JScript.

JScript Variables and Constants
: Lists links to topics that explains how to declare variables and constants and how to use them to refer to objects.

JScript Objects
: Provides a brief overview of objects and lists links that explain how to create and use objects in JScript.

JScript Modifiers
: Explains how to use visibility modifiers, inheritance modifiers, version-safe modifiers, expando modifiers, and static modifiers.

JScript Operators
: Lists the computational, logical, bitwise, assignment, and miscellaneous operators and provides links to information that explains how to use them efficiently.

JScript Functions
: Describes the concept of functions and provides links to topics that explain how to use and create functions.

Coercion in JScript
: Explains how the JScript compiler performs actions on values of different data types.

Copying, Passing, and Comparing Data
: Explains how JScript copies, passes, and compares data when dealing with arrays, functions, objects, and so on.

JScript Conditional Structures
: Describes how JScript normally handles program flow and provides links to information that explains how to regulate the flow of a program's execution.

JScript Reserved Words
: Explains the concept of reserved words and lists the reserved words in JScript.

Security Considerations for JScript
: Explains how to avoid common security problems in JScript .NET code.

Related Sections

JScript .NET Language Reference
: Lists elements that comprise JScript Language Reference and links to topics that explain the details behind the proper use of language elements.

.NET Framework Class Library
: Contains links to topics that explain the classes in the .NET Framework class library and explains how to use the class library documentation.

JScript Arrays

An array groups related pieces of data in one variable. A unique number, called an index or subscript, combined with the shared variable name distinguishes each element in the array. Arrays result in shorter and simpler code in many situations because loops can deal efficiently with any number of elements by using the index number.

JScript provides two different types of arrays, JScript array objects and typed arrays. In JScript array objects, which are sparse, a script can dynamically add and remove elements, and elements can be of any data type. In typed arrays, which are dense, the size is fixed, and elements must be the same type as the base type of the array.

In This Section

Arrays Overview
Describes the two types of JScript arrays, the differences between the types, and how to choose the appropriate array type.

Array Declaration
Explains the concept of declaring arrays and how declaring arrays with the **new** operator differs from declaring array literals.

Array Use
Demonstrates how to access the elements of one-dimensional arrays, multidimensional arrays, and arrays of arrays.

Arrays of Arrays
Explains the concept of arrays of arrays, why arrays of arrays are useful, and how to use them.

Multidimensional Arrays
Explains the concept of multidimensional arrays, how they differ from arrays of arrays, and how to use them.

Related Sections

Array Data
Illustrates how to compose array literal values and how to combine data types in the same array.

Array Object
Reference information that describes the JScript **Array** object, how to use it, and how it interoperates with the **System.Array** data type.

Arrays Overview

An array collects more than one piece of data in one variable. A single index number (for a one-dimensional array) or several index numbers (for an array of arrays or a multidimensional array) references the data in the array. You can refer to an individual element of an array with the array identifier followed with the array index in square brackets (**[]**). To refer to the array as a whole, just use the array identifier. Collecting data in arrays simplifies data management. For example, by using an array, a method can pass a list of names to a function using only one parameter.

There are two types of arrays in JScript, JScript arrays and typed arrays. While the two types of arrays are similar, there are a few differences. JScript arrays and typed arrays can interoperate with each other. Consequently, a JScript **Array** object can call the methods and properties of any typed array, and typed arrays can call many of the methods and properties of the **Array** object. Furthermore, functions that accept typed arrays accept **Array** objects, and vice versa. For more information, see Array Object.

Typed Arrays

Typed arrays (also called native arrays) are similar to arrays used in languages such as C and C++. Typed arrays provide type safety by only storing data that corresponds to the type that the array type declaration specified.

> **Note** You can define a typed array of type **Object** to store data of any type.

When the script creates or initializes the array, it sets the number of elements in a typed array. The only way to change the number of elements is by recreating the array. A typed array created with *n* elements has elements numbered 0 through *n*-1. An attempt to access elements outside that range generates an error. In addition, typed arrays are *dense*, that is, every index in the allowed range refers to an element.

A script can assign a declared, typed array to a variable or constant, or it can pass the array to a function, operator, or statement. When assigning to a variable (or constant), make sure that the data type of the variable matches the type of the array and that the dimensionalities of the arrays match.

A typed array is an instance of the .NET Framework **System.Array** object. To access static members of the **System.Array** object or to explicitly create a **System.Array** object requires the fully qualified name **System.Array**. This syntax distinguishes it from **Array**, the intrinsic JScript **Array** object.

JScript Arrays

A JScript **Array** object, which provides more flexibility than a typed array, is convenient when you want a generic stack, when you want a list of items, and when performance is not a top concern. However, since typed arrays provide type safety, performance improvements, and better interaction with other languages, developers typically choose typed arrays instead of JScript arrays.

A JScript array can store data of any type, which makes it easy to quickly write scripts that use arrays without considering type conflicts. Since this bypasses the strong type checking that JScript .NET provides, use this feature carefully.

Scripts can dynamically add elements to or remove elements from JScript arrays. To add an array element, assign a value to the element. The **delete** operator can remove elements.

A JScript array is *sparse*. That is, if an array has three elements that are numbered 0, 1, and 2, element 50 can exist without the presence of elements 3 through 49. Each JScript array has a **length** property that is automatically updated when an element is added. In the previous example, the addition of element 50 causes the value of the length variable to change to 51 rather than to 4.

A JScript **Array** object and a JScript **Object** are almost identical. The two main differences are that an **Object** (by default) does not have an automatic length property, and a JScript **Object** does not have the properties and methods of an **Array**. For more information, see JScript Array Object.

See Also

JScript Arrays | Array Data | Array Object | JScript Array Object

Array Declaration

As with all other data in JScript, variables can store arrays. Type annotation can specify that a variable must contain an array object or a typed array, but it does not provide an initial array. To store an array in a variable, you must *declare* an array and assign it to that variable.

Declaring a JScript array object creates a new **Array** object, while declaring a typed array reserves a section of memory large enough to store every element of the array. Both types of arrays can be declared either by using the **new** operator to explicitly construct a new array, or by using an array literal.

Array Declaration with the new Operator

To declare a new JScript **Array** object, you can use the **new** operator with the **Array** constructor. Since you can dynamically add members to a JScript array, you do not need to specify an initial size for the array. In this example, `a1` is assigned a zero length array.

```
var a1 = new Array();
```

To assign an initial length to the array created with the **Array** constructor, pass an integer to the array constructor. The array length must be zero or positive. The following code assigns an array of length 10 to `a2`.

```
var a2 = new Array(10);
```

If more than one parameter or a single non-numeric parameter is passed to the **Array** constructor, the resulting array contains all the parameters as the array elements. For example, the following code creates an array in which element 0 is the number 10, element 1 is the string "Hello", and element 2 is the current date.

```
var a3 = new Array(10, "Hello", Date());
```

The **new** operator can also declare typed arrays. Since typed arrays cannot accept dynamically added elements, the declaration must specify the size of the array. The constructor for typed arrays uses square brackets instead of parentheses around the array size. For example, the following code declares an array of five integers.

```
var i1 : int[] = new int[5];
```

The **new** operator can also declare multidimensional arrays. The following example declares a three-by-four-by-five array of integers.

```
var i2 : int[,,] = new int[3,4,5];
```

When declaring an array of arrays, the base array must be declared before declaring the subarrays; they cannot all be declared at the same time. This provides flexibility in determining the sizes of the subarrays. In this example, the first subarray has length 1, the second has length 2, and so on.

```
// First, declare a typed array of type int[], and initialize it.
var i3 : int[][] = new (int[])[4];
// Second, initialize the subarrays.
for(var i=0; i<4; i++)
   i3[i] = new int[i+1];
```

Array Declaration With Array Literals

An alternative way to simultaneously declare and initialize arrays is to use array literals. An array literal represents a JScript **Array**. Since JScript arrays interoperate with typed arrays, however, literals can also be used to initialize typed arrays. For more information, see Array Data.

Array literals can easily initialize one-dimensional arrays. Note that the compiler will attempt to convert the data from the array literal to the correct type when assigned to a typed array. In this example, a literal array is assigned to a JScript array and a typed array.

```
var al1 : Array = [1,2,"3"];
var il1 : int[] = [1,2,"3"];
```

Array literals can also initialize arrays of arrays. In the following example, an array of two arrays of integers initializes both the JScript array and the typed array.

```
var al1 : Array = [[1,2,3],[4,5,6]];
var il1 : int[][] = [[1,2,3],[4,5,6]];
```

Array literals cannot initialize multidimensional typed arrays.

See Also

JScript Arrays | Array Data | JScript Array Object | new Operator

Array Use

There are several types of arrays that can be used in JScript .NET. The following information explains how to use some of these arrays and how to choose an appropriate array for your particular application.

One-Dimensional Arrays

The following example shows how to access the first and last elements of the `addressBook` array. This assumes that another part of the script defines and assigns a value to `addressBook`. Since arrays are zero-indexed in JScript, the first element of an array is zero and the last element is the length of the array minus one.

```
var firstAddress = addressBook[0];
var lastAddress = addressBook[addressBook.length-1];
```

Arrays of Arrays vs. Multidimensional Arrays

You can store data that is referenced by several indices in either arrays of arrays or multidimensional arrays. Each type of array has unique features.

Arrays of arrays are useful for applications in which each subarray has a different length. The subarrays can easily be reorganized, which helps with sorting of array elements. An example of a typical use is a calendar, where a `Year` array stores twelve `Month` arrays, and each `Month` array stores data for the appropriate number of days.

Multidimensional arrays are useful for applications where the size of the array in each dimension is known at the time of declaration. Multidimensional arrays are more efficient than arrays of arrays in terms of speed and memory consumption. Multidimensional arrays must be typed arrays. An example of a typical use is a matrix used for mathematical calculations, where the array size is fixed and known from the start.

Looping over JScript Array Elements

Since JScript arrays are sparse, an array may have a large numbers of undefined elements between the first element and the last element. This means that if you use a **for** loop to access the array elements, you must check if each element is **undefined**.

Fortunately, JScript provides a **for...in** loop that allows you to easily access only the defined elements of a JScript array. The following example defines a sparse JScript array and displays its elements using both the **for** loop and the **for...in** loop.

```
var a : Array = new Array;
a[5] = "first element";
a[100] = "middle element";
a[100000] = "last element";
print("Using a for loop. This is very inefficient.")
for(var i = 0; i<a.length; i++)
   if(a[i]!=undefined)
      print("a[" + i + "] = " + a[i]);
print("Using a for...in loop. This is much more efficient.");
for(var i in  a)
   print("a[" + i + "] = " + a[i]);
```

The output of this program is:

```
Using a for loop. This is very inefficient.
a[5] = first element
a[100] = middle element
a[100000] = last element
Using a for...in loop. This is much more efficient.
a[5] = first element
a[100] = middle element
a[100000] = last element
```

See Also

JScript Arrays | Arrays of Arrays | Multidimensional Arrays | for...in Statement

Arrays of Arrays

It is possible to create an array and populate it with other arrays. The base array can be either a JScript array or a typed array. JScript arrays allow more flexibility in the types of stored data, while typed arrays prevent storage of inappropriately typed data in the array.

Arrays of arrays are useful for applications in which each subarray has a different length. If each subarray has the same length, a multidimensional array may be more useful. For more information, see Multidimensional Arrays.

Typed Arrays of Arrays

In the following example, an array of arrays of strings stores pet names. Since the number of elements in each subarray is independent of the others (the number of cat names can be different from the number of dog names), an array of arrays is used instead of a multidimensional array.

```
// Create two arrays, one for cats and one for dogs.
// The first element of each array identifies the species of pet.
var cats : String[] = ["Cat","Beansprout", "Pumpkin", "Max"];
var dogs : String[] = ["Dog","Oly","Sib"];

// Create a typed array of String arrays, and initialze it.
var pets : String[][] = [cats, dogs];

// Loop over all the types of pets.
for(var i=0; i<pets.length; i++)
   // Loop over the pet names, but skip the first element of the list.
   // The first element identifies the species.
   for(var j=1; j<pets[i].length; j++)
      print(pets[i][0]+": "+pets[i][j]);
```

The output of this program is:

```
Cat: Beansprout
Cat: Pumpkin
Cat: Max
Dog: Oly
Dog: Sib
```

You can also use a typed array of type **Object** to store arrays.

JScript Array of Arrays

Using a JScript array as the base array provides flexibility in the types of stored subarrays. For example, the following code creates a JScript array that stores JScript arrays containing strings and integers.

```
// Declare and initialize the JScript array of arrays.
var timecard : Array;
timecard = [ ["Monday", 8],
             ["Tuesday", 8],
             ["Wednesday", 7],
             ["Thursday", 9],
             ["Friday", 8] ];
// Display the contents of timecard.
for(var i=0; i<timecard.length; i++)
   print("Worked " + timecard[i][1] + " hours on " + timecard[i][0] + ".");
```

The preceding example displays:

```
Worked 8 hours on Monday.
Worked 8 hours on Tuesday.
Worked 7 hours on Wednesday.
Worked 9 hours on Thursday.
Worked 8 hours on Friday.
```

See Also

JScript Arrays | Array Data | Multidimensional Arrays

Multidimensional Arrays

You can create multidimensional typed arrays in JScript. A multidimensional array uses more than one index to access data. When the script declares the array, it sets the range for each index. Multidimensional arrays are similar to arrays of arrays, where each subarray can have a different length. For more information, see Arrays of Arrays.

The data type name followed by a pair of square brackets (**[]**) specifies a one-dimension array data type. The same procedure specifies a multidimensional array data type, but commas (**,**) are between the brackets. The dimensionality of the array is equal to the number of commas plus one. The following example illustrates the difference between defined a one-dimensional array and a multidimensional array.

```
// Define a one-dimensional array of integers. No commas are used.
var oneDim : int[];
// Define a three-dimensional array of integers.
// Two commas are used to produce a three dimensional array.
var threeDim : int[,,];
```

In the following example, a two-dimensional array of characters is used to store the state of a tic-tac-toe board.

```
// Declare a variable to store two-dimensional game board.
var gameboard : char[,];
// Create a three-by-three array.
gameboard = new char[3,3];
// Initialize the board.
for(var i=0; i<3; i++)
   for(var j=0; j<3; j++)
      gameboard[i,j] = " ";
// Simulate a game. 'X' goes first.
gameboard[1,1] = "X"; // center
gameboard[0,0] = "O"; // upper-left
gameboard[1,0] = "X"; // center-left
gameboard[2,2] = "O"; // lower-right
gameboard[1,2] = "X"; // center-right, 'X" wins!
// Display the board.
var str : String;
for(var i=0; i<3; i++) {
   str = "";
   for(var j=0; j<3; j++) {
      if(j!=0) str += "|";
      str += gameboard[i,j];
   }
   if(i!=0)
      print("-+-+-");
   print(str);
}
```

The output of this program is:

```
O| |
-+-+-
X|X|X
-+-+-
 | |O
```

You can use a multidimensional typed array of type **Object** to store data of any type.

See Also

JScript Arrays | Array Data | Arrays of Arrays

JScript Assignments and Equality

In JScript, the assignment operator assigns a value to a variable. An equality operator compares two values.

Assignments

Like many programming languages, JScript uses the equal sign (=) to assign values to variables: it is the assignment operator. The left operand of the = operator must be an *Lvalue*, which means that it must be a variable, array element, or object property.

The right operand of the = operator must be an *Rvalue*. *Rvalues* can be an arbitrary value of any type, including a value that is the result of an expression. Following is an example of a JScript assignment statement.

```
anInteger = 3;
```

JScript interprets this statement as meaning:

"Assign the value 3 to the variable `anInteger`,"

or

"`anInteger` takes the value 3."

An assignment will always be successful if type annotation has not bound the variable in the statement to a particular data type. Otherwise, the compiler will attempt to convert the *Lvalue* to the data type of the *Rvalue*. If the conversion can never be successfully performed, the compiler generates an error. If the conversion will succeed for some values but fail for others, the compiler generates a warning that the conversion may fail when the code is run.

In this example, the integer value stored in the variable `i` is converted to a double value when assigned to the variable `x`.

```
var i : int = 29;
var x : double = i;
```

For more information, see Type Conversion.

Equality

Unlike some other programming languages, JScript does not use the equal sign as a comparison operator. The equal sign (=) is used only as the assignment operator. For comparison between two values, you can use either the equality operator (==) or the strict equality operator (===).

The equality operator compares primitive strings, numbers, and Booleans by value. If two variables have the same value, after type conversion if necessary, the equality operator returns **true**. Objects (including **Array**, **Function**, **String**, **Number**, **Boolean**, **Error, Date** and **RegExp** objects) compare by reference. Even if two object variables have the same value, the comparison returns **true** only if they refer to exactly the same object.

The strict equality operator compares both the value and type of two expressions; **true** is returned only if the two expressions compare as equal with the equality operator and the data type is the same for both operands.

> **Note** The strict equality operator does not distinguish between the different numeric data types. Be certain you understand the difference between the assignment operator, the equality operator, and the strict equality operator.

Comparisons in your scripts always have a Boolean outcome. Consider the following line of JScript code.

```
y = (x == 2000);
```

Here, the value of the variable `x` is tested to see if it is equal to the number 2000. If it is, the result of the comparison is the Boolean value **true**, which is assigned to the variable `y`**.** If `x` is not equal to 2000, then the result of the comparison is the Boolean value **false**, assigned to `y`.

The equality operator will type convert to check if the values are the same. In the following line of JScript code, the literal string "42" will be converted to a number before comparing it to the number 42. The result is **true**.

```
42 == "42";
```

Objects are compared using different rules. The behavior of the equality operator depends on the type of the objects. If the objects are instances of a class that is defined with an equity operator, the returned value depends on the implementation of the equity operator. Classes which provide an equity operator cannot be defined in JScript .NET, although other .NET Framework languages allow for such class definitions.

Objects without a defined equity operator, such as an object based on the JScript **Object** object or an instance of a JScript .NET class, compare as equal only if both objects refer to the same object. This means that two distinct objects that contain the same data compare as different. The following example illustrates this behavior.

```
// A primitive string.
var string1 = "Hello";
// Two distinct String objects with the same value.
var StringObject1 = new String(string1);
var StringObject2 = new String(string1);

// An object converts to a primitive when
// comparing an object and a primitive.
print(string1 == StringObject1);   // Prints true.

// Two distinct objects compare as different.
print(StringObject1 == StringObject2);   // Prints false.

// Use the toString() or valueOf() methods to compare object values.
print(StringObject1.valueOf() == StringObject2);   // Prints true.
```

Equality operators are especially useful in the condition statements of control structures. Here, you combine an equality operator with a statement that uses it. Consider the following JScript code sample.

```
if (x == 2000)
   z = z + 1;
else
   x = x + 1;
```

The **if...else** statement in JScript performs one action (in this case, `z = z + 1`) if the value of `x` is 2000, and an alternate action (`x = x + 1`) if the value of `x` is not 2000. For more information, see JScript Conditional Structures.

The strict equality operator (===) only performs type conversion on numeric data types. This means that the integer 42 is considered to be identical to the double 42, but both are not identical to the string "42". This behavior is demonstrated by this JScript code.

```
var a : int = 42;
var b : double = 42.00;
var c : String = "42";
print(a===b); // Displays "true".
print(a===c); // Displays "false".
print(b===c); // Displays "false".
```

See Also

JScript Language Tour | Boolean Data | JScript Conditional Structures | Type Conversion

JScript Comments

A single-line JScript comment begins with a pair of forward slashes (//). Here is an example of a single-line comment, followed by a line of code.

```
// This is a single-line comment.
aGoodIdea = "Comment your code for clarity.";
```

A multiline JScript comment begins with a forward slash and asterisk (/*), and ends with the reverse (*/).

```
/*
This is a multiline comment that explains the preceding code statement.
The statement assigns a value to the aGoodIdea variable. The value,
which is contained between the quote marks, is called a literal. A
literal explicitly and directly contains information; it does not
refer to the information indirectly. The quote marks are not part
of the literal.
*/
```

If you attempt to embed one multiline comment within another, JScript interprets the resulting multiline comment in an unexpected way. The */ that marks the end of the embedded multiline comment is interpreted as the end of the entire multiline comment. Consequently, the text following the embedded multiline comment is interpreted as JScript code and may generate syntax errors.

In the following example, the third line of text is interpreted as JScript code because JScript has interpreted the innermost */ to be the end of the outermost comment:

```
/* This is the outer-most comment
/* And this is the inner-most comment */
...Unfortunately, JScript will try to treat all of this as code. */
```

It is recommended that you write all comments as blocks of single-line comments. This allows you to comment out large segments of code with a multiline comment later.

```
// This is another multiline comment, written as a series of single-line comments.
// After the statement is executed, you can refer to the content of the aGoodIdea
// variable by using its name, as in the next statement, in which a string literal is
// appended to the aGoodIdea variable by concatenation to create a new variable.
var extendedIdea = aGoodIdea + " You never know when you'll have to figure out what it does."
;
```

Alternatively, you can use conditional compilation to safely and effectively comment out large segments of code.

See Also

JScript Reference | JScript Language Tour | Conditional Compilation

JScript Expressions

A JScript expression is a combination of keywords, operators, variables, and literals that yield a value. An expression can perform a calculation, manipulate data, call a function, test data, or perform other operations.

The simplest expressions are literals. Here are some examples of JScript literal expressions. For more information, see Data in JScript.

```
3.9                                // numeric literal
"Hello!"                           // string literal
false                              // Boolean literal
null                               // literal null value
[1,2,3]                            // Array literal
var o = {x:1, y:2}                 // Object literal
var f = function(x){return x*x;}  // function literal
```

Expressions that are more complicated can contain variables, function calls, and other expressions. You can use operators to combine expressions and create complex expressions. Examples of using operators are:

```
4 + 5         // additon
x += 1        // addition assignment
10 / 2        // division
a & b         // bitwise AND
```

Here are some examples of JScript complex expressions.

```
radius = 10;
anExpression = 3 * (4 / 5) + 6;
aSecondExpression = Math.PI * radius * radius;
aThirdExpression = "Area is " + aSecondExpression + ".";
myArray = new Array("hello", Math.PI, 42);
myPi = myArray[1];
```

See Also

JScript Reference | JScript Language Tour | JScript Operators | JScript Variables and Constants

JScript Identifiers

In JScript, identifiers are used to:

- name variables, constants, functions, classes, interfaces, and enumerations
- provide labels for loops (not used very often).

JScript is a case-sensitive language. Consequently, a variable named `myCounter` is different from a variable named `MyCounter`. Variable names can be of any length. The rules for creating valid variable names are as follows:

- The first character must be a Unicode letter (either uppercase or lowercase) or an underscore (_) character. Note that a number cannot be used as the first character.
- Subsequent characters must be letters, numbers, or underscores.
- The variable name must not be a reserved word.

Here are some examples of valid identifiers:

```
_pagecount
Part9
Number_Items
```

Here are some examples of *invalid* identifiers:

```
99Balloons // Cannot begin with a number.
Smith&Wesson // The ampersand (&) character is not a valid character for variable names.
```

When choosing identifiers, avoid JScript reserved words and words that are already the names of intrinsic JScript objects or functions, such as **String** or **parseInt**.

See Also

JScript Variables and Constants | JScript Functions | JScript Objects | JScript Reserved Words

JScript Statements

A JScript program is a collection of statements. A JScript statement, which is equivalent to a complete sentence in a natural language, combines expressions that perform one complete task.

A statement consists of one or more expressions, keywords, or operators (symbols). Typically, a statement is contained within a single line, although two or more statements can appear on the same line if they are separated with semicolons. In addition, most statements can span multiple lines. The exceptions are:

- The postfix increment and decrement operators must appear on the same line as their argument. For example, `x++` and `i--`.
- The **continue** and **break** keywords must appear on the same line as their label. For example, `continue label1` and `break label2`.
- The **return** and **throw** keywords must appear on the same line as their expression. For example, `return (x+y)`, and `throw "Error 42"`.
- A custom attribute must appear on the same line as the declaration it is modifying, unless it is preceded by a modifier. For example, `myattribute class myClass`.

Although explicit termination of statements at the end of a line is not required, most of the JScript .NET examples provided here are explicitly terminated for clarity. This is done with the semicolon (;), which is the JScript statement termination character. Here are two examples of JScript statements.

```
var aBird = "Robin"; // Assign the text "Robin" to the variable aBird.
var today = new Date(); // Assign today's date to the variable today.
```

A group of JScript statements surrounded by braces ({}) is called a block. Statements within a block can generally be treated as a single statement. This means you can use blocks in most places that JScript expects a lone statement. Notable exceptions include the headers of **for** and **while** loops. The following example illustrates a typical **for** loop:

```
var i : int = 0;
var x : double = 2;
var a = new Array(4);
for (i = 0; i < 4; i++) {
   x *= x;
   a[i] = x;
}
```

Notice that the individual statements within the block end with semicolons, but the block itself does not.

Generally, functions, conditionals, and classes use blocks. Notice that unlike C++ and most other languages, JScript does not consider a block to be a new scope; only functions, classes, static initializers, and catch blocks create a new scope.

In the following example, the first statement begins the definition of a function that consists of an **if...else** sequence of three statements. Following that block is a statement that is not enclosed in the braces of the function block. Therefore, the last statement is not part of the function definition.

```
function FeetToMiles(feet, cnvType) {
   if (cnvType == "NM")
      return( (feet / 6080) + " nautical miles");
   else if (cnvType == "M")
      return( (feet / 5280) + " statute miles");
   else
      return ("Invalid unit of measure");
}
var mradius = FeetToMiles(52800, "M");
```

See Also

JScript Reference | JScript Language Tour | class Statement | function Statement | if...else Statement | static Statement

JScript Data Types

JScript provides 13 primitive data types and 13 reference data types. In addition to these, you can declare new data types or use any of the Common Language Specification (CLS)-compliant .NET Framework data types. This section includes information about the intrinsic data types, how to extend those types, how to define your own data types, how to enter data, and how to convert data from one type to another.

In This Section

Data in JScript
  Links to topics that explain how to enter array, Boolean, numeric, string, and object data.

Data Type Summary
  Includes a tables of the primitive and reference data types in JScript and corresponding classes in the .NET Framework.

User Defined Data Types
  Explains how to use the class statement to define new data types.

Typed Arrays
  Illustrates how to define, initialize, and use typed arrays.

Type Conversion
  Explains the concept of type conversion and the distinction between implicit and explicit conversion.

Related Sections

JScript Objects
  Provides a brief overview of objects and lists links that explain how to create and use objects in JScript.

Data Types
  Lists links to reference topics that explain the intrinsic data types and their associated properties and methods.

Data in JScript

As with most languages, JScript uses several fundamental kinds of data. Among these are numeric data and string data. A string is a block of text. There are several ways to enter this data in a JScript program. Data is often entered with literal expressions.

In This Section

Array Data
Explains the concept of arrays in JScript and how to enter array data using array literals in a script.

Boolean Data
Explains the concept of Boolean data and how to use either of its two literal values in JScript code.

Numeric Data
Describes the difference between integral and floating-point data and how to enter numeric data in a script.

String Data
Explains the concept of string data, its syntax, and the use of escape characters.

Object Data
Describes the concept of object data in Jscript, its initialization and use.

Related Sections

JScript Data Types
Includes links to topics that explain how to use primitive data types, reference data types, and .NET Framework data types in JScript.

Data Types
Lists links to reference topics that explain the intrinsic data types and their associated properties and methods.

Array Data

An array literal can initialize an array in JScript. An array literal, which represents a JScript **Array** object, is represented by a comma-delimited list that is surrounded by a pair of square brackets ([]). Each element of the list can be either a valid JScript expression or empty (two consecutive commas). The index number of the first element in the array literal list is zero; each subsequent element in the list corresponds to a subsequent element in the array. A JScript **Array** is sparse; if an element of the array literal list is empty, the corresponding element in the JScript **Array** is not initialized.

In this example, the variable `arr` is initialized to be an array with three elements.

```
var arr = [1,2,3];
```

You can use empty elements in the **Array** literal list to create a sparse array. For example, the following Array literal represents an array that defines only elements 0 and 4.

```
var arr = [1,,,,5];
```

An array literal can include data of any type, including other arrays. In the following array of arrays, the second subarray has both string and numeric data.

```
var cats = [ ["Names", "Beansprout", "Pumpkin", "Max"], ["Ages", 6, 5, 4] ];
```

Since JScript **Array** objects interoperate with typed arrays, array literals can initialize typed arrays as well with a few restrictions. The data in the array literal must be convertible to the data type of the typed array. An array literal cannot initialize a multidimensional typed array, but an array literal can initialize a typed array of typed arrays. A two-step process occurs when an array literal initializes a typed array. First, the array literal is converted to a typed array, which is used to initialize the typed array. As a part of the conversion, each empty element of the array literal is first interpreted as **undefined**, and then every element of the literal is converted to the appropriate data type for the typed array. In the following example, the same array literal is used to initialize a JScript array, an integer array, and a double array.

```
var arr = [1,,3];
var arrI : int[] = [1,,3];
var arrD : double[] = [1,,3];
print(arr);   // Displays  1,,3.
print(arrI);  // Displays  1,0,3.
print(arrD);  // Displays  1,NaN,3.
```

The empty element of the array literal is represented as 0 in the integer array and **NaN** in the double array, since **undefined** maps to those values.

See Also

Data in JScript | JScript Expressions | Data Types | JScript Arrays | Intrinsic Objects | Array Object | Type Conversion

Boolean Data

Whereas numeric and string data types can have a virtually unlimited number of different values, the **boolean** data type can have only two. They are the literals **true** and **false**. A Boolean value expresses the validity of a condition (tells whether the condition is true or false).

You can use a literal Boolean value (**true** or **false**) as the condition statement in a control structure. For example, you can create a potentially infinite loop using **true** as the condition for the **while** statement.

```
var s1 : String = "Sam W.";
var s2 : String = "";
while (true) {
   if(s2.Length<s1.Length)
      s2 = s2 + "*";
   else
      break;
}
print(s1);   // Prints Sam W.
print(s2);   // Prints ******
```

Note that the condition for breaking out of an infinite loop can be moved to the loop control, making it an explicitly finite loop. However, some loops can be written much more simply using the infinite loop construction.

Using a Boolean literal in an **if...else** statement allows you to easily include a statement or choose between statements in your program. This technique is useful for developing programs. However, it is more efficient to include a statement directly (without an **if** statement) or use comments to prevent inclusion of a statement.

For more information, see JScript Conditional Structures.

See Also

Data in JScript | JScript Expressions | JScript Conditional Structures | true Literal | false Literal | boolean Data Type | Boolean Object

Numeric Data

The choice between the two types of numeric data in JScript, integral data and floating-point data, depend on the particular circumstances in which they are used. There are also different ways of representing integral data and floating-point data literally.

Positive whole numbers, negative whole numbers, and the number zero are integers. They can be represented in base 10 (decimal), base 8 (octal), and base 16 (hexadecimal). Most numbers in JScript are written in decimal. You denote octal integers by prefixing them with a leading 0 (zero). They can contain digits 0 through 7 only. A number with a leading 0, containing the digits 8 and/or 9 is interpreted as a decimal number. Use of octal numbers is not generally recommended.

You denote hexadecimal (hex) integers by prefixing them with a leading "0x" (zero and x|X). They can contain digits 0 through 9, and letters A through F (either uppercase or lowercase) only. The letters A through F represent, as single digits, 10 through 15 in base 10. That is, 0xF is equivalent to 15, and 0x10 is equivalent to 16.

Both octal and hexadecimal numbers, which can be negative, cannot have a decimal portion and cannot be written in scientific (exponential) notation.

Floating-point values are whole numbers with a decimal portion. Like integers, they can be represented literally with digits followed by a decimal point and more digits. Additionally, they can be expressed in scientific notation. That is, an uppercase or lowercase letter *e* is used to represent "times ten to the power of". A number that begins with a single 0 and contains a decimal point is interpreted as a decimal floating-point literal and not an octal literal.

Additionally, floating-point numbers in JScript can represent special numerical values that integral data types cannot. These are:

- **NaN** (not a number). This is used when a mathematical operation is performed on inappropriate data, such as strings or the undefined value.
- **Infinity**. This is used when a positive number is too large to represent in JScript.
- **-Infinity** (negative Infinity) This is used when the magnitude of a negative number is too large to represent in JScript.
- Positive and Negative 0. JScript differentiates between positive and negative zero in some situations.

Here are some examples of JScript numbers. Notice that a number that begins with "0x" and contains a decimal point will generate an error.

Number	Description	Decimal Equivalent
.0001, 0.0001, 1e-4, 1.0e-4	Four equivalent floating-point numbers.	0.0001
3.45e2	A floating-point number.	345
42	An integer.	42
0378	An integer. Although this looks like an octal number (it begins with a zero), 8 is not a valid octal digit, so the number is treated as a decimal. This produces a level 1 warning.	378
0377	An octal integer. Notice that although it only appears to be one less than the number above, its actual value is quite different.	255
0.0001, 00.0001	A floating point number. Even though this begins with a zero, it is not an octal number because it has a decimal point.	0.0001
0Xff	A hexadecimal integer.	255
0x37CF	A hexadecimal integer.	14287
0x3e7	A hexadecimal integer. Notice that the letter *e* is not treated as exponentiation.	999
0x3.45e2	This is an error. Hexadecimal numbers cannot have decimal parts.	N/A (compiler error)

Variables of any integral data type can represent only a finite range of numbers. If you attempt to assign a numeric literal that is too large or too small to an integral data type, a type-mismatch error will be generated at compile time. For more information, see Data Type Summary.

Data Types of Literals

In most situations, the data type that JScript interprets numeric literals as is inconsequential. However, when the numbers are very large or very precise, these details are important.

Integer literals in JScript can represent data of type **int**, **long**, **ulong**, **decimal**, or **double**, depending on the size of the literal and its use. Literals in the range of the **int** type (-2147483648 to 2147483647) are interpreted as type **int**. Literals outside of that range but within the range of the **long** type (-9223372036854775808 to 9223372036854775807) are interpreted as **long**. Literals outside of that range but within the range of the **ulong** type (9223372036854775807 to 18446744073709551615) are interpreted as **ulong**. All others integer literals are interpreted as type **double**, which entails a loss of precision. An exception to the last rule is that the literal will be interpreted as a **decimal** if the literal is immediately stored in a variable or constant typed as **decimal**, or if it is passed to a function that is typed to receive a **decimal**.

A JScript floating-point literal is interpreted as the data type **double**, unless the literal is immediately used as **decimal** (like integer literals), in which case the literal is interpreted as a **decimal**. The **decimal** data type cannot represent **NaN**, positive **Infinity**, or negative **Infinity**.

See Also

Data in JScript | JScript Expressions | Data Types | NaN Property (Global) | Infinity Property

Object Data

An object literal can initialize a JScript **Object** object. An object literal is represented by a comma-delimited list that is surrounded with a pair of curly braces ({}). Each element of the list is a property followed by a colon and the value of the property. The value can be any valid JScript expression.

In this example, the variable `obj` is initialized to be an object with two properties, `x` and `y`, with the values 1 and 2 respectively.

```
var obj = { x:1, y:2 };
```

Object literals can be nested. In this example, an identifier `cylinder` refers to an object with three properties, `height`, `radius`, and `sectionAreas`. The `sectionAreas` property is an object with its own properties, `top`, `bottom`, and `side`.

```
var r = 3;
var h = 2;
var cylinder = { height : h, radius : r,
                 sectionAreas : { top : 4*Math.PI*r*r,
                                  bottom : 4*Math.PI*r*r,
                                  side : 2*Math.PI*r*h } };
```

> **Note** An object literal cannot be used to initialize an instance of a class-based object. The appropriate constructor function must be used to perform the initialization. For more information, see Class-based Objects.

See Also

Data in JScript | JScript Expressions | Intrinsic Objects | Object Object

String Data

A string value is a chain of zero or more concatenated, Unicode characters (letters, digits, and punctuation marks). The string data type represents text in JScript. To include string literals in your scripts, enclose them in matching pairs of single or double quotation marks. Double quotation marks can be contained within strings surrounded by single quotation marks, and single quotation marks can be contained within strings surrounded by double quotation marks. The following are examples of strings:

```
"The earth is round."
'"Come here Watson. I need you." said Alexander.'
"42"
"15th"
'c'
```

JScript provides escape sequences that you can include in strings to create characters that you cannot type directly. Each of these sequences begins with a backslash. The backslash is an escape character that informs the JScript interpreter that the next character is special.

Escape sequence	Meaning
\b	Backspace
\f	Form feed (rarely used)
\n	Line feed (newline)
\r	Carriage return. Use with the line feed (\r\n) to format output.
\t	Horizontal tab
\v	Vertical tab (rarely used)
\'	Single quote (')
\"	Double quote (")
\\	Backslash (\)
n	ASCII character represented by the octal number *n*. The value of *n* must be in the range 0 to 377 (octal).
\x*hh*	ASCII character represented by the two-digit hexadecimal number *hh*.
\u*hhhh*	Unicode character represented by the four-digit hexadecimal number *hhhh*.

Any escape sequence not included in this table simply codes for the character that follows the backslash in the escape sequence. For example, "\a" is interpreted as "a".

Since the backslash itself represents the start of an escape sequence, you cannot directly type one in your script. If you want to include a backslash, you must type two sequential characters (\\).

```
'The image path is C:\\webstuff\\mypage\\gifs\\garden.gif.'
```

The single quote and double quote escape sequences can be used to include quotes in string literals. This example shows embedded quotes.

```
'The caption reads, \"After the snow of \'97. Grandma\'s house is covered.\"'
```

JScript uses the intrinsic **char** data type to represent a single character. A string containing one character or one escape sequence can be assigned to a variable of type **char**, although the string is not itself of type **char**.

A string that contains zero characters ("") is an empty (zero-length) string.

See Also

Data in JScript | JScript Expressions | String Data Type | String Object

Data Type Summary

JScript .NET provides many data types to use in your programs. These types can be divided into two main categories, value data types and reference data types (also referred to as JScript objects). To add types to JScript, you can import namespaces or packages that contain new data types, or you can define new classes that can be used as new data types.

The following table shows the value data types supported by JScript. The second column describes the equivalent Microsoft .NET Framework data type. You can declare a variable of the .NET Framework type or the JScript value type and achieve exactly the same results. The storage size (where applicable) and range are also given for each type. The third column lists the amount of storage required for one instance of a given type, if applicable. The fourth column provides the range of values that can be stored by a given type.

JScript value type	**.NET Framework type**	**Storage size**	**Range**
boolean	System.Boolean	N/A	**true** or **false**
char	System.Char	2 bytes	Any Unicode character
float (single-precision floating-point)	System.Single	4 bytes	Approximate range is -10^{38} to 10^{38} with accuracy of about 7 digits. Can represent numbers as small as 10^{-44}
Number, double (double-precision floating-point)	System.Double	8 bytes	Approximate range is -10^{308} to 10^{308} with accuracy of about 15 digits. Can represent numbers as small as 10^{-323}
decimal	System.Decimal	12 bytes (integral part)	Approximate range is -10^{28} to 10^{28} with accuracy of 28 digits. Can represent numbers as small as 10^{-28}
byte (unsigned)	System.Byte	1 byte	0 to 255
ushort (unsigned short integer)	System.UInt16	2 bytes	0 to 65,535
uint (unsigned integer)	System.UInt32	4 bytes	0 to 4,294,967,295
ulong (unsigned extended integer)	System.UInt64	8 bytes	0 to approximately 10^{20}
sbyte (signed)	System.SByte	1 byte	-128 to 127
short (signed short integer)	System.Int16	2 bytes	-32,768 to 32,767
int (signed integer)	System.Int32	4 bytes	-2,147,483,648 to 2,147,483,647
long (signed extended integer)	System.Int64	8 bytes	Approximately -10^{19} to 10^{19}
void	N/A	N/A	Used as the return type for a function that does not return a value.

The next table shows the reference data types (JScript objects) that JScript provides and that can be used as types. Reference types do not have a pre-defined specific storage size.

JScript reference type	**.NET Framework type**	**Refers to**
ActiveXObject	No direct equivalent	An Automation object.
Array	Interoperates with System.Array and typed arrays	Arrays of any type.
Boolean	Interoperates with System.Boolean	A Boolean value, either **true** or **false.**
Date	Interoperates with System.DateTime	Dates are implemented using the JScript **Date** object. The range is approximately 285,616 years on either side of January 1, 1970.
Enumerator	No direct equivalent	An enumeration of items in a collection. For backwards compatibility only.
Error	No direct equivalent	An **Error** object.

Function	No direct equivalent	A **Function** object.
Number	Interoperates with System.Double	A numeric value with an approximate range of -10^{308} to 10^{308} and with an accuracy of about 15 digits. Can represent numbers as small as 10^{-323}.
Object	Interoperates with System.Object	An **Object** reference.
RegExp	Interoperates with System.Text.RegularExpression.Regex Class	A regular expression object.
String Data Type (variable-length)	System.String	0 to approximately 2 billion Unicode characters. Each character is 16 bits (two bytes).
String Object (variable-length)	Interopertates with System.String	0 to approximately 2 billion Unicode characters. Each character is 16 bits (two bytes).
VBArray	No direct equivalent	A read-only Visual Basic array. For backwards compatibility only.

See Also

Data Types | User Defined Data Types | Objects | import Statement | package Statement | class Statement | JScript Objects | Copying, Passing, and Comparing Data

User Defined Data Types

Sometimes you need a data type that is not provided by JScript. In this situation, you can import a package that defines a new class, or you can create your own data type using the **class** statement. Classes can be used for type annotation and for making typed arrays in exactly the same way as the predefined data types in JScript.

The following example uses the **class** statement to define a new data type, `myIntVector`. The new type is used in a function declaration to denote the type of the function's parameter. A variable is also type annotated with the new type.

```
// Define a class that stores a vector in the x-y plane.
class myIntVector {
   var x : int;
   var y : int;
   function myIntVector(xIn : int, yIn : int) {
      x = xIn;
      y = yIn;
   }
}

// Define a function to compute the magnitude of the vector.
// Passing the parameter as a user defined data type.
function magnitude(xy : myIntVector) : double {
   return( Math.sqrt( xy.x*xy.x + xy.y*xy.y ) );
}

// Declare a variable of the user defined data type.
var point : myIntVector = new myIntVector(3,4);
print(magnitude(point));
```

This output of this code is:

```
5
```

See Also

Data Types | class Statement | package Statement | JScript Objects | Type Annotation

Typed Arrays

A typed array is a data type that can annotate variables, constants, functions, and parameters as if it were an intrinsic data type. Every typed array has a base data type, and each element of the array is of that base type. The base type can itself be an array type, allowing for arrays of arrays.

A data type that is followed by a set of square brackets defines a one-dimensional typed array. To define an *n*-dimensional array, the base data type is followed by a set of square brackets with *n*-1 commas between the brackets.

No storage is initially allocated for a variable of a typed array type, and the initial value is **undefined**. To initialize an array variable, use the **new** operator, an array literal, an array constructor, or another array. The initialization can occur when the typed array variable is declared or later, as with variables of other types. A type mismatch error will result if the dimensionality of a variable or parameter does not match the dimensionality (or type) of the array assigned to the variable or passed to the parameter.

Using an array constructor, you can create an array of a given native type with a specified (fixed) size. Each argument must be an expression that evaluates to a non-negative integer. The value of each argument determines the size of the array in each dimension; the number of arguments determines the dimensionality of the array.

The following shows some simple array declarations:

```
// Simple array of strings; initially empty. The variable 'names' itself
// will be null until something is assigned to it
var names : String[];

// Create an array of 50 objects; the variable 'things' won't be null,
// but each element of the array will be until they are assigned values.
var things : Object[] = new Object[50];
// Put the current date and time in element 42.
things[42] = new Date();

// An array of arrays of integers; initially it is null.
var matrix : int[][];
// Initialize the array of arrays.
matrix = new (int[])[5];
// Initialize each array in the array of arrays.
for(var i = 0; i<5; i++)
   matrix[i] = new int[5];
// Put some values into the matrix.
matrix[2][3] = 6;
matrix[2][4] = 7;

// A three-dimensional array
var multidim : double[,,] = new double[5,4,3];
// Put some values into the matirx.
multidim[1,3,0] = Math.PI*5.;
```

See Also

var Statement | new Operator | function Statement | Type Annotation | Data Types

Type Conversion

Type conversion is the process of changing a value from one type to another. For example, you can convert the string, "1234" to a number. Furthermore, you can convert data of any type to the **String** type. Some type conversions will never succeed. For example, a **Date** object cannot be converted to an **ActiveXObject** object.

Type conversions may either be *widening* or *narrowing*: widening conversions never overflow and always succeed, whereas narrowing conversions entail the possible loss of information and may fail.

Both types of conversion may be explicit (using the data type identifier) or implicit (without the data type identifier). Valid explicit conversions always succeed, even if it results in a loss of information. Implicit conversions succeed only when the process loses no data; otherwise they fail and generate a compile or run-time error.

Lossy conversions happen when the original data type does not have an obvious analogue in the target conversion type. For example, the string, "Fred", cannot be converted to a number. In these cases, a default value is returned from the type conversion function. For the **Number** type, the default is **NaN**; for the **int** type the default is the number zero.

Some types of conversions, such as from a string to a number, are time-consuming. The fewer conversions your program uses, the more efficient it will be.

Implicit Conversions

Most type conversions, such as assigning a value to a variable, occur automatically. The data type of the variable determines the target data type of the expression conversion.

This example demonstrates how data can be implicitly converted between an **int** value, a **String** value, and a **double** value.

```
var i : int;
var d : double;
var s : String;
i = 5;
s = i;  // Widening: the int value 5 coverted to the String "5".
d = i;  // Widening: the int value 5 coverted to the double 5.
s = d;  // Widening: the double value 5 coverted to the String "5".
i = d;  // Narrowing: the double value 5 coverted to the int 5.
i = s;  // Narrowing: the String value "5" coverted to the int 5.
d = s;  // Narrowing: the String value "5" coverted to the double 5.
```

When this code is compiled, compile-time warnings may state that the narrowing conversions may fail or are slow.

Implicit narrowing conversions may not work if the conversion requires a loss of information. For example, the following lines will not work.

```
var i : int;
var f : float;
var s : String;
f = 3.14;
i = f;  // Run-time error. The number 3.14 cannot be represented with an int.
s = "apple";
i = s;  // Run-time error. The string "apple" cannot be converted to an int.
```

Explicit Conversions

To explicitly convert an expression to a particular data type, use the data type identifier followed by the expression to convert in parentheses. Explicit conversions require more typing than implicit conversions, but you can be more certain of the result. Furthermore, explicit conversions can handle lossy conversions.

This example demonstrates how data can be explicitly converted between an **int** value, a **String** value, and a **double** value.

```
var i : int;
var d : double;
var s : String;
i = 5;
s = String(i);  // Widening: the int value 5 coverted to the String "5".
d = double(i);  // Widening: the int value 5 coverted to the double 5.
```

```
s = String(d);  // Widening: the double value 5 coverted to the String "5".
i = int(d);     // Narrowing: the double value 5 coverted to the int 5.
i = int(s);     // Narrowing: the String value "5" coverted to the int 5.
d = double(s);  // Narrowing: the String value "5" coverted to the double 5.
```

Explicit narrowing conversions will usually work, even if the conversion requires a loss of information. Explicit conversion cannot be used to convert between incompatible data types. For example, you cannot convert **Date** data to or from **RegExp** data. In addition, conversions are not possible for some values because there is no sensible value to which to convert. For example, an error is thrown when attempting to explicitly convert the double value **NaN** to a **decimal**. This occurs because there is no natural **decimal** value that could be identified with **NaN**.

In this example, a number with a decimal part is converted to an integer, and a string is converted to an integer.

```
var i : int;
var d : double;
var s : String;
d = 3.14;
i = int(d);
print(i);
s = "apple";
i = int(s);
print(i);
```

The output is

```
3
0
```

The behavior of the explicit conversion depends on both the original data type and the target data type.

See Also

undefined Property | JScript Data Types | Type Annotation

JScript Variables and Constants

In any programming language, data represents information. For example, this string literal contains a question:

```
'How old am I?'
```

Variables and constants store data that scripts can easily reference by using the name of the variable or constant. Data stored by a variable may change as a program runs, while data stored by a constant cannot change. A script that uses a variable actually accesses the data that the variable represents. Here is an example in which the variable named `NumberOfDaysLeft` is assigned the value derived from the difference between `EndDate` and `TodaysDate`.

```
NumberOfDaysLeft = EndDate - TodaysDate;
```

In a mechanical sense, a script uses variables to store, retrieve, and manipulate the values that appear in scripts. Constants reference data that that does not change. Always create a meaningful variable name to help you remember the variable's purpose and to help others determine the script's functionality.

In This Section

Types of JScript Variables and Constants
Discusses how to choose an appropriate data type for a variable and the benefits of proper variable data type selection.

Declaring JScript Variables and Constants
Explains how to declare typed and untyped variables and constants and how to initialize them.

Scope of Variables and Constants
Illustrates the difference between global and local scopes in JScript and how the local scope shadows the global scope.

Undefined Values
Explains the concept of undefined values, how to determine if a variable or property is undefined, and how to undefine variables and properties.

Related Sections

JScript Identifiers
Explains how to create valid names for identifiers in JScript.

JScript Data Types
Includes links to topics that explain how to use primitive data types, reference data types, and .NET Framework data types in JScript.

JScript Assignments and Equality
Explains how JScript assigns values to variables, array elements, and property elements and explains the equality syntax used by JScript.

JScript Reference
Lists elements that comprise JScript Language Reference and links to topics that explain the details behind the proper use of language elements.

Types of JScript Variables and Constants

JScript .NET has a number of data types that you may find useful as you write your code. Efficient use of data types allows a program to load and run faster than a program that uses the default JScript data type. In addition, the compiler can provide valuable error messages and warnings about the misuse of the various types.

If, for example, a variable will always store an integer whose value will never exceed 1,000,000, it makes no sense to use an 8-byte **double** type. Instead, the most efficient type for this data is **int**, which is a 4-byte integer type that stores data values from –2,147,483,648 to 2,147,483,647.

Declaring a variable or constant with type annotation ensures that the variable or constant stores only data of the appropriate type. JScript has a number of other data types for type annotation. For more information, see Data Type Summary. To add types to JScript, either import an assembly that contains types or declare user-defined types (classes).

See Also

JScript Variables and Constants | JScript Data Types | Declaring JScript Variables and Constants

Declaring JScript Variables and Constants

A JScript program must specify the name of each variable that the program will use. In addition, the program may specify what data type each variable will store. Both of these tasks are accomplished with the **var** statement.

Declaring Typed Variables and Constants

In JScript, you can declare a variable and concurrently declare its type using type annotation. In the following example, the variable `count` is declared to be of type **int** (integer). Since no initial value is provided, `count` has the default value of **int**, which is 0 (zero).

```
var count : int; // An integer variable.
```

You can also assign an initial value to a variable:

```
var count : int = 1; // An initialized integer variable.
```

Constants, which are declared in much the same way as variables, must be initialized. Once a constant value is defined, its value cannot be changed. For example:

```
const daysInWeek : int = 7;            // An integer constant.
const favoriteDay : String = "Friday"; // A string constant.
const maxDaysInMonth : int = 31, maxMonthsInYear : int = 12
```

Of course, when you declare a variable of a specific type, the assigned value must make sense for that type. For example, it does not make sense to assign a character string value to an integer variable. When you do this, the program throws a **TypeError** exception that indicates a type mismatch in your code. A **TypeError** is one kind of exception or error that can occur in a running script. A catch block can catch exceptions thrown by a JScript program. For more information, see try...catch...finally Statement.

You can concurrently declare the type and initial value of multiple variables, although it is easier to read code when each declaration is on a separate line. For example, this code segment is hard to read:

```
var count : int = 1; amount : int = 12, level : double = 5346.9009
```

The following code segment is easier to read:

```
var count : int = 1;
var amount : int = 12;
var level : double = 5346.9009;
```

Something else to keep in mind when declaring several variables on a single line is that a type annotation applies only to the variable that immediately precedes it. In the following code, `x` is an **Object** because that is the default type and `x` does not specify a type, while *y* is an **int**.

```
var x, y : int;
```

Declaring Untyped Variables and Constants

You do not need to use typed variables, but programs that use untyped variables are slower and more prone to errors.

The following simple example declares a single variable named `count`.

```
var count;  // Declare a single declaration.
```

Without a specified data type, the default type for a variable or constant is **Object**. Without an assigned value, the default value of the variable is **undefined**. The following code demonstrates these defaults for a command-line program:

```
var count; // Declare a single declaration using default type and value.
```

```
print(count); //Print the value of count.
print(typeof(count)); // Prints undefined.
```

You can give a variable an initial value without declaring its type:

```
var count = 1; // An initialized variable.
```

The following example declares several variables using a single **var** statement:

```
var count, amount, level;  // multiple declarations with a single var keyword.
```

To declare a variable and initialize it without assigning it a particular value, assign it the JScript value **null**. Here is an example.

```
var bestAge = null;
```

A declared variable without an assigned a value exists but has the JScript value **undefined**. Here is an example.

```
var currentCount;
var finalCount = 1 * currentCount; // finalCount has the value NaN since currentCount is unde
fined.
```

In JScript, the main difference between **null** and **undefined** is that **null** converts to zero (although it is not zero), while **undefined** converts to the special value **NaN** (Not a Number). Ironically, a **null** value and an **undefined** value always compare as equal when using the equality operator (==).

The process of declaring untyped constants is similar to the process of declaring variables, but you must provide an initial value for untyped constants. For example:

```
const daysInWeek  = 7;
const favoriteDay  = "Friday";
const maxDaysInMonth  = 31, maxMonthsInYear = 12
```

Declaring Variables Without var

You can declare a variable without using the **var** keyword in the declaration and assign a value to it. This is known as an *implicit declaration* and it is not recommended. An implicit declaration creates a property of the global object with the assigned name; the property behaves like a variable with global scope visibility. When you declare a variable at the procedure level, though, you typically do not want it to be visible at the global scope. In this case, you *must* use the **var** keyword in your variable declaration.

```
noStringAtAll = ""; // The variable noStringAtAll is declared implicitly.
```

You cannot use a variable that has never been declared.

```
var volume = length * width; // Error - length and width do not yet exist.
```

> **Note** Declaring variables without the **var** keyword generates a compile-time error when running in fast mode, the default mode for JScript .NET. To compile a program from the command line that does not use the **var** keyword, you must turn off the fast option by using **/fast-**. It is not safe to turn off the fast option in ASP.NET because of threading issues.

See Also

JScript Identifiers | JScript Variables and Constants | JScript Data Types | Compiling JScript code from the Command Line

Scope of Variables and Constants

JScript has three scopes, global, local, and class. If you declare a variable or constant outside a function or class definition, it is a global variable, and its value is accessible and modifiable throughout your program. If you declare a variable inside a function definition, that variable is local. It is created and destroyed every time the function is executed; it cannot be accessed from outside the function. If you declare a variable inside a class definition, that variable is available inside the class, and it cannot be accessed directly from the global scope. For more information, see Class-based Objects.

Languages such as C++ also have "block scope"; any set of braces ({}) defines a new scope. JScript does not support block scopes.

A local variable can have the same name as a global variable, but it is entirely distinct and separate. Consequently, changing the value of one variable has no effect on the other. Inside the function in which the local variable is declared, only the local version has meaning. This is known as *visibility*.

```
// Define two global variables.
var name : String = "Frank";
var age : int = "34";

function georgeNameAge() {
   var name : String; // Define a local variable.
   name = "George";   // Modify the local variable.
   age = 42;          // Modify the global variable.
   print(name + " is " + age + " years old.");
}

print(name + " is " + age + " years old.");
georgeNameAge();
print(name + " is " + age + " years old.");
```

The output of this program shows that a local variable can be modified without changing the value of the global variable. Changes to the global variable from inside the function do affect the value in the global scope.

```
Frank is 34 years old.
George is 42 years old.
Frank is 42 years old.
```

Since JScript processes variable and constant declarations before executing any code, it does not matter whether the declaration is inside a conditional block or some other construct. Once JScript has found all the variables and constants, it executes the code in the function. This means that the value of a local constant is undefined until the constant declaration statement is reached and that a local variable is undefined until the variable is assigned to in the function.

Sometimes this results in unexpected behaviors. Consider the following program.

```
var aNumber = 100;
var anotherNumber = 200;
function tweak() {
   var s = "aNumber is " + aNumber + " and ";
   s += "anotherNumber is " + anotherNumber + "\n";
   return s;
   if (false)  {
      var aNumber;                // This statement is never executed.
      aNumber = 123;              // This statement is never executed.
      const anotherNumber = 42;   // This statement is never executed.
   } // End of the conditional.
} // End of the function definition.

print(tweak());
```

This output of this program is:

```
aNumber is undefined and anotherNumber is undefined
```

You might expect that `aNumber` would be 100 or 123 and that `anotherNumber` would be 200 or 42, but both values are **undefined**. Since both `aNumber` and `anotherNumber` are defined with local scope, they shadow the global variable and constant with the same name. Since the code that initializes the local variable and constant is never run, their values are **undefined**.

Explicit variable declaration is required in fast mode. When fast mode is turned off, implicit variable declaration is required. An implicitly declared variable inside a function — that is, one that appears on the left side of an assignment expression without the var keyword —is a global variable.

See Also

JScript Variables and Constants | Undefined Values

Undefined Values

In JScript, you can declare a variable without assigning a value to it. A type-annotated variable assumes the default value for that type. For example, the default value for a numeric type is zero, and the default for the **String** data type is the empty string. However, a variable without a specified data type has an initial value of **undefined** and a data type of **undefined**. Likewise, code that accesses an expando object property or an array element that does not exist returns a value of **undefined**.

To determine if a variable or object property exists, compare it to the keyword **undefined** (which will work only for a declared variable or property), or check if its type is "undefined" (which will work even for an undeclared variable or property). In the following code example, assume that the programmer is trying to test if the variable `x` has been declared:

```
// One method to test if an identifier (x) is undefined.
// This will always work, even if x has not been declared.
if (typeof(x) == "undefined"){
   // Do something.
}
// Another method to test if an identifier (x) is undefined.
// This gives a compile-time error if x is not declared.
if (x == undefined){
   // Do something.
}
```

Another way to check if a variable or object property is undefined is to compare the value to **null**. A variable that contains **null** contains "*no value*" or "*no object*." In other words, it holds no valid number, string, Boolean, array, or object. You can erase the contents of a variable (without deleting the variable) by assigning it the **null** value. Note that the value **undefined** and **null** compare as equal using the equality (==) operator.

> **Note** In JScript, **null** does not compare as equal to 0 using the equality operator. This behavior is different from other languages, such as C and C++.

In this example, the object `obj` is tested to see if it has the property `prop`.

```
// A third method to test if an identifier (obj.prop) is undefined.
if (obj.prop == null){
   // Do something.
}
```

This comparison is **true**,

- if the property `obj.prop` contains the value **null**,
- if the property `obj.prop` does not exist.

There is another way to check if an object property exists. The **in** operator returns true if the specified property is in the provided object. For example, the following code tests if the property `prop` is in the object `obj`.

```
if ("prop" in someObject)
// someObject has the property 'prop'
```

To remove a property from an object, use the **delete** operator.

See Also

JScript Variables and Constants | Data in JScript | JScript Data Types | Data Types | null Literal | undefined Property | in Operator | delete Operator

JScript Objects

A JScript object is an encapsulation of data and functionality. Objects are comprised of properties (values) and methods (functions). Properties are the data component of the object, while methods provide the functionality to manipulate the data or the object. JScript supports five kinds of objects: intrinsic objects, prototype-based objects, class-based objects, host objects (provided by a host, such as **Response** in ASP.NET) and .NET Framework classes (external components).

The **new** operator in conjunction with the constructor function for the selected object creates and initializes an instance of an object. Here are a few examples that use constructors.

```
var myObject = new Object();           // Creates a generic object.
var birthday = new Date(1961, 5, 10);  // Creates a Date object.
var myCar : Car = new Car("Pinto");    // Creates a user-defined object.
```

JScript supports two types of user-defined objects (class-based and prototype-based). Both types have unique advantages and disadvantages. Prototype-based objects are dynamically extensible, but they are slow and do not interoperate efficiently with objects from other .NET Framework languages. Class-based objects, on the other hand, can extend existing .NET Framework classes, provide type safety, and ensure efficient operation. Class-based objects can be dynamically extensible (like prototype-based objects) by defining the class with the **expando** modifier.

In This Section

Intrinsic Objects
Lists some of the common objects used in JScript scripts and links to information that describes how to use them.

Class-based Objects
Provides a guide to using the JScript class-based object model and describes how to define classes (with methods, fields, and properties), how to define a class that inherits from another class, and how to define expando classes.

Prototype-based Objects
Provides a guide to using the JScript prototype-based object model and links to information that describes custom constructor functions and inheritance for prototype-based objects.

Related Sections

JScript Data Types
Includes links to topics that explain how to use primitive data types, reference data types, and .NET Framework data types in JScript.

JScript Reference
Lists elements that comprise JScript Language Reference and links to topics that explain the details behind the proper use of language elements.

Introduction to ASP.NET
Introduces ASP.NET, explains how it can be used with any .NET compatible language, including JScript .NET, to create enterprise-class Web applications, and links to reference information.

Introduction to the .NET Framework Class Library
Introduces the .NET Framework class library, explains naming conventions and system namespaces, and links to reference information.

Intrinsic Objects

JScript provides 16 intrinsic objects as part of the language specification. Each intrinsic object has associated methods and properties, which are described in detail in the language reference. Several commonly used objects are discussed in this section to illustrate the basic syntax and use of intrinsic objects.

In This Section

JScript Array Object
: Describes how to use array objects, how to take advantage of their expando properties, and how they compare to typed arrays.

JScript Date Object
: Describes the range of acceptable dates and how to create an object with either the current date and time or an arbitrary date and time.

JScript Math Object
: Illustrates how to use methods and properties to manipulate numerical data.

JScript Number Object
: Explains the purpose of the number object and the meaning of its properties.

JScript Object Object
: Describes how to add expando properties and methods to objects and explains the difference between using the dot operator and the index operator to access object members.

JScript String Object
: Explains the purpose of the string object and how string literals can use the methods of the **String** object.

Related Sections

JScript Objects
: Links to topics that explain the syntax and uses of the intrinsic objects in JScript.

JScript Reference
: Lists elements that comprise JScript Language Reference and links to topics that explain the details behind the proper use of language elements.

Objects
: Lists all the objects provided by the JScript language and links to language reference information that explains the proper use and syntax for each object.

JScript Array Object

An **Array** object is a variable that groups related pieces of data. A unique number, called an index or subscript, references each piece of data in the array. To access the data stored in the array, the array identifier and the index are combined with the index operator "[]", for example, `theMonths[0]`.

To create a new array, use the **new** operator and the **Array** constructor. In this example, the array constructor is used to construct an array with length 12. Then, data is entered into the array.

```
var theMonths = new Array(12);
theMonths[0] = "Jan";
theMonths[1] = "Feb";
theMonths[2] = "Mar";
theMonths[3] = "Apr";
theMonths[4] = "May";
theMonths[5] = "Jun";
theMonths[6] = "Jul";
theMonths[7] = "Aug";
theMonths[8] = "Sep";
theMonths[9] = "Oct";
theMonths[10] = "Nov";
theMonths[11] = "Dec";
```

When you create an array using the **Array** keyword, JScript includes a **length** property, which records the number of entries. If you do not specify a number, the length is set to zero, and the array has no entries. If you specify a number, the length is set to that number. If you specify more than one parameter, the parameters are used as entries in the array. In addition, the number of parameters is assigned to the length property, as in the following example, which is equivalent to the preceding example.

```
var theMonths = new Array("Jan", "Feb", "Mar", "Apr", "May", "Jun",
"Jul", "Aug", "Sep", "Oct", "Nov", "Dec");
```

Array literals provide another technique for entering data in an array. For more information, see Array Data.

The **Array** object stores *sparse* arrays. That is, if an array has three elements that are numbered 0, 1, and 2, element 50 can exist without the presence of elements 3 through 49. JScript automatically changes the value of the **length** property when you add elements to an **Array** object. Array indices in JScript always start at 0, not 1, so the length property is always one greater than the largest index in the array.

Using expando properties of arrays

Array objects, just as any other object based on the JScript **Object** object, support expando properties. Expando properties are new properties that you dynamically add and delete from an array, like array indices. Unlike array indices, which must be whole numbers, expando properties are strings. In addition, adding or deleting expando properties does not change the **length** property.

For example:

```
// Initialize an array with three elements.
var myArray = new Array("Hello", 42, new Date(2000,1,1));
print(myArray.length); // Prints 3.
// Add some expando properties. They will not change the length.
myArray.expando = "JScript .NET";
myArray["another Expando"] = "Windows";
print(myArray.length); // Still prints 3.
```

Typed Arrays

Another faster way to create the `theMonths` array shown above is to create a typed (native) array, in this case, an array of strings:

```
var theMonths : String[] = ["Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug", "Sep", "O
ct", "Nov", "Dec"];
```

Elements of typed arrays can be accessed faster than elements in JScript array objects. Typed arrays are compatible with arrays in other .NET Framework languages and provide type safety.

JScript **Arrays** objects are very flexible and great to use for lists, queues, stacks, and so on, but native arrays are much better for storing fixed-size items of the same type. In general, unless the special features of the **Array** object are needed (dynamic resizing and so on), typed arrays should be used.

All non-destructive JScript **Array** methods (methods that do not change the length) can be called on typed arrays.

See Also

Intrinsic Objects | Array Object

JScript Date Object

The JScript **Date** object can be used to represent arbitrary dates and times, to get the current system date, and to calculate differences between dates. It has several predefined properties and methods. The **Date** object stores a day of the week; a month, day, and year; and a time in hours, minutes, seconds, and milliseconds. This information is based on the number of milliseconds since January 1, 1970, 00:00:00.000 Coordinated Universal Time (UTC), formerly known as Greenwich Mean Time. JScript can handle dates that are in the approximate range from 250,000 B.C. to 255,000 A.D., although some formatting functionality is only supported for dates in the range 0 A.D. through 9999 A.D.

To create a new **Date** object, use the **new** operator. The following example calculates the number of days that have passed and the number of days that remain for the current year.

```
// Get the current date and read the year.
var today : Date = new Date();
// The getYear method should not be used. Always use getFullYear.
var thisYear : int = today.getFullYear();

// Create two new dates, one for January first of the current year,
// and one for January first of next year. The months are numbered
// starting with zero.
var firstOfYear : Date = new Date(thisYear,0,1);
var firstOfNextYear : Date = new Date(thisYear+1,0,1);

// Calculate the time difference (in milliseconds) and
// convert the differnce to days.
const millisecondsToDays = 1/(1000*60*60*24);
var daysPast : double = (today - firstOfYear)*millisecondsToDays;
var daysToGo : double = (firstOfNextYear - today)*millisecondsToDays;

// Display the information.
print("Today is: "+today+".");
print("Days since first of the year: "+Math.floor(daysPast));
print("Days until the end of the year: "+Math.ceil(daysToGo));
```

The output of this program is similar to this:

```
Today is: Sun Apr 1 09:00:00 PDT 2001.
Days since first of the year: 90
Days until the end of the year: 275
```

See Also

Intrinsic Objects | Date Object

JScript Math Object

The **Math** object has a number of intrinsic properties and methods. The properties are specific numbers. One of these specific numbers is the value of pi (approximately 3.14159...). This is the value of the **Math.PI** property, which is used in the following example.

```
// A radius variable is declared and assigned a numeric value.
var radius = 4;
var area = Math.PI * radius * radius;
// Note capitalization of Math and PI.
```

One built-in method of the **Math** object is the exponentiation method, or **pow**, which raises a number to a specified power. The following example uses both pi and exponentiation.

```
// This formula calculates the volume of a sphere with the given radius.
var volume = (4/3)*(Math.PI*Math.pow(radius,3));
```

Another

```
var x = Math.floor( Math.random()*10 ) + 1;
```

The **Math** object cannot be explicitly constructed; it is always available to the program.

See Also

Intrinsic Objects | Math Object

JScript Number Object

The primary purpose of the **Number** object is to collect properties and methods that are used for the default numeric data type, the **Number** data type. The numeric constants provided by the properties of the **Number** object are listed in the table below.

Property	Description
MAX_VALUE	Largest possible number, about 1.79E+308; can be positive or negative. (Value varies slightly from system to system.)
MIN_VALUE	Smallest possible number, about 2.22E-308; can be positive or negative. (Value varies slightly from system to system.)
NaN	Special nonnumeric value, "not a number."
POSITIVE_INFINITY	Any positive value larger than the largest positive number (**Number.MAX_VALUE**) is automatically converted to this value; represented as infinity.
NEGATIVE_INFINITY	Any value more negative than the largest negative number (-**Number.MAX_VALUE**) is automatically converted to this value; represented as -infinity.

Number.NaN is a special property that is defined as *not a number*. **Number.NaN** is returned when an expression that cannot be represented as a number is used in a numeric context. For example, **NaN** is returned when either the string "Hello" or 0/0 (zero divided by zero) is used as a number. **NaN** compares as unequal to any number and to itself. To test for a **NaN** result, do not compare against **Number.NaN**; use the **isNaN** method of the **Global** object instead.

The **toLocaleString** method of the **Number** object produces a string value that represents the value of the number formatted as appropriate for the host environment's current locale. The formatting used makes large numbers easier to read by separating groups of digits to the left of the decimal point with a (locale dependent) character. For more information, see toLocaleString Method.

See Also

Intrinsic Objects | Number Object | toLocaleString Method

JScript Object Object

All objects in JScript based on the **Object** object support *expando* properties, or properties that can be added and removed while the program is running. These properties can have any name, including numbers. A name of a property that is a simple identifier can be written after a period that follows the object name, such as:

```
var myObj = new Object();
// Add two expando properties, 'name' and 'age'
myObj.name = "Fred";
myObj.age = 53;
```

You can also access an object's properties using the index operator, **[]**. This is required if the name of the property is not a simple identifier, or if the name of the property is not known when you write the script. An arbitrary expression, including a simple identifier, inside square brackets can index the property. The names of all expando properties in JScript are converted to strings before being added to the object.

When using the index operator, the object is treated as an *associative array*. An associative array is a data structure that dynamically associates arbitrary data values with arbitrary strings. In this example, expando properties are added that do not have simple identifiers.

```
var myObj = new Object();
// This identifier contains spaces.
myObj["not a simple identifier"] = "This is the property value";
// This identifier is a number.
myObj[100] = "100";
```

Although the index operator is more commonly associated with accessing array elements, the index is always the property name expressed as a string literal when used with objects.

Array objects have a special **length** property that changes when new elements are added; in general, objects do not have a length property even when the index operator is used to add properties.

Notice the important difference between the two ways of accessing object properties.

Operator	**The property name is treated as**	**Meaning the property name**
Period (.)	An identifier	*Cannot* be manipulated as data
Index ([])	A string literal	*Can* be manipulated as data

This difference becomes useful when you do not know the property names until runtime (for example, when you are constructing objects based on user input). To extract all the properties from an associative array, you must use the **for ... in** loop.

See Also

Intrinsic Objects | Object Object

JScript String Object

A **String** object in JScript represents textual data such as words, sentences, and so on. String objects are rarely created explicitly with the **new** operator because they are usually created implicitly by assigning a string literal to a variable. For more information, see String Object.

The **String** object has many built-in methods. One of these is the **substring** method, which returns part of the string. It takes two numbers as its arguments. The first number is the zero-based index that indicates the beginning of the substring, and the second number indicates the end of the substring.

```
var aString : String = "0123456789";
var aChunk : String = aString.substring(4, 7);  // Sets aChunk to "456".
```

The **String** object also has a **length** property. This property contains the number of characters in the string (0 for an empty string). This a numeric value and can be used directly in calculations. This example obtains the length of a string literal.

```
var howLong : int = "Hello World".length  // Sets the howLong variable to 11.
```

See Also

Intrinsic Objects | String Object | String Data

Class-based Objects

Since JScript .NET is a class-based, object-oriented programming language, it is possible to define classes that can inherit from other classes. Defined classes can have methods, fields, properties, and subclasses. Inheritance enables classes to build upon existing classes and override selected base-class methods and properties. The classes in JScript .NET, which are similar to the classes in C++ and C#, are quite different from the prototype-based objects.

In This Section

Creating Your Own Classes
Describes how to define a class with fields, methods, and constructors.

Advanced Class Creation
Describes how to define a class with properties, how to inherit from a class, and how to create a class that supports expando properties.

Related Sections

JScript Objects
Includes links to topics that explain the syntax and uses of the intrinsic JScript objects.

JScript Modifiers
Describes the modifiers that can be used to control the visibility of class members, how classes inherit, and how classes behave.

Prototype-based Objects
Provides a guide to using the JScript prototype-based object model and links to information that describes custom constructor functions and inheritance for prototype-based objects.

JScript Reference
Lists elements that comprise JScript Language Reference and links to topics that explain the details behind the proper use of language elements.

Creating Your Own Classes

The **class** statement defines classes. By default, class members are publicly accessible, which means that any code that can access the class can also access the class member. For more information, see JScript Modifiers.

Classes with Fields

Fields define the data used by an object and are similar to the properties in a prototype-based object. Here is an example of a simple class that has two fields. An instance of the class is created with the **new** operator:

```
class myClass {
   const answer : int = 42; // Constant field.
   var distance : double;   // Variable field.
}

var c : myClass = new myClass;
c.distance = 5.2;
print("The answer is " + c.answer);
print("The distance is " + c.distance);
```

The output of this program is:

```
The answer is 42
The distance is 5.2
```

Classes with Methods

Classes can also contain methods, which are functions contained in the class. Methods define the functionality to manipulate the data of an object. The class `myClass` defined above can be redefined to include a method.

```
class myClass {
   const answer : int = 42;         // Constant field.
   var distance : double;           // Variable field.
   function sayHello() :String {    // Method.
      return "Hello";
   }
}

var c : myClass = new myClass;
c.distance = 5.2;
print(c.sayHello() + ", the answer is " + c.answer);
```

The output of this program is:

```
Hello, the answer is 42
```

Classes with Constructors

You can define a constructor for a class. A constructor, which is a method with the same name as the class, is run when a class is created with the **new** operator. You may not specify a return type for a constructor. In this example, a constructor is added to the `myClass` class.

```
class myClass {
   const answer : int = 42;         // Constant field.
   var distance : double;           // Variable field.
   function sayHello() :String {    // Method.
      return "Hello";
   }
   // This is the constructor.
   function myClass(distance : double) {
      this.distance = distance;
   }
}

var c : myClass = new myClass(8.5);
print("The distance is " + c.distance);
```

The output of this program is:

```
The distance is 8.5
```

See Also

Class-based Objects | JScript Objects | Advanced Class Creation

Advanced Class Creation

When you define a JScript class, you can assign properties and the defined class can subsequently inherit from other classes. Properties, which are class members similar to fields, provide greater control over how data is accessed. By using inheritance, a class can extend (or add behavior to) another class.

You can define a class so that instances of the class support expando properties. This means that class-based objects can have properties and methods dynamically added to the object. Class-based expando objects provide some of the same functionality as prototype-based objects.

Classes with Properties

JScript uses **function get** and **function set** statements to specify properties. You can specify either or both accessors to create read-only, write-only, or read-write properties, although write-only properties are rare and may indicate a problem with the design of the class.

The calling program accesses properties in the same way that it accesses fields. The main difference is that the getter and setter for the property are used to perform the access, whereas fields are accessed directly. A property enables the class to check that only valid information is being entered, to keep track of the number of times the property is read or set, to return dynamic information, and so on.

Properties are usually used to access private or protected fields of the class. Private fields are marked with the **private** modifier, and only other members of the class can access them. Protected fields are marked with the **protected** modifier, and only other members of the class or derived classes can access them. For more information, see JScript Modifiers.

In this example, properties are used to access a protected field. The field is protected to prevent outside code from changing its value while allowing derived classes to access it.

```
class Person {
   // The name of a person.
   // It is protected so derived classes can access it.
   protected var name : String;

   // Define a getter for the property.
   function get Name() : String {
      return this.name;
   }

   // Define a setter for the property which makes sure that
   // a blank name is never stored in the name field.
   function set Name(newName : String) {
      if (newName == "")
         throw "You can't have a blank name!";
      this.name = newName;
   }
   function sayHello() {
      return this.name + " says 'Hello!'";
   }
}

// Create an object and store the name Fred.
var fred : Person = new Person();
fred.Name = "Fred";
print(fred.sayHello());
```

The output of this code is:

```
Fred says 'Hello!'
```

When a blank name is assigned to the `Name` property, an error is generated.

Inheritance from Classes

The **extends** keyword is used when defining a class that builds upon another class. JScript .NET can extend most common-language specification (CLS) compliant classes. A class defined using the **extends** keyword is called a derived class, and the class that is extends is called the base class.

In this example, a new `Student` class is defined, which extends the `Person` class in the previous example. The Student class reuses the `Name` property defined in the base class but defines a new `sayHello` method that overrides the `sayHello` method of the base class.

```
// The Person class is defined in the code above.
class Student extends Person {
   // Override a base-class method.
   function sayHello() {
      return this.name + " is studying for finals.";
   }
}

var mary : Person = new Student;
mary.Name = "Mary";
print(mary.sayHello());
```

The output of this code is:

```
Mary is studying for finals.
```

Redefining a method in a derived class does not change the corresponding method in the base class.

Expando Objects

If you just want a generic object to be expando, use the **Object** constructor.

```
// A JScript Object object, which is expando.
var o = new Object();
o.expando = "This is an expando property.";
print(o.expando);  // Prints This is an expando property.
```

If you want one of your classes to be expando, define the class with the **expando** modifier. Expando members can only be accessed using the index (**[]**) notation; they cannot be accessed using the dot (**.**) notation.

```
// An expando class.
expando class MyExpandoClass {
   function dump() {
      // print all the expando properties
      for (var x : String in this)
         print(x + " = " + this[x]);
   }
}
// Create an instance of the object and add some expando properties.
var e : MyExpandoClass = new MyExpandoClass();
e["answer"] = 42;
e["greeting"] = "hello";
e["new year"] = new Date(2000,0,1);
print("The contents of e are...");
// Display all the expando properites.
e.dump();
```

The output of this program is:

```
The contents of e are...
answer = 42
greeting = hello
new year = Sat Jan 1 00:00:00 PST 2000
```

See Also

Class-based Objects | JScript Objects | JScript Modifiers | Creating Your Own Classes

Prototype-based Objects

Since JScript is an object-oriented programming language, it supports the definition of custom constructor functions and inheritance. Constructor functions (also called constructors) provide the ability to design and implement your own prototype-based objects. Inheritance allows prototype-based objects to share a common set of properties and methods that can be dynamically added or removed.

In many cases, the class-based objects should be used instead of prototype-based objects. Class-based objects can be passed to methods written in other .NET Framework languages. Furthermore, class-based objects provide type safety and produce efficient code.

In This Section

Creating Your Own Objects with Constructor Functions
Explains how to use constructor functions to create objects with properties and methods.

Advanced Object Creation
Illustrates how to use inheritance to add a common set of properties and methods to objects created with a given constructor function.

Related Sections

JScript Objects
Includes links to topics that explain the syntax and uses of the intrinsic JScript objects.

Class-based Objects
Provides a guide to using the JScript class-based object model and describes how to define classes (with methods, fields, and properties), how to define a class that inherits from another class, and how to define expando classes.

JScript Reference
Lists elements that comprise JScript Language Reference and links to topics that explain the details behind the proper use of language elements.

Creating Your Own Objects with Constructor Functions

A powerful feature of JScript is the ability to define constructor functions to create custom prototype-based objects for use in your scripts. To create an instance of a prototype-based object, you first must define a constructor function. This process creates a new object and initializes it (creates properties and assigns an initial value). When completed, the constructor returns a reference to the constructed object. Inside the constructor, the created object is referred to with the **this** statement.

Constructors with Properties

The following example defines a constructor function for `pasta` objects. The **this** statement allows the constructor to initialize the object.

```
// pasta is a constructor that takes four parameters.
function pasta(grain, width, shape, hasEgg) {
   this.grain = grain;    // What grain is it made of?
   this.width = width;    // How many centimeters wide is it?
   this.shape = shape;    // What is the cross-section?
   this.hasEgg = hasEgg;  // Does it have egg yolk as a binder?
}
```

After defining an object constructor, you create instances of the object with the **new** operator. Here the `pasta` constructor is used to create `spaghetti` and `linguine` objects.

```
var spaghetti = new pasta("wheat", 0.2, "circle", true);
var linguine = new pasta("wheat", 0.3, "oval", true);
```

You can dynamically add properties to an instance of an object, but those changes only affect that one instance.

```
// Additional properties for spaghetti. The properties are not added
// to any other pasta objects.
spaghetti.color = "pale straw";
spaghetti.drycook = 7;
spaghetti.freshcook = 0.5;
```

If you want to add an extra property to all instances of the object without modifying the constructor function, you can add the property to the constructor's prototype object. For more information, see Advanced Object Creation.

```
// Additional property for all pasta objects.
pasta.prototype.foodgroup = "carbohydrates";
```

Constructors with Methods

It is possible to include methods (functions) in the definition of an object. One way to do this is to include a property in the constructor function that refers to a function defined elsewhere. Like the constructor functions, these functions also refer to the current object with the **this** statement.

The following example expands on the `pasta` constructor function defined above to include a **toString** method that will be called if the function displays the value of the object. (In general, JScript will use the **toString** method of an object when the object is used in a situation where a string is required. You rarely need to call the **toString** method explicitly.)

```
// pasta is a constructor that takes four parameters.
// The properties are the same as above.
function pasta(grain, width, shape, hasEgg) {
   this.grain = grain;    // What grain is it made of?
   this.width = width;    // How many centimeters wide is it?
   this.shape = shape;    // What is the cross-section?
   this.hasEgg = hasEgg;  // Does it have egg yolk as a binder?
   // Add the toString method (defined below).
   // Note that the function name is not followed with parentheses;
   // this is a reference to the function itself, not a function call.
   this.toString = pastaToString;
}

// The function to display the contents of a pasta object.
function pastaToString() {
   return "Grain: " + this.grain + "\n" +
          "Width: " + this.width + " cm\n" +
          "Shape: " + this.shape + "\n" +
          "Egg?:  " + Boolean(this.hasEgg);
}

var spaghetti = new pasta("wheat", 0.2, "circle", true);
// Call the method explicitly.
print(spaghetti.toString());
// The print statement takes a string as input, so it
//  uses the toString() method to display the properties
// of the spaghetti object.
print(spaghetti);
```

This displays the following output.

```
Grain: wheat
Width: 0.2 cm
Shape: circle
Egg?:  true
Grain: wheat
Width: 0.2 cm
Shape: circle
Egg?:  true
```

See Also

Prototype-based Objects | JScript Objects

Advanced Object Creation

JScript supports inheritance with custom prototype-based objects. Through inheritance, prototype-based objects can share a common set of properties and methods that can be dynamically added or removed. Moreover, individual objects can override the default behavior.

To create an instance of prototype-based object, you first must define a constructor function. For more information, see Creating Your Own Objects with Constructor Functions. Once this constructor is written, you can use properties of the **prototype** object (which is itself a property of every constructor) to create inherited properties and shared methods. The constructor provides the instance-specific information to an object, while the **prototype** object provides the object-specific information and methods to the object.

> **Note** To affect all instances of the object, there must be a change to the **prototype** object of the constructor. Changing the **prototype** property of one instance of an object has no effect on the other instances of the same object.

Since the properties and methods of the **prototype** object are copied by reference into each instance of an object, all instances have access to the same information. You can change the value of a prototype property in one instance to override the default, but the change will only affect that one instance. Here is an example that uses the custom constructor, `Circle`. The **this** statement enables the method to access members of the object.

```
// Define the constructor and add instance specific information.
function Circle (radius) {
    this.r = radius;  // The radius of the circle.
}
// Add a property the Circle prototype.
Circle.prototype.pi = Math.PI;
function ACirclesArea () {
   // The formula for the area of a circle is pi*r^2.
   return this.pi * this.r * this.r;
}
// Add a method the Circle prototype.
Circle.prototype.area = ACirclesArea;
// This is how you would invoke the area function on a Circle object.
var ACircle = new Circle(2);
var a = ACircle.area();
```

Using this principle, you can define additional properties for existing constructor functions (which all have prototype objects). This will work only when fast mode is turned off. For more information, see /fast.

For example, if you want to remove leading and trailing spaces from strings (similar to the **Trim** function in Visual Basic), you can create your own method on the **String** prototype object, and all strings in your script will automatically inherit the method. This example uses a regular expression literal to remove the spaces. For more information, see Regular Expression Object.

```
// Add a function called trim as a method of the prototype
// object of the String constructor.
String.prototype.trim = function() {
   // Use a regular expression to replace leading and trailing
   // spaces with the empty string
   return this.replace(/(^\s*)|(\s*$)/g, "");
}

// A string with spaces in it
var s = "    leading and trailing spaces    ";
print(s + " (" + s.length + ")");

// Remove the leading and trailing spaces
s = s.trim();
print(s + " (" + s.length + ")");
```

After compiling this program with the **/fast-** flag, the output of this program is:

```
    leading and trailing spaces     (35)
leading and trailing spaces (27)
```

See Also

Prototype-based Objects | JScript Objects

JScript Modifiers

JScript modifiers change the behavior and visibility of classes, interfaces, or members of classes or interfaces. You may use modifiers when defining classes and interfaces, but they are usually not required.

Visibility Modifiers

Visibility modifiers restrict how outside code accesses classes, interfaces, and their members. You can use restrictions to encourage good object-oriented programming practices by preventing calls to specialized internal methods and fields.

By default, any code that can access a class can access any of the members of that class. Using the visibility modifiers, you can selectively prevent outside code from accessing particular class members, allow only classes from the same package to access members, or allow only derived classes to access class members.

Visibility modifiers cannot be applied to global functions or variables. The only visibility modifiers that can be used together are **protected** and **internal**.

Visibility Modifier	**Valid for**	**Meaning**
public	class, class member, interface, or interface member, enumerations	Member is visible to any code that has access to the class with no restrictions on visibility. By default in JScript .NET, classes, interfaces, and their members are public.
private	class member	Member is visible only within the class in which it is declared. It is not visible to derived classes. Code outside the current class cannot access private members.
protected	class member	Member is visible only within the class in which it is declared and to any derived classes of that class. The protected modifier cannot be used for classes at package scope, but it can be used for nested classes.
internal	class, class member, enumeration	Class, class member, or enumeration is visible everywhere within the package in which it is declared. It is not visible outside the package.

Inheritance Modifiers

Inheritance modifiers control how methods and properties from derived classes override methods and properties in a base class. By using this control, you can manage whether methods from derived classes will override a class you create.

By default, methods from a derived class will override base class methods unless the version-safe **hide** attribute is used in the derived class. This attribute prevents overriding. Using inheritance modifiers enables you to control whether particular methods are overridden always or never.

In some situations, you may need to ensure that a base class method is not overridden. For example, if you define a class in a package, you can use the **final** modifier to ensure that derived classes will not change the methods and properties of the class.

On the other hand, you may want to require your class to have certain methods overridden. For example, you can create a class that provides some basic functionality but use the **abstract** modifier for some methods. The implementations of the abstract methods are up to the writer of the derived class.

Version-safe modifiers, which also manage overriding, manage it from the derived-class side rather than the base-class side. Version-safe modifiers have an effect only if the base-class method they are overriding does not have inheritance modifiers.

You cannot combine two inheritance modifiers or combine an inheritance modifier with the **static** modifier.

Inheritance Modifier	**Valid for**	**Meaning**
abstract	Class, method, or property	For methods or properties, this modifier indicates that the member does not have an implementation. For classes, this modifier indicates that there are one or more unimplemented methods. An abstract class or a class that contains an abstract member cannot be instantiated using the **new** keyword, but it can be used as a base class.

final	Class, method, or property	For classes that cannot be extended or methods that cannot be overridden. Using **final** prevents derived classes from changing the behavior of the clas s by overriding important functions. Methods with the **final** modifier can be hidden or overloaded, but not overridden.

Version-Safe Modifiers

Version-safe modifiers control the methods from a derived class that override methods in a base class. By using this control, you can manage whether a class you create will override methods in the base class.

By default, methods from a derived class will override methods in a base class, although inheritance modifiers in the definition of the derived class can prevent overriding as well. Using version-safe modifiers enables you to control whether particular methods are overridden or not.

In some situations, you may need to ensure that base-class methods are not overridden. For example, you may extend a class to change the behavior of the base-class methods. If you do not want those methods to be overridden in the base class, you can use the **hide** modifier for your method declarations.

On the other hand, you may want to override certain base-class methods. For example, you may want to change the methods of a class without modifying the class. By extending the class and using the **override** modifier for your method declarations, you can cause the new methods to override the base class.

Successful use of version-safe modifiers depends on whether or not the declaration of the base class methods used inheritance modifiers. Base-class methods marked with the **final** modifier cannot be overridden, and base-class methods marked with the **abstract** modifier cannot be hidden unless an explicit implementation for the abstract base-class method is given.

You cannot combine two version-safe modifiers or combine a version-safe modifier with the **static** modifier. When running in version-safe mode, only one version-safe modifier may be used for each method that overrides a base-class method.

Version-Safe Modi fier	Valid for	Meaning
hide	Method or property	Member does not override a member with the same name in the base class.
override	Method or property	By default, members override members with the same name in the base cla ss.

expando Modifier

The **expando** modifier causes a class-based object to behave as if it were a JScript object. Methods and properties can be dynamically added to an expando object. For more information, see Prototype-based Objects.

You can use the **expando** modifier independently of the other modifiers.

Modifier	Valid for	Meaning
expando	Class or method	For a class, the class is given a default, indexed property that is capable of st oring and retrieving dynamic properties (expandos). For a method, indicates that it is a constructor for an expando object.

static Modifier

The **static** modifier signifies that a member of a class belongs to the class itself rather than to instances of the class. Consequently, class-specific data and methods may not be associated with any particular instance.

You cannot combine the **static** modifier with any of the version-safe or inheritance modifiers.

Modifier	Valid for	Meaning
static	Method, property, field, or class	For methods, indicates that it can be called without an instance of the class. For properties and fields, designates that one copy is shared by all instances . The **static** modifier should not be confused with the **static** statement, whic h denotes code that initializes the class.

See Also

Modifiers | class Statement | interface Statement | function Statement | function get Statement | function set Statement | var Statement | const Statement | static Statement

JScript Operators

JScript has a full range of operators, including computational, logical, bitwise, assignment, as well as some miscellaneous operators. Operators combine simple expressions to comprise more complex expressions.

In This Section

Operator Summary
Provides tables of the JScript operators that are grouped by type of operator.

Operator Precedence
Provides a table that lists operators and corresponding precedence and an example of how operator precedence works.

Coercion By Bitwise Operators
Describes the rules that govern the coercion of the operands of bitwise operators. Coercion is required so the binary formats of the operands are compatible with each other and the bitwise operator.

Related Sections

JScript Assignments and Equality
Explains how to use the assignment, equality, and strict equality operators.

Coercion in JScript
Explains the concept of coercion, how to use it, and the limitations of coercion.

Operators
Lists links to reference topics for the operators in JScript.

Operator Summary

The following tables list JScript operators. Each name in the description column links to a corresponding topic that explains the proper syntax and use.

Computational Operators

Description	Symbol
Addition	+
Decrement	--
Division	/
Increment	++
Modulus arithmetic	%
Multiplication	*
Subtraction	-
Unary negation	-

All computational operators perform calculations with numeric data. The addition operator also performs string concatenation when either operand is a string.

Logical Operators

Description	Symbol
Equality	==
Greater than or equal to	>=
Greater than	>
Identity	===
In	in
Inequality	!=
Less than or equal to	<=
Less than	<
Logical AND	&&
Logical NOT	!
Logical Or	||
Non-identity	!==

A logical operator returns a **Boolean** value. Depending on the operator, the value may represent the result of the comparison, test, or combination.

Bitwise Operators

Description	Symbol
Bitwise AND	&
Bitwise Left Shift	<<
Bitwise NOT	~
Bitwise Or	|
Bitwise Right Shift	>>
Bitwise XOR	^
Unsigned Right Shift	>>>

Bitwise operators operate on the binary representation of the operands. If operands are not compatible with each other, they will be coerced to the appropriate type. For more information, see Coercion By Bitwise Operators.

Assignment Operators

Description	Symbol
Assignment	=
Compound Addition Assignment	+=

Compound Bitwise AND Assignment	&=
Compound Bitwise Or Assignment	|=
Compound Bitwise XOR Assignment	^=
Compound Division Assignment	/=
Compound Left Shift Assignment	<<=
Compound Modulus Assignment	%=
Compound Multiplication Assignment	*=
Compound Right Shift Assignment	>>=
Compound Subtraction Assignment	-=
Compound Unsigned Right Shift Assignment	>>>=

All assignment operators return the value that is assigned to the left operand.

Miscellaneous Operators

Description	Symbol
Comma	,
Conditional (Ternary)	?:
Delete	delete
Instanceof	instanceof
New	new
Typeof	typeof
Void	void

See Also

JScript Operators | Operator Precedence

Operator Precedence

Operator precedence is a set of rules in JScript that controls the order in which the compiler performs operations when evaluating an expression. Operations with a higher precedence are performed before those with a lower one. For example, multiplication is performed before addition.

The following table lists the JScript operators, ordered from highest to lowest precedence.

Precedence	**Evaluation Order**	**Operator**	**Description**
15	left to right	., [], ()	Field access, array indexing, function calls, and expression grouping
14	right to left	++, --, -, ~, !, delete, new, typeof, void	Unary operators, return data type, object creation, undefined values
13	left to right	*, /, %	Multiplication, division, modulo division
12	left to right	+, -	Addition and string concatenation, subtraction
11	left to right	<<, >>, >>>	Bit shifting
10	left to right	<, <=, >, >=, instanceof	Less than, less than or equal, greater than, greater than or equal, instanceof
9	left to right	==, !=, ===, !==	Equality, inequality, strict equality, and strict inequality
8	left to right	&	Bitwise AND
7	left to right	^	Bitwise XOR
6	left to right	|	Bitwise OR
5	left to right	&&	Logical AND
4	left to right	||	Logical OR
3	right to left	?:	Conditional
2	right to left	=, *OP*=	Assignment, compound assignment
1	left to right	, (comma)	Multiple evaluation

Parentheses in an expression alter the order of evaluation determined by operator precedence. This means an expression within parentheses is fully evaluated before its value is used in the remainder of the expression.

For example:

```
z = 78 * (96 - 3 + 45)
```

There are five operators in the preceding expression: =, *, (), -, and +. According to the rules of operator precedence, they are evaluated in the following order: (), -, +, *, =.

1. 5Evaluation of the expression within the parentheses occurs first. Within the parentheses, there is an addition operator and a subtraction operator. The operators both have the same precedence, and they are evaluated from left to right. The number 3 is subtracted from 96 first, resulting in 93. Then the number 45 is added to 93, resulting in a value of 139.
2. Multiplication occurs next. The number 78 is multiplied by the number 139, resulting in a value of 10764.
3. Assignment occurs last. The number 10764 is assigned to z.

See Also

JScript Operators | Operator Summary

Coercion By Bitwise Operators

The bitwise operators in JScript .NET are fully compatible with the bitwise operators in previous versions of JScript. In addition, the JScript .NET operators can also be used on the new numeric data types. The behavior of the bitwise operators depends on the binary representation of the data, so it is important to understand how the operators coerce the types of the data.

Three types of arguments can be passed to bitwise operators: early-bound variables, late-bound variables, and literal data. Early-bound variables are variables defined with an explicit type annotation. Late-bound variables are variables of type **Object** that contain numeric data.

Bitwise AND (&), OR (|) and XOR (^) Operators

If either operand is late-bound or if both operands are literals, then both operands are coerced to **int** (**System.Int32**), the operation is performed, and the return value is an **int**.

If both operands are early-bound or if one operand is literal and the other is early-bound, more steps are performed. Both operands are coerced to a type determined by two conditions:

- If neither operand is integral, both operands are coerced to **int**.
- If only one operand is integral, the non-integral operand is either coerced to the integral type or coerced to **int** whichever type is longer.
- If one operand is longer, then the type to which the operand is coerced has the same length as the longer operand.
- If either operand is unsigned, then the type to which the operand is coerced is unsigned. Otherwise, the type coerced is signed.

The operands are then coerced to the appropriate type, the bitwise operation is performed, and the result is returned. The data type of the result is the same as the type of the coerced operands.

When using an integral literal with a bitwise operator and an early-bound variable, the literal will be interpreted as either an **int**, **long**, **ulong**, or **double**, depending on which is the smallest type that can represent the number. Literal **decimal** values are coerced to **double**. The data type of the literal may undergo further coercion according to the rules described above.

Bitwise NOT (~) Operator

If the operand is late-bound, floating-point early-bound, or a literal, it is coerced to **int** (**System.Int32**), the NOT operation is performed, and the return value is an **int**.

If the operand is early-bound integral data type, the NOT operation is performed, and the return type is the same as the type of the operand.

Bitwise Left Shift (<<), Right Shift (>>) Operators

If the left operand is late-bound, floating-point early-bound, or a literal, it is coerced to an **int** (**System.Int32**). Otherwise, the left operand is early-bound integral data type and no coercion is performed. The right operand is always coerced to an integral data type. The shift operation is then performed on the coerced values and the result returned has the same type as the left operand (if early-bound) or as type **int**.

Unsigned Right Shift (>>>) Operators

If the left operand is late-bound, floating-point early-bound, or a literal, it is coerced to a **uint** (**System.UInt32**). Otherwise, the left operand is early-bound integral data type and it is coerced to an unsigned type of the same size. For example, an **int** would be coerced to a **uint**. The right operand is always coerced to an integral data type. The shift operation is then performed on the coerced values and the result returned has the same type as the coerced left operand (if early-bound) or as type **uint**.

The result of the unsigned right shift is always small enough to be stored in the signed version of the return type without overflow.

See Also

JScript Operators | Operator Precedence | Type Conversion | Coercion in JScript | Numeric Data

JScript Functions

JScript functions can perform actions, return values, or do both. For example, a function could display the current time and return a string that represents the time. Functions are also called global methods.

Functions combine several operations under one name, which makes code streamlined and reusable. You can write a set of statements, name it, and then execute the entire set by calling its name and passing to it the necessary information.

To pass information to a function, enclose the information in parentheses after the name of the function. Pieces of information that are passed to a function are called arguments or parameters. Some functions do not take arguments while others take one or more arguments. In some functions, the number of arguments depends on how you are using the function.

JScript supports two kinds of functions, those that are built into the language and those that you create.

In This Section

Type Annotation
: Describes the concept of type annotation and how to use it in a function definition to control the input and output data types.

User Defined JScript Functions
: Illustrates how to define new functions in JScript and how to use them.

Recursion
: Explains the concept of recursion and illustrates how to write recursive functions.

Related Sections

JScript Operators
: Lists the computational, logical, bitwise, assignment, and miscellaneous operators and provides links to information that explains how to use them efficiently.

JScript Data Types
: Includes links to topics that explain how to use primitive data types, reference data types, and .NET Framework data types in JScript.

Coercion in JScript
: Explains the concept of coercion, how to use it, and the limitations of coercion.

function Statement
: Describes the syntax for declaring functions.

Type Annotation

Type annotation in a function specifies a required type for function arguments, a required type for returned data, or a required type for both. If you do not type annotate the parameters of a function, the parameters will be of type **Object**. Likewise, if the return type for a function is not specified, the compiler will infer the appropriate return type.

Using type annotation for function parameters helps ensure that a function will only accept data that it can process. Declaring a return type explicitly for a function improves code readability since the type of data that the function will return is immediately clear.

The following example illustrates the use of type annotations for both the parameters and return type of the function.

```
// Declare a function that takes an int and returns a String.
function Ordinal(num : int) : String{
   switch(num % 10) {
   case 1: return num + "st";
   case 2: return num + "nd";
   case 3: return num + "rd";
   default: return num + "th";
   }
}

// Test the function.
print(Ordinal(42));
print(Ordinal(1));
```

The output of this program is:

```
42nd
1st
```

A type mismatch error would be generated if an argument were passed to the `Ordinal` function that could not be coerced to an integer. For example, `Ordinal(3.14159)` would fail.

See Also

JScript Functions | function Statement | Data Types

User Defined JScript Functions

Although JScript includes many built-in functions, you can create your own functions. A function definition consists of a function statement and a block of JScript statements.

The `checkTriplet` function in the following example takes the lengths of the sides of a triangle as its arguments. It calculates whether the triangle is a right triangle by checking whether the three numbers constitute a Pythagorean triplet (the square of the length of the hypotenuse of a right triangle is equal to the sum of the squares of the lengths of the other two sides). The `checkTriplet` function calls one of two other functions to make the actual test.

Notice the use of a very small number (`epsilon`) as a testing variable in the floating-point version of the test. Because of uncertainties and round-off errors in floating-point calculations, it is not practical to make a direct test of whether the three numbers constitute a Pythagorean triplet unless all three values in question are known to be integers. Because a direct test is more accurate, the code in this example determines whether it is appropriate and, if it is, uses it.

Type annotation is not used when defining these functions. For this application, it is useful for the `checkTriplet` function to take both integer and floating-point data types.

```
const epsilon = 0.00000000001; // Some very small number to test against.

// Type annotate the function parameters and return type.
function integerCheck(a : int, b : int, c : int) : boolean {
   // The test function for integers.
   // Return true if a Pythagorean triplet.
   return ( ((a*a) + (b*b)) == (c*c) );
} // End of the integer checking function.

function floatCheck(a : double, b : double, c : double) : boolean {
   // The test function for floating-point numbers.
   // delta should be zero for a Pythagorean triplet.
   var delta = Math.abs( ((a*a) + (b*b) - (c*c)) * 100 / (c*c));
   // Return true if a Pythagorean triplet (if delta is small enough).
   return (delta < epsilon);
} // End of the floating-poing check function.

// Type annotation is not used for parameters here. This allows
// the function to accept both integer and floating-point values
// without coercing either type.
function checkTriplet(a, b, c) : boolean {
   // The main triplet checker function.
   // First, move the longest side to position c.
   var d = 0; // Create a temporary variable for swapping values
   if (b > c) { // Swap b and c.
      d = c;
      c = b;
      b = d;
   }
   if (a > c) { // Swap a and c.
      d = c;
      c = a;
      a = d;
   }

   // Test all 3 values. Are they integers?
   if ((int(a) == a) && (int(b) == b) && (int(c) == c)) { // If so, use the precise check.
      return integerCheck(a, b, c);
   } else { // If not, get as close as is reasonably possible.
      return floatCheck(a, b, c);
   }
} // End of the triplet check function.

// Test the function with several triplets and print the results.
// Call with a Pythagorean triplet of integers.
print(checkTriplet(3,4,5));
// Call with a Pythagorean triplet of floating-point numbers.
print(checkTriplet(5.0,Math.sqrt(50.0),5.0));
```

```
// Call with three integers that do not form a Pythagorean triplet.
print(checkTriplet(5,5,5));
```

The output of this program is:

```
true
true
false
```

See Also

JScript Functions | JScript Data Types | function Statement

Recursion

Recursion is an important programming technique that causes a function to call itself. One example is the calculation of factorials. The factorial of 0 is defined specifically to be 1. The factorial of *n*, an integer greater than 0, is the product of all the integers in the range from 1 to *n*.

The following paragraph is a function, defined in words, that calculates a factorial.

> "If the number is less than zero, reject it. If it is not an integer, reject it. If the number is zero, its factorial is one. If the number is larger than zero, multiply it by the factorial of the next lesser number."

To calculate the factorial of any number that is larger than zero, you must calculate the factorial of at least one other number. The function must call itself for the next smaller number before it can execute on the current number. This is an example of recursion.

Recursion and iteration (looping) are strongly related — a function can return the same results either with recursion or iteration. Usually a particular computation will lend itself to one technique or the other, and you simply choose the most natural or preferable approach.

Despite the usefulness of recursion, you can easily create a recursive function that never returns a result and cannot reach an endpoint. Such a recursion causes the computer to execute an *infinite* loop. Here is an example: omit the first rule (the one about negative numbers) from the verbal description of calculating a factorial, and try to calculate the factorial of any negative number. This fails because to calculate the factorial of, for example, -24, you must calculate the factorial of -25. In order to calculate the factorial of -25, you must first calculate the factorial of -26, and so on. Obviously, this never reaches a conclusion.

Another problem that can occur with recursion is a recursive function can use all the available resources (such as system memory, stack space, and so on). Each time a recursive function calls itself (or calls another function that calls the original function), it uses some resources. These resources are freed when the recursive function exits, but a function that has too many levels of recursion may use all the available resources. When this happens, an exception is thrown.

Thus, it is important to design recursive functions with care. If you suspect any chance of an excessive (or infinite) recursion, design the function to count the number of times it calls itself and set a limit on the number of calls. If the function calls itself more times than the threshold, the function can quit automatically. The optimum maximum number of iterations depends on the recursive function.

Here is the factorial function again, this time written in JScript code. Type annotation is used so the function accepts only integers. If an invalid number is passed in (that is, a number less than zero), the throw statement generates an error. Otherwise, a recursive function is used to calculate the factorial. The recursive function takes two arguments, one for the factorial argument and one for the counter that keeps track of the current recursion level. If the counter does not reach the maximum recursion level, the factorial of the original number is returned.

```
function factorialWork(aNumber : int, recursNumber : int ) : double {
   // recursNumber keeps track of the number of iterations so far.
   if (aNumber == 0) {  // If the number is 0, its factorial is 1.
      return 1.;
   } else {
      if(recursNumber > 100) {
         throw("Too many levels of recursion.");
      } else {  // Otherwise, recurse again.
         return (aNumber * factorialWork(aNumber - 1, recursNumber + 1));
      }
   }
}

function factorial(aNumber : int) : double {
   // Use type annotation to only accept numbers coercible to integers.
   // double is used for the return type to allow very large numbers to be returned.
   if(aNumber < 0) {
      throw("Cannot take the factorial of a negative number.");
   } else {  // Call the recursive function.
      return  factorialWork(aNumber, 0);
   }
}

// Call the factorial function for two values.
print(factorial(5));
print(factorial(80));
```

The output of this program is:

```
120
7.156945704626378e+118
```

See Also

JScript Functions | Type Annotation

Coercion in JScript

JScript can perform operations on values of different types without the compiler raising an exception. Instead, the JScript compiler automatically changes (coerces) one of the data types to that of the other before performing the operation. Other languages have much stricter rules governing coercion.

The compiler allows all coercions unless it can prove that the coercion will always fail. Any coercion that *may* fail generates a warning at compile time, and many produce a runtime error if the coercion fails. For example:

Operation	Result
Add a number and a string	The number is coerced into a string.
Add a Boolean and a string	The Boolean is coerced into a string.
Add a number and a Boolean	The Boolean is coerced into a number.

Consider the following example.

```
var x = 2000;      // A number.
var y = "Hello";   // A string.
x = x + y;         // the number is coerced into a string.
print(x);          // Outputs 2000Hello.
```

To explicitly convert a string to an integer, you can use the **parseInt** Method. For more information, see the parseInt Method. To explicitly convert a string to a number, you can use the **parseFloat** method. For more information, see the parseFloat Method. Notice that strings are automatically converted to equivalent numbers for comparison purposes but are left as strings for addition (concatenation).

Since JScript .NET is also a strongly typed language, another coercion mechanism is available. The new mechanism uses the target type name as if it were a function that accepts the expression to convert as an argument. This works for all JScript primitive types, JScript reference types, and .NET Framework types.

For example, the following code converts an integer value to a Boolean:

```
var i : int = 23;
var b : Boolean;
b = i;
b = Boolean(i);
```

Because the value of `i` is a value other than zero, `b` is **true**.

The new coercion mechanism also works with many user-defined types. However, some coercions to or from user-defined types may not work because JScript may misinterpret the user's intent when converting dissimilar types. This is particularly true when the type that is converted from is comprised of several values. For example, the following code creates two classes (types). One contains a single variable `i` that is an integer. The other contains three variables (`s`, `f`, and `d`), each of a different type. In the final statement, it is impossible for JScript to determine how to convert a variable of the first type to the second type.

```
class myClass {
   var i : int = 42;
}
class yourClass {
   var s : String = "Hello";
   var f : float = 3.142;
   var d : Date = new Date();
}
// Define a variable of each user-defined type.
var mine : myClass = new myClass();
var yours : yourClass;

// This fails because there is no obvious way to convert
// from myClass to yourClass
yours = yourClass(mine);
```

See Also

JScript Functions | Type Conversion | Coercion By Bitwise Operators

JScript Conditional Structures

Statements in JScript are normally executed sequentially in the order in which they appear in the script. This is called sequential execution and is the default direction of program flow.

An alternative to sequential execution transfers the program flow to another part of a script as a result of conditions that a script encounters. That is, instead of executing the next statement in the sequence, a statement at another location is executed. Another alternative is called *iteration*, which involves repeating the same sequence of statements multiple times. Iteration is frequently achieved using loops.

In This Section

Conditional Statements
Describes the concept of condition statements and explains several common types of expressions used as condition statements.

Control Structures
Describes the two types of control structures provided by JScript, selection control structure and repetition control structure, and the uses of both.

Conditional Statement Use
Illustrates how to use some conditional statements by providing examples of the **if** statement and **do**...**while** loop.

Conditional Operator
Describes how to use the conditional operator and its relationship to the **if**...**else** statement.

Loops in JScript
Introduces the concept of loops in JScript and lists links to information that explains how to use the loop constructs within JScript code.

Related Sections

JScript Data Types
Includes links to topics that explain how to use primitive data types, reference data types, and .NET Framework data types in JScript.

JScript Reference
Lists elements that comprise JScript Language Reference and links to topics that explain the details behind the proper use of language elements.

Conditional Statements

Instructions in JScript code execute sequentially by default. It might be useful, however, to alter the logical sequence and transfer control to a non-sequential part of the code depending on specific conditions. A control structure transfers program control to one of two places depending on whether a conditional statement tests true or false. Any expression coercible to a Boolean value can be used as a conditional statement. Some common condition statements are mentioned here.

Equality and Strict Equality

The equality operator (**==**) in a condition statement checks whether the two arguments passed to it have the same value, performing type conversion if necessary to make a comparison. The strict equality operator (**===**) compares both the value and type of two expressions; true is returned only if the value and data type are the same for the two operands. Note that the strict equality operator does not distinguish between the different numeric data types.

The following JScript code combines an equality operator with an **if** statement that uses it. For more information, see Control Structures.

```
function is2000(x) : String {
   // Check if the value of x can be converted to 2000.
   if (x == 2000) {
      // Check is the value of x is strictly equal to 2000.
      if(x === 2000)
         print("The argument is number 2000.");
      else
         print("The argument can be converted to 2000.");
   } else {
      print("The argument is not 2000.");
   }
}
// Check several values to see if they are 2000.
print("Check the number 2000.");
is2000(2000);
print("Check the string \"2000\".");
is2000("2000")
print("Check the number 2001.");
is2000(2001);
```

Following is the output of this code.

```
Check the number 2000.
The argument is number 2000.
Check the string "2000".
The argument can be converted to 2000.
Check the number 2001.
The argument is not 2000.
```

Inequality and Strict Inequality

The inequality operator (**!=**) returns the opposite result of the equality operator. If the operands have the same value, the inequality operator returns **false**; otherwise it returns **true**. Likewise, the strict inequality operator (**!==**) returns the opposite result of the strict equality operator.

Consider the following JScript code sample in which the inequality operator is used to control a **while** loop. For more information, see Control Structures.

```
var counter = 1;
// Loop over the print statement while counter is not equal to 5.
while (counter != 5) {
   print(counter++);
}
```

The following is the output of this code.

```
1
2
3
4
```

Comparison

The equality and inequality operators are useful if a piece of data has a particular value. However, in some situations code may need to check if a value is within a particular range. The relational operators, less than (<), greater than (>), less than or equal to (<=), and greater than or equal to (>=), are appropriate for these cases.

```
if(tempInCelsius < 0)
   print("Water is frozen.")
else if(tempInCelsius > 100)
   print("Water is vapor.");
else
   print("Water is liquid.);
```

Short-Circuit

If you want to test several conditions together and you know that one is more likely to pass or fail than the others, you can use a feature called short circuit evaluation to speed the execution of your script and avoid side effects that might cause errors. When JScript evaluates a logical expression, it only evaluates as many sub-expressions as required to get a result.

The logical AND (**&&**) operator evaluates the left expression passed to it first. If that expression converts to **false**, then the result of the logical AND operator cannot be **true** regardless of the value of the right expression. Therefore, the right expression is not evaluated.

For example, in the expression `((x == 123) && (y == 42))`, JScript first checks if `x` is 123. If it is not, the test for `y` is never made, and JScript returns the value **false**.

Similarly, the logical OR operator (**||**) evaluates the left expression first and if it converts to **true**, the right expression is not evaluated.

Short-circuiting is useful when the conditions to be tested involve the execution of function calls or other complex expressions. To make a script run most efficiently, place the conditions most likely to be **true** first for the logical OR operator. For the logical AND operator, place the conditions most likely to be **false** first.

An example of the a benefit of designing your script in this manner is that **runsecond()** will not be executed in the following example if the return value of **runfirst()** converts to **false**.

```
if ((runfirst() == 0) || (runsecond() == 0)) {
   // some code
}
```

Another example of the a benefit of designing your script in this manner is that **runsecond()** will not be executed in the following example if the return value of **runfirst()** converts to **false**.

```
if ((x == 0) && (y/x == 5)) {
   // some code
}
```

Other

Any expression that can be converted to a Boolean value can be used as a condition statement. For example, you could use an expression such as:

```
if (x = y + z) // This may not do what you expect - see below!
```

Note that the above code does *not* check if `x` is equal to `y + z`, since the syntax uses only a single equal sign (assignment). Instead, the code above assigns the value of `y + z` to the variable `x`, and then checks if the result of the entire expression (the value of `x`) can be converted to the value **true**. To check if `x` is equal to `y + z`, use the following code.

```
if (x == y + z) // This is different from the code above!
```

See Also

JScript Conditional Structures | JScript Data Types | JScript Reference | Boolean Data | Operators

Control Structures

For all control structures except the **switch** statement, transfer of program control is based upon a decision, the result of which is a truth statement (returning a Boolean **true** or **false**). You create an expression and then test whether its result is **true**. There are two main kinds of program control structures.

Selection Control Structure

The selection structure specifies alternate courses of program flow, creating a junction in your program (like a fork in a road). There are four selection structures available in JScript.

- the single-selection structure (**if**),
- the double-selection structure (**if...else**),
- the multiple-selection structure (**switch**),
- the inline conditional operator **?:**

Repetition Control Structure

The repetition structure specifies the repetition of an action while some condition remains true. When the conditions of the control statement have been met (usually after some specific number of iterations), control passes to the next statement beyond the repetition structure. There are four repetition structures available in JScript.

- the expression is tested at the top of the loop (**while**),
- the expression is tested at the bottom of the loop (**do...while**),
- operate on an object's properties or an array's elements (**for...in**),
- counter controlled repetition (**for**).

Combination Control Structure

Complex scripts nest and stack selection and repetition control structures.

Exception handling, which provides another way to control program flow, is not covered here. For more information, see try...catch...finally Statement.

See Also

JScript Conditional Structures | JScript Reference

Conditional Statement Use

JScript supports **if** and **if...else** conditional statements. An **if** statement tests a condition. It executes the relevant JScript code if the condition evaluates as true. An **if...else** statement tests a condition and executes one of two blocks of code depending on the result of the condition statement. The simplest form of an **if** statement can be written on one line, but multiline **if** and **if...else** statements are more common.

The following examples demonstrate syntaxes you can use with **if** and **if...else** statements. The first example shows the simplest kind of Boolean test. If (and only if) the item between the parentheses evaluates to (or can be coerced to) **true**, the statement or block of statements after the **if** is executed.

In the following example, the `registerUser` function is called if the value of `newUser` converts to true.

```
if (newUser)
   registerUser();
```

In this example, the test fails unless both conditions are true.

```
if (rind.color == "deep yellow " && rind.texture == "wrinkled") {
   theResponse = ("Is it a Crenshaw melon?");
}
```

In this example, the code in the body of the **do**...**while** loop is executed until the variable `quit` is **true**.

```
var quit;
do {
   // ...
   quit = getResponse()
}
while (!quit)
```

See Also

JScript Conditional Structures | JScript Reference | if...else Statement

Conditional Operator

JScript supports an implicit conditional form, the conditional operator. It takes three operands. A question mark separates the first two operands, and a colon separates the second and third operands. The first operand is a conditional expression. The second operand is a statement that is executed if the conditional expression evaluates to true. The third operand is executed if the conditional is false. For more information, see Conditional (Ternary) Operator (?:). The conditional operator is similar to the **if...else** statement.

In this example, the conditional operator determines if an hour in 24-hour time is before noon ("AM") or after noon ("PM").

```
var hours : String = (the24Hour >= 12) ? " PM" : " AM";
```

In general, an **if ... then ... else** structure is appropriate when choosing between statements to be executed, whereas the conditional operator (?:) is appropriate when choosing between two expressions. Do not try to use the conditional operator to choose between more than two alternatives or to execute blocks of statements. In those cases, use the **if...then...else** construct.

See Also

JScript Conditional Structures | JScript Reference

Loops in JScript

JScript includes several ways to execute a statement or block of statements repeatedly. In general, repetitive execution is called *looping* or *iteration*. An iteration is simply a single execution of a loop. It is typically controlled by a test of a variable, where the value is changed each time the loop is executed. JScript supports four types of loops: **for** loops, **for...in** loops, **while** loops, and **do...while** loops.

In This Section

for Loops
: Discusses how JScript uses **for** loops and provides some practical examples.

for...in Loops
: Describes the concept of **for**...**in** loops and explains how to use them in JScript.

while Loops
: Discusses the two types of **while** loops and explains how they differs from **for** loops.

break and continue Statements
: Describes how to use the **break** and **continue** statements to override the behavior of a loop.

Related Sections

JScript Conditional Structures
: Describes how JScript normally handles program flow and provides links to information that explains how to regulate the flow of a program's execution.

JScript Reference
: Lists elements that comprise JScript Language Reference and links to topics that explain the details behind the proper use of language elements.

for Loops

The **for** statement specifies a counter variable, a test condition, and an action that updates the counter. Before each iteration of the loop, the condition is tested. If the test is successful, the code inside the loop is executed. If the test is unsuccessful, the code inside the loop is not executed, and the program continues on the first line of code immediately following the loop. After the loop is executed, the counter variable is updated before the next iteration begins.

If the condition for looping is never met, the loop is never executed. If the test condition is always met, an infinite loop results. While the former may be desirable in certain cases, the latter rarely is, so be cautious when writing your loop conditions. In this example, the **for** loop is used to initialize the elements of an array with the sum of the previous elements.

```
var sum = new Array(10); // Creates an array with 10 elements
sum[0] = 0;              // Define the first element of the array.
var iCount;

// Counts from 0 through one less than the array length.
for(iCount = 0; iCount < sum.length; iCount++) {
   // Skip the assignment if iCount is 0, which avoids
   // the error of reading the -1 element of the array.
   if(iCount!=0)
      // Add the iCount to the previous array element,
      // and assign to the current array element.
      sum[iCount] = sum[iCount-1] + iCount;
   // Print the current array element.
   print(iCount + ": " + sum[iCount]);
}
```

The output of this program is:

```
0: 0
1: 1
2: 3
3: 6
4: 10
5: 15
6: 21
7: 28
8: 36
9: 45
```

There are two loops in the next example. The code block of the first loop is never executed, while the second loop is an infinite loop.

```
var iCount;
var sum = 0;
for(iCount = 0; iCount > 10; iCount++) {
   // The code in this block is never executed, since iCount is
   // initially less than 10, but the condition checks if iCount
   // is greater than 10.
   sum += iCount;
}
// This is an infinite loop, since iCount is always greater than 0.
for(iCount = 0; iCount >= 0; iCount++) {
   sum += iCount;
}
```

See Also

Loops in JScript | JScript Conditional Structures | JScript Reference | for Statement

for...in Loops

JScript provides a special kind of loop for looping through all the user-defined properties of an object, all the elements of an array, or all the elements of a collection. The loop counter in a **for...in** loop is a string or object rather than a number. It contains the name of the current property, the index of the current array element, or the current element in the collection.

The following code illustrates the use of the **for...in** construct.

```
// Create an object with some properties.
var prop, myObject = new Object();
myObject.name = "James";
myObject.age = 22;
myObject.phone = "555 1234";
// Loop through all the properties in the object.
for (prop in myObject){
   print("myObject." + prop + " equals " + myObject[prop]);
}
```

The output of this program is:

```
myObject.name equals James
myObject.age equals 22
myObject.phone equals 555 1234
```

Note that the new behavior of the **for...in** loop construct in JScript .NET eliminates the need to use the **Enumerator** object to iterate elements of a collection.

See Also

Loops in JScript | JScript Conditional Structures | JScript Reference | for...in Statement

while Loops

A **while** loop is similar to a **for** loop in that it enables repeated execution of a block of statements. However, a **while** loop does not have a built-in counter variable or update expression. To control repetitive execution of a statement or block of statements with a more complex rule than simply "run this code n times", use a **while** loop. The following example demonstrates the **while** statement:

```
var x = 1;
while (x < 100) {
   print(x);
   x *= 2;
}
```

The output of this program is:

```
1
2
4
8
16
32
64
```

> **Note** Because **while** loops do not have explicit built-in counter variables, they are more vulnerable to infinite looping than the other types of loops. Moreover, because it is not necessarily easy to discover where or when the loop condition is updated, it is easy to write a **while** loop in which the condition never gets updated. For this reason, you should be careful when you design **while** loops.

As noted above, there is a **do...while** loop in JScript similar to the **while** loop. A **do...while** loop is guaranteed to always execute at least once since the condition is tested at the end of the loop rather than at the start. For example, the loop above can be rewritten as:

```
var x = 1;
do {
   print(x);
   x *= 2;
}
while (x < 100)
```

This output of this program is identical the output shown above.

See Also

Loops in JScript | JScript Conditional Structures | JScript Reference | while Statement | do...while Statement

break and continue Statements

The **break** statement in JScript stops the execution of a loop if some condition is met. (Note that **break** is also used to exit a **switch** block). The **continue** statement can be used to jump immediately to the next iteration, skipping the rest of the code block while updating the counter variable if the loop is a **for** or **for...in** loop.

The following example illustrates the use of the **break** and **continue** statements to control a loop.

```
for(var i = 0;i <=10 ;i++) {
   if (i > 7) {
      print("i is greater than 7.");
      break; // Break out of the for loop.
   }
   else {
      print("i = " + i);
      continue; // Start the next iteration of the loop.
      print("This never gets printed.");
   }
}
```

The output of this program is

```
i = 0
i = 1
i = 2
i = 3
i = 4
i = 5
i = 6
i = 7
i is greater than 7.
```

See Also

Loops in JScript | JScript Conditional Structures | JScript Reference | break Statement | continue Statement

JScript Reserved Words

JScript has a number of reserved words that have a specific meaning in the JScript language syntax. Consequently, these words are not good choices for names of functions, variables, or constants in scripts. There are three classes of reserved words.

Protected Reserved Words

break	case	catch	class	const
continue	debugger	default	delete	do
else	export	extends	false	finally
for	function	if	import	in
instanceof	new	null	protected	return
super	switch	this	throw	true
try	typeof	var	while	with

Protected reserved keywords cannot be used as identifiers. Using a protected reserved word as an identifier causes a compilation error when loading your script.

> **Note** Although "export" is a protected reserved word, it has no implementation.

New Reserved Words

abstract	boolean	byte	char	decimal
double	enum	final	float	get
implements	int	interface	internal	long
package	private	protected	public	sbyte
set	short	static	uint	ulong
ushort	void			

JScript also has a list of new reserved words. Like protected reserved words, these keywords have special meaning within the current version of JScript. For backwards compatibility reasons, new reserved words can be used as an identifiers. Once a new reserved word is used as an identifier, it loses its meaning as a keyword in the script. Using new reserved words as identifiers can cause confusion and should be avoided.

Future Reserved Words

assert	ensure	event	goto	invariant
namespace	native	require	synchronized	throws
transient	use	volatile		

JScript has a list of future reserved words, which are proposed as keywords in future extensions of JScript. Like the new reserved words, these may be used as identifiers in the current version of JScript. However, by avoiding these words, it will be easier to update scripts to take advantage of the features in upcoming versions of JScript.

When choosing identifiers it is also important to avoid words that are already the names of intrinsic JScript objects or functions, such as **String** or **parseInt**.

See Also

JScript Reference | JScript Language Tour

Security Considerations for JScript

Writing secure code is a challenge in any language. JScript .NET includes a few areas where developers might unknowingly use the language in an insecure way because the language does not force developers to use the best practices. Although JScript .NET has been designed with security as a goal, its primary goal is to promote rapid development of useful applications. In some cases, these two goals are in opposition.

You can avoid security issues if you are aware of the potential problems in several key areas that are listed below. These security considerations, except the **eval** method, are due to the new functionality that the .NET Framework introduces.

The eval Method

The most easily misused feature of JScript is the **eval** method, which allows the dynamic execution of JScript source code. Because a JScript .NET application that uses the **eval** method may execute any code that a program passes to it, every call to the **eval** method poses a security risk. Unless your application requires the flexibility of executing any code, consider explicitly writing the code that the application passes to the **eval** method.

To increase the security of applications that require the full flexibility provided by the **eval** method, the code passed to **eval** runs within a restricted context by default in JScript .NET 2003. The restricted security context forbids all access to system resources, such as the file system, the network, or the user interface. A security exception is generated if the code attempts to access those resources. However, code run by the **eval** method can still modify local and global variables. For more information, see eval Method.

Code written in previous versions of JScript may require the **eval** method to run code in the same security context as the calling code. To enable this behavior, you can pass the string "unsafe" as the optional second parameter to the **eval** method. You should only execute code strings obtained from trustworthy sources because in "unsafe" mode the code string executes with the same permissions as the calling code.

Security Attributes

The security attributes of the .NET Framework can be used to explicitly override the default security settings in JScript .NET. However, the security defaults should not be modified unless you know exactly what you are doing. In particular, one thing that you should not apply is the **AllowPartiallyTrustedCallers** attribute (APTCA) custom attribute because untrusted callers cannot safely call JScript code, in general. If you create a trusted assembly with APTCA that is then loaded by an application, a partially trusted caller could access fully trusted assemblies in the application. For more information, see Secure Coding Guidelines for the .NET Framework.

Partially Trusted Code and Hosted JScript Code

The engine which hosts JScript .NET allows any called code to modify parts of the engine, such as global variables, local variables, and prototype chains of any object. In addition, any function can modify the expando properties or methods of any expando object passed to it. Consequently, if a JScript .NET application calls partially trusted code or if it is running in an application with other code (such as from within a Visual Studio for Applications [VSA] host), the behavior of the application could be modified.

A consequence of this is that any JScript code in an application (or in an instance of an **AppDomain** class) should run at a trust level no higher than the rest of the code in the application. Otherwise, the other code could manipulate the engine for the JScript class, which could in turn modify data and affect the other code in the application. For more information, see AppDomain Class.

Assembly Access

JScript can reference assemblies using both strong names and simple text names. A strong name reference includes the version information of the assembly as well as a cryptographic signature that confirms the integrity and identity of the assembly. Although it is easier to use a simple name when referring to an assembly, a strong name protects your code in case another assembly on your system has the same simple name but different functionality. For more information, see Referencing a Strong-Named Assembly.

Threading

The JScript runtime is not designed to be thread-safe. Consequently, multithreaded JScript code may have unpredictable behavior. If you develop an assembly in JScript .NET, keep in mind that it may be used in a multithreaded context. You should use classes from the **System.Threading** namespace, such as the **Mutex** class, to ensure that the JScript .NET code in the assembly runs with the proper synchronization.

Because proper synchronization code is difficult to write in any language, you should not attempt to write general-purpose assemblies in JScript unless you have a good understanding of how to implement the necessary synchronization code. For more information, see System.Threading Namespace.

> **Note** You do not need to write synchronization code for ASP.NET applications written in JScript .NET because ASP.NET manages the synchronization of all the threads it spawns. However, Web controls written in JScript must contain synchronization code because they behave like assemblies.

Runtime Errors

Because JScript .NET is a loosely typed language, it is more tolerant of potential type-mismatches than some other languages, such as Visual Basic .NET and C#. Because type mismatches can cause run-time errors in applications, it is important to discover potential type-mismatches as you develop the code. You can use the **/warnaserror** flag with the command-line compiler or the **warninglevel** attribute of the **@ Page** directive in ASP.NET pages. For more information, see /warnaserror and @ Page.

Compatibility Mode

Assemblies complied in compatibility mode (with the **/fast-** option) are less secure than those compiled in fast mode (the default mode). The **/fast-** option enables language features that are not available by default, but are required for compatibility with scripts written for JScript version 5.6 and earlier. For example, expando properties can be dynamically added to intrinsic objects, such as the **String** object, in compatibility mode.

Compatibility mode is provided to help developers build standalone executables from legacy JScript code. When developing new executables or libraries, use default mode. Not only does this help secure applications, but it also ensures better performance and better interaction with other assemblies. For more information, see /fast.

See Also

Security Portal | Upgrading Applications Created in Previous Versions of JScript

JScript .NET Language Reference

In This Section

Data Types

Directives

Errors

Functions

Literals

Methods

Modifiers

Objects

Operators

Properties

Statements

Related Sections

Introduction to ASP.NET
Explains how ASP.NET evolved from Active Server Pages (ASP) and how the .NET Framework enables functionality in this new environment.

.NET Framework Reference
Lists links to topics that explain the syntax and structure of the .NET Framework class library and other essential elements.

Data Types

A data type specifies the type of value that a variable, constant, or function can accept. Type annotation of variables, constants, and functions helps reduce programming errors by limiting data appropriate types. Furthermore, type annotation also produces faster, more efficient code.

In This Section

boolean Data Type

byte Data Type

char Data Type

decimal Data Type

double Data Type

float Data Type

int Data Type

long Data Type

Number Data Type

sbyte Data Type

short Data Type

String Data Type

uint Data Type

ulong Data Type

ushort Data Type

Related Sections

JScript Data Types
Includes links to topics that explain how to use primitive data types, reference data types, and .NET Framework data types in JScript.

Data Type Summary
Lists value and reference data types supported by JScript, corresponding .NET Framework equivalents, storage sizes, and ranges.

JScript Reference
Lists elements that comprise JScript Language Reference and links to topics that explain the details behind the proper use of language elements.

boolean Data Type

The value of the boolean type, which is either true or false, depends on whether the true or false keyword is assigned to the type.

The corresponding .NET Framework data type is **System.Boolean**. The **Boolean** data type is identical to the **boolean** data type.

Remarks

The properties and methods of the **boolean** data type are the same as the **System.Boolean** properties and methods.

JScript also defines a **Boolean** object. The **boolean** data type interoperates with the **Boolean** object. Consequently, a **Boolean** object can call the methods and properties of the **boolean** data type, and a **boolean** data type can call the methods and properties of the **Boolean** object. For additional information, see Boolean Object Properties and Methods. Furthermore, **Boolean** objects are accepted by functions that take **boolean** data types, and vice versa.

The **boolean** data type should be used instead of the **Boolean** object in most circumstances.

Properties and Methods

boolean Members

Requirements

Version .NET

See Also

Data Types | Data Type Summary | Boolean Object | Boolean Structure | true Literal | false Literal

byte Data Type

The **byte** type is stored as one unsigned byte.

The **byte** type can represent integers in the range from 0 to 255, inclusive.

The corresponding .NET Framework data type is **System.Byte**. The properties and methods of the **byte** data type are the same as the **System.Byte** properties and methods.

Properties and Methods

Byte Members

Requirements

Version .NET

See Also

Data Types | Data Type Summary | Byte Structure

char Data Type

The **char** type is stored as a two-byte Unicode character.

The **char** type can represent any of the 65,536 Unicode characters.

The corresponding .NET Framework data type is **System.Char**. The properties and methods of the **char** data type are the same as the **System.Char** properties and methods.

Properties and Methods

Char Members

Requirements

Version .NET

See Also

Data Types | Data Type Summary | Char Structure| String Data Type

decimal Data Type

The **decimal** type is stored as a 12-byte integral part, a 1-bit sign, and a scaling factor.

The **decimal** type can accurately represent very large or very precise decimal numbers. Numbers as large as 10^{28} (positive or negative) and with as many as 28 significant digits can be stored as a **decimal** type without loss of precision. This type is useful for applications (such as accounting) where rounding errors must be avoided.

The corresponding .NET Framework data type is **System.Decimal**. The properties and methods of the **decimal** data type are the same as the **System.Decimal** properties and methods.

Properties and Methods

Decimal Members

Requirements

Version .NET

See Also

Data Types | Data Type Summary | Decimal Structure

double Data Type

The **double** type is stored as an eight-byte, double-precision, floating-point number. It represents a double-precision 64-bit IEEE 754 value.

The **double** type can represent numbers as large as 10^{308} (positive or negative) with an accuracy of about 15 digits, and as small as 10^{-323}. The **double** type can also represent **NaN** (Not a Number), positive and negative infinity, and positive and negative zero.

This type is useful for applications that need large numbers but do not need precise accuracy. If you require very accurate numbers, consider using the **Decimal** data type.

The corresponding .NET Framework data type is **System.Double**. The **double** type is equivalent to the **Number** type.

Remarks

The properties and methods of the **double** data type are the same as the **System.Double** properties and methods.

JScript defines a **Number** object. The **double** data type interoperates with **Number** object. Consequently, a **Number** object can call the methods and properties of the **double** data type, and a **double** data type can call the methods and properties of the **Number** object. For additional information, see Number Object Properties and Methods. Furthermore, **Number** objects are accepted by functions that take **double** data types, and vice versa.

The **double** data type should be used instead of the **Number** object in most circumstances.

Properties and Methods

Double Members

Requirements

Version .NET

See Also

Data Types | Data Type Summary | Number Data Type | decimal Data Type | Double Structure

float Data Type

The **float** type is stored as a four-byte, single-precision, floating-point number. It represents a single-precision 32-bit IEEE 754 value.

The **float** type can represent numbers as large as 10^{38} (positive or negative) with an accuracy of about seven digits, and as small as 10^{-44}. The **float** type can also represent **NaN** (Not a Number), positive and negative infinity, and positive and negative zero.

This type is useful for applications that need large numbers but do not need precise accuracy. If you require very accurate numbers, consider using the **Decimal** data type.

The corresponding .NET Framework data type is **System.Single**. The properties and methods of the **float** data type are the same as the **System.Single** properties and methods.

Properties and Methods

Single Members

Requirements

Version .NET

See Also

Data Types | Data Type Summary | decimal Data Type | Single Structure

int Data Type

The **int** data type is stored as a four-byte integer.

The **int** type can represent integers in the range from negative 2,147,483,648 to positive 2,147,483,647, inclusive.

The corresponding .NET Framework data type is **System.Int32**. The properties and methods of the **int** data type are the same as the **System.Int32** properties and methods.

Properties and Methods

Int32 Members

Requirements

Version .NET

See Also

Data Types | Data Type Summary | Int32 Structure

long Data Type

The **long** type is stored as an eight-byte integer.

The **long** type can represent integers in the approximate range from negative 10^{19} through 10^{19}.

The corresponding .NET Framework data type is **System.Int64**. The properties and methods of the **long** data type are the same as the **System.Int64** properties and methods.

Properties and Methods

Int64 Members

Requirements

Version .NET

See Also

Data Types | Data Type Summary | Int64 Structure

Number Data Type

The **Number** type is stored as an eight-byte, double-precision, floating-point number. It represents a double-precision 64-bit IEEE 754 value.

The **Number** type can represent numbers as large as 10^{308} (positive or negative) with an accuracy of about 15 digits, and as small as 10^{-323}. The **Number** type can also represent **NaN** (Not a Number), positive and negative infinity, and positive and negative zero.

This type is useful for applications that need large numbers but do not need precise accuracy. If you require very accurate numbers, consider using the **Decimal** data type.

The corresponding .NET Framework data type is **System.Double**. The **Number** type is equivalent to the **double** type.

Remarks

The properties and methods of the **Number** data type are the same as the **System.Double** properties and methods.

JScript also defines a **Number** object. The **Number** data type interoperates with **Number** object. Consequently, a **Number** object can call the methods and properties of the **Number** data type, and a **Number** data type can call the methods and properties of the **Number** object. For additional information, see Number Object Properties and Methods. Furthermore, **Number** objects are accepted by functions that take **Number** data types, and vice versa.

The **Number** data type should be used instead of the **Number** object in most circumstances.

Properties and Methods

Double Members

Requirements

Version .NET

See Also

Data Types | Data Type Summary | double Data Type | decimal Data Type | Number Object | Double Structure

sbyte Data Type

The **sbyte** type is stored as one signed byte.

The **sbyte** type can represent integers in the range from negative 128 to positive 127, inclusive.

The corresponding .NET Framework data type is **System.SByte**. The properties and methods of the **sbyte** data type are the same as the **System.SByte** properties and methods.

Properties and Methods

SByte Members

Requirements

Version .NET

See Also

Data Types | Data Type Summary | Sbyte Structure

short Data Type

The **short** data type is stored as a two-byte integer.

The **short** type can represent integers in the range from negative 32,768 to positive 32,767, inclusive.

The corresponding .NET Framework data type is **System.Int16**. The properties and methods of the **short** data type are the same as the **System.Int16** properties and methods.

Properties and Methods

Int16 Members

Requirements

Version .NET

See Also

Data Types | Data Type Summary | Int16 Structure

String Data Type

The length of a **String** can be from zero to approximately two billion characters. Each character is a 16-bit Unicode value.

The equivalent .NET Framework data type is **System.String**.

Remarks

The properties and methods of the **String** data type are the same as the **System.String** properties and methods.

JScript also defines a **String** object, which provides different properties and methods from the **String** data type. You cannot create properties or add methods to variables of the **String** data type, while you can for instances of the **String** object.

The **String** object interoperates with **String** data. Consequently, a **String** object can call the methods and properties of the **String** data type, and a **String** data type can call the methods and properties of the **String** object. For additional information, see String Object Properties and Methods. Furthermore, **String** objects are accepted by functions that take **String** data types, and vice versa.

Escape sequences can be used in string literals to represent special characters that cannot be used directly in a string, such as the newline character or Unicode characters. When a script is compiled, each escape sequence in a string literal is converted into the characters it represents. For additional information, see String Data.

JScript does not interpret special Unicode sequences, such as surrogate pairs, nor does it normalize strings when comparing them.

> **Note** Pairs of Unicode characters that represent a single character and only have meaning when combined are known as surrogate pairs.

Some characters can be represented by more than one sequence of Unicode characters. Separate normalized sequences are interpreted identically if they represent the same character.

Properties and Methods

String Members

Requirements

Version .NET

See Also

Data Types | Data Type Summary | String Object | String Class | char Data Type | String Data

uint Data Type

The **uint** type is stored as a four-byte unsigned integer.

The **uint** type can represent integers in the range from 0 to 4,294,967,295, inclusive.

The corresponding .NET Framework data type is **System.UInt32**. The properties and methods of the **uint** data type are the same as the **System.UInt32** properties and methods.

Properties and Methods

UInt32 Members

Requirements

Version .NET

See Also

Data Types | Data Type Summary | UInt32 Structure

ulong Data Type

The **ulong** type is stored as an eight-byte unsigned integer.

The **ulong** type can represent integers in the range from 0 through about 10^{20}.

The corresponding .NET Framework data type is **System.UInt64**. The properties and methods of the **ulong** data type are the same as the **System.UInt64** properties and methods.

Properties and Methods

UInt64 Members

Requirements

Version .NET

See Also

Data Types | Data Type Summary | UInt64 Structure

ushort Data Type

The **ushort** data type is stored as a two-byte unsigned integer.

The **ushort** type can represent integers in the range from 0 to 65,535, inclusive.

The corresponding .NET Framework data type is **System.UInt16**. The properties and methods of the **ushort** data type are the same as the **System.UInt16** properties and methods.

Properties and Methods

UInt16 Members

Requirements

Version .NET

See Also

Data Types | Data Type Summary | UInt16 Structure

Directives

JScript directives control specific compiler, debugger, and error message options.

In This Section

@debug Directive
Turns on or off the emission of debug symbols.

@position Directive
Provides meaningful position information in error messages.

Related Sections

JScript Reference
Lists elements that comprise JScript Language Reference and links to topics that explain the details behind the proper use of language elements.

.NET Framework Reference
Lists links to topics that explain the syntax and structure of the .NET Framework class library and other essential elements.

@debug Directive

Turns the emission of debug symbols on or off.

```
@set @debug(on | off)
```

Arguments

on
Default. Keyword that turns debug on.
off
Optional. Keyword that turns debug off.

Remarks

Program code that a JScript author writes sometimes differs from the actual code being compiled and run. Host environments, such as ASP.NET, or development tools may generate their own code and add it into the program. This code is generally of no interest to the author during debugging. Consequently, when debugging their code, code authors generally only want to see the parts of the program that they wrote without the parts generated by their development tools. Package authors may want to turn off debugging for similar reasons.

The compiler emits debugging symbols only when compiling from the command line with the **/debug** option or when compiling an ASP.NET page with the debug flag set in the **@page** directive. In those circumstances, the **debug** directive is on by default. When the **debug** directive appears, it remains in effect until the end of the file is encountered or until the next **debug** directive is found.

When the **debug** directive is off, the compiler does not emit debugging information for local variables (variables defined within functions or methods). However, the **debug** directive does not prevent emission of the debugging information for global variables.

Example

The following code emits debug symbols for the local variable `debugOnVar`, but not for `debugOffVar`, when compiled from the command line with the **/debug** option:

```
function debugDemo() {
   // Turn debugging information off for debugOffVar.
   @set @debug(off)
   var debugOffVar = 42;
   // Turn debugging information on.
   @set @debug(on)

   // debugOnVar has debugging information.
   var debugOnVar = 10;

   // Launch the debugger.
   debugger;
}

// Call the demo.
debugDemo();
```

Requirements

Version .NET

See Also

@set Statement | @position Directive | /debug | debugger Statement | Writing, Compiling, and Debugging JScript Code

@position Directive

Provides meaningful position information in error messages.

```
@set @position(end | [file = fname ;] [line = lnum ;] [column = cnum])
```

Arguments

fname
Required if **file** is used. A string literal that represents a filename, with or without drive or path information.

lnum
Required if **line** is used. Any non-negative integer that represents a line of authored code.

cnum
Required if **column** is used. Any non-negative integer that represents a column in authored code.

Remarks

Program code that a JScript author writes sometimes differs from the actual code being compiled and run. Host environments, such as ASP.NET, or development tools may generate their own code and add it into the program. This code is generally of no interest to the author, but it has the potential to cause confusion for the author when errors occur.

Instead of correctly identifying the line of the author's code where an error occurred, the compiler may incorrectly identify an error line that doesn't even exist in the original authored code. This is because the additional generated code has changed the relative position of the author's original code.

Example

In the following example, the line number in a file is changed to accommodate code inserted into the author's code by a JScript host. The line numbers in the left column represent the original source as seen by the user.

```
01  ..  // 10 lines of host-inserted code.
..  ..  //...
10  ..  // End of host-inserted code.
11  ..  @set @position(line = 1)
12  01  var i : int = 42;
13  02  var x = ; // Error reported as being on line 2.
14  03  //Remainder of file.
```

Requirements

Version .NET

See Also

@set Statement | @debug Directive

Errors

Error messages help you troubleshoot unexpected results or behaviors in scripts. The following sections explain how to resolve errors that occur during runtime or as the result of syntax inconsistencies.

In This Section

JScript Run-time Errors

JScript Syntax Errors

Related Sections

JScript Reference
Lists elements that comprise JScript Language Reference and links to topics that explain the details behind the proper use of language elements.

Error Messages
Provides a list of links to error messages that pertain to the Visual Studio Integrated Development Environment (IDE) and other development languages.

JScript Run-time Errors

JScript run-time errors result when JScript script attempts to perform an action that the system cannot execute. JScript run-time errors occur when variable expressions are being evaluated during script execution and memory is being dynamically allocated.

Run-time errors can be trapped and examined by a JScript program. By enclosing the code that produces the error in a **try** block, any thrown error can be caught with a **catch** block. The errors that JScript throws are **Error** objects. JScript code can generate custom errors of any data type, including **Error** objects, by using the **throw** statement. A program can display the error number and message of a caught **Error** object to help identify the error. If an error is not caught, the script will terminate.

There are several ways to get help for a particular error message:

- Look for the error number and message in the table of contents contained in the JScript Run-time Errors node.
- Type the error number in the **Look for** box in the **Index**. The error number is in the format JSxxxx, where xxxx is the four-digit error code.
- Type the error message in the **Look for** box in the **Search**. Remember that some error messages include words in single quotes. The quoted words refer to identifiers from the code and are not part of the error message. Do not include the quoted words as part of your search.

See Also

JScript Syntax Errors | Error Object | try...catch...finally Statement | throw Statement

JS5000: Cannot assign to 'this'

A value is assigned to **this**, a JScript keyword that refers to either:

- the object currently executing a method,
- the global object if there is no current method (or the method does not belong to any other object).

A method is a JScript function that is invoked through an object. Inside a method, the **this** keyword is a reference to the object through which the method was invoked (which happens to be the object created by invoking the class constructor with the **new** operator).

Inside a method, you can use **this** to refer to the current object, but you cannot assign a new value to **this**.

To correct this error

- Do not assign a value to **this**. To access a property or method of an instantiated object, use the dot operator (for example, circle**.**radius).

Note You cannot name a user-created variable **this**; it is a JScript reserved word.

See Also

this Statement | Troubleshooting Your Scripts | JScript Reference

JS5001: Number expected

The code invoked **Number.prototype.toString** or **Number.prototype.valueOf** method on an object of a type other than **Number**. The object of this type of invocation must be of type **Number**.

To correct this error

- Only invoke the **Number.prototype.toString** or **Number.prototype.valueOf** methods on objects of type **Number**.

See Also

Number Object | toString Method | valueOf Method | prototype Property | Objects

JS5002: Function expected

The code either invoked one of the **Function prototype** methods on an object that was not a **Function** object, or it used an object in a function call context. For example, the following code produces this error because **mysample** is not a function.

```
var mysample = new Object();  // Create a new object called "mysample".
var x = mysample();           // Try and call mysample as if it were a function.
```

To correct this error

- Only call **Function prototype** methods on **Function** objects.
- Ensure that the function call operator **()** calls functions only.

See Also

Function Object | prototype Property | Objects

JS5003: Cannot assign to a function result

The code attempts to assign a value to a function result. The result of a function can be assigned to a variable, but it cannot be used as a variable. If you want to assign a new value to the function itself, omit the parentheses (the function call operator).

To correct this error

- Do not use the value of a function call result as something you can *assign to*. You can assign the result of the function call *to a variable*, though.

  ```
  myVar = myFunction(42);
  ```

- Alternatively, you can assign the function itself (and not its return value) to a variable.

  ```
  myFunction = new Function("return 42;");
  ```

See Also

Function Object | JScript Language Features | JScript Functions | Methods

JS5005: String expected

The code invoked **String.prototype.toString** or **String.prototype.valueOf** method on an object of a type other than **String**. The object of this type of invocation must be of type **String**.

To correct this error

- Only invoke the **String.prototype.toString** or **String.prototype.valueOf** methods on objects of type **String**.

See Also

String Object | toString Method | valueOf Method | prototype Property | Objects

JS5006: Date object expected

The code invoked the **Date.prototype.toString** or **Date.prototype.valueOf** method on an object of a type other than **Date**. The object of this type of invocation must be of type **Date**.

To correct this error

- Only invoke the **Date.prototype.toString** or **Date.prototype.valueOf** methods on objects of type **Date**.

See Also

Date Object | toString Method | valueOf Method | prototype Property | Objects

JS5007: Object expected

The code invoked the **Object.prototype.toString** or **Object.prototype.valueOf** method on an object of a type other than **Object**. The object of this type of invocation must be of type **Object**.

To correct this error

- Only invoke the **Object.prototype.toString** or **Object.prototype.valueOf** methods on objects of type **Object**.

See Also

Object Object | toString Method | valueOf Method | prototype Property | Objects

JS5008: Illegal assignment

The code attempted to assign a value to a read-only identifier. A read-only identifier cannot be assigned a value. For example, host-defined objects and external COM objects are read-only identifiers.

To correct this error

- Do not assign values to read-only identifiers.

See Also

Assignment Operator (=)

JS5009: Undefined identifier

The JScript compiler does not recognize an identifier. This may occur when a referenced variable does not exist or when a **with** block is used to access an object property that does not exist.

To correct this error

- Declare the variable with a **var** statement (as in **var** x;).
- Make sure that only valid object members are referenced in a **with** block.
- Reference the object member explicitly instead of using the **with** statement.

See Also

JScript Variables and Constants | Scope of Variables and Constants | var Statement | with Statement | JScript Reference

JS5010: Boolean expected

The code invoked the **Boolean.prototype.toString** or **Boolean.prototype.valueOf** method on an object of a type other than **Boolean**. The object of this type of invocation must be of type **Boolean**.

To correct this error

- Only invoke the **Boolean.prototype.toString** or **Boolean.prototype.valueOf** methods on objects of type **Boolean.**

See Also

Boolean Object | toString Method | valueOf Method | prototype Property | Objects

JS5013: VBArray expected

An object supplied to **VBArray** was not a Visual Basic safeArray, when one was expected. Visual Basic safeArrays, which cannot be created directly in JScript, must be imported by retrieving the value from an existing ActiveX or other object or from a Visual Basic script on the same Web page.

To correct this error

- Ensure that only a Visual Basic safeArray object is passed to the **VBArray** constructor.
- Use a **System.Array** object. This allows for any .NET language (including JScript .NET and Visual Basic .NET) to access and modify the array.

See Also

VBArray Object | Using Arrays

JS5015: Enumerator object expected

The code invoked the **Enumerator.prototype.atEnd**, **Enumerator.prototype.item**, **Enumerator.prototype.moveFirst**, or **Enumerator.prototype.moveNext** method on an object of a type other than **Enumerator**. The object of this type of invocation must be of type **Enumerator**.

To correct this error

- Only invoke the **Enumerator.prototype.atEnd**, **Enumerator.prototype.item**, **Enumerator.prototype.moveFirst**, or **Enumerator.prototype.moveNext** methods on objects of type **Enumerator**. To find out if your object is an **Enumerator** object, use:

  ```
  if(x instanceof Enumerator)
  ```

See Also

Enumerator Object | atEnd Method | item Method | moveFirst Method | moveNext Method | prototype Property | Objects

JS5016: Regular Expression object expected

The code invoked the **RegExp.prototype.toString** or **RegExp.prototype.valueOf** method on an object of a type other than **RegExp**. The object of this type of invocation must be of type **RegExp**.

To correct this error

- Only invoke the **RegExp.prototype.toString** or **RegExp.prototype.valueOf** methods on objects of type **RegExp**.

See Also

Regular Expression Object | Regular Expression Syntax | toString Method | valueOf Method | prototype Property | Objects

JS5017: Syntax error in regular expression

A regular expression search string's syntax violates one or more of the grammatical rules of a JScript regular expression.

To correct this error

- Make sure that a regular expression search string adheres to the JScript regular expression syntax.

See Also

Regular Expression Object | Regular Expression Syntax | compile Method

JS5022: Exception thrown and not caught

The code includes a **throw** statement that is not enclosed within a **try** block, or the code does not include an associated **catch** block to trap the error. Exceptions that are thrown from within the **try** block using the **throw** statement are caught outside the **try** block by a **catch** statement.

To correct this error

- Enclose code that can throw an exception in a **try** block, and ensure there is a corresponding **catch** block.
- Make sure a catch statement expects the correct form of exception.
- If the exception is rethrown, make sure there is another corresponding catch statement.

See Also

Error Object | throw Statement | try...catch...finally Statement

JS5023: Function does not have a valid prototype object

The code uses **instanceof** to determine if an object was derived from a particular function class, but the code redefined the object's **prototype** property as either **null** or an external object type (both not valid JScript objects). An external object can be an object from the host object model (for example, Internet Explorer's document or window object), or an external COM object.

To correct this error

- Ensure the function's **prototype** property refers to a valid JScript object.

See Also

Function Object | prototype Property | Objects

JS5024: The URI to be encoded contains an invalid character

A string encoded as a Uniform Resource Identifier (URI) contains invalid characters. Although most characters are valid inside strings to be converted to URIs, some Unicode character sequences are invalid in this context.

To correct this error

- Ensure the string to be encoded contains only valid Unicode sequences.

  A complete URI is comprised of a sequence of components and separators. The general form is:

  ```
  <Scheme>:<first>/<second>;<third>?<fourth>
  ```

  The names in angle brackets represent components, and the ":", "/", ";" and "?" are reserved characters used as separators.

See Also

encodeURI Method | encodeURIComponent Method

JS5025: The URI to be decoded is not a valid encoding

The code attempted to decode an improperly formed Uniform Resource Identifier (URI). URIs have a special syntax; most non-alphanumeric characters must be encoded before they can be used in a URI. The **encodeURI** and **encodeURIComponent** methods cannot create a URI from a normal JScript string.

A complete URI is comprised of a sequence of components and separators. The general form is:

```
<Scheme>:<first>/<second>;<third>?<fourth>
```

The names in angle brackets represent components, and the ":", "/", ";" and "?" are reserved characters used as separators.

To correct this error

- Ensure that the code is trying to decode valid URIs only. For example, a normal JScript string may not be a valid URI because it may contain invalid characters.

See Also

decodeURI Method | decodeURIComponent Method

JS5026: The number of fractional digits is out of range

The function **Number.prototype.toExponential** cannot accept an invalid argument. The argument to the function **toExponential()** must be between 0 and 20 (inclusive).

To correct this error

- Ensure the argument to **toExponential()** is not too large or too small.

See Also

Number Object | toExponential Method | prototype Property | Objects

JS5027: The precision is out of range

The function **Number.prototype.toPrecision** cannot accept an invalid argument. The argument to the function **toPrecision** must between 1 and 21 (inclusive).

To correct this error

- Ensure the argument to **toPrecision** is not too large or too small.

See Also

Number Object | toPrecision Method | prototype Property | Objects

JS5029: Array length must be zero or a positive integer

An argument that calls the **Array** constructor is negative or not a number (**NaN**). Note that JScript automatically converts fractional numbers to whole integers.

To correct this error

- Use only positive integers or the number zero when creating a new **Array** object. To create an array with a single element, use a two-step process. First, create an array with one element. Second, place the value in the first element (array[0]).

  The following example demonstrates the correct way to specify an array with a single numeric element.

  ```
  var piArray = new Array(1);
  piArray [0] = 3.14159;
  ```

  There is no upper limit for the size of an array, other than the maximum integer value (approximately 4 billion).

See Also

Using Arrays | JScript Reference

JS5030: Array length must be assigned a positive integer or zero

A value assigned to the **length** property of an **Array** object is negative or not a number (**NaN**). Note that JScript automatically converts fractional numbers to whole integers.

To correct this error

- Assign a positive integer or zero to the length property. The following example demonstrates the correct way to set the **length** property of an **Array** object.

  ```
  var my_array = new Array();
  my_array.length = 99;
  ```

  There is no upper limit for the size of an array, other than the maximum integer value (approximately 4 billion).

See Also

Using Arrays | JScript Reference

JS5031: Array object expected

The program is attempting to use something that is not an **Array** object in a context where an Array object is required.

To correct this error

- Make sure that an **Array** object is used in this context.

See Also

Troubleshooting Your Scripts | JScript Reference | Array Object

JS5032: No such constructor

The **new** operator is applied to an identifier, but the identifier does not correspond to either a constructor function or a class constructor.

To correct this error

- Make sure that the new operator is applied to a constructor function or a class constructor.

See Also

Troubleshooting Your Scripts | JScript Reference | Creating Your Own Objects with Constructor Functions | Creating Your Own Classes

JS5033: Eval may not be called via an alias

The program, which has a variable set equal to the **eval** method, defines an alias and then uses that alias as a function. An alias may not be used for the **eval** method.

To correct this error

- Call the **eval** method directly.

See Also

Troubleshooting Your Scripts | JScript Reference | eval Method

JS5034: Not yet implemented

The program is attempting to use a feature that has not been implemented.

To correct this error

- Remove the reference to the unimplemented feature.

See Also

Troubleshooting Your Scripts | JScript Reference

JS5035: Cannot provide null or empty named parameter name

Named parameters call a JScript function or method, and one of the named parameter names is empty or null. This is not allowed since all parameters have non-null names.

> **Note** Named parameters cannot be used when calling functions and methods using JScript .NET. However, JScript .NET functions and methods can be called from other languages (such as Visual Basic .NET) that support named parameters. For more information, see Argument Passing by Position and by Name.

To correct this error

- Provide a parameter name for each named parameter name.

See Also

Troubleshooting Your Scripts | JScript Reference

JS5036: Duplicate named parameter name

Named parameters call a JScript function or method, and one of the named parameter names is used twice. This is not allowed since each parameter name must be unique.

> **Note** Named parameters cannot be used when calling functions and methods using JScript .NET. However, JScript .NET functions and methods can be called from other languages (such as Visual Basic .NET) that support named parameters. For more information, see Argument Passing by Position and by Name.

To correct this error

- Provide a unique name for each named parameter name.

See Also

Troubleshooting Your Scripts | JScript Reference

JS5037: The specified name is not the name of a parameter

Named parameters call a JScript function or method, and one of the named parameter names does not correspond to a parameter name. This is not allowed since each named parameter name must refer to a parameter name.

> **Note** Named parameters cannot be used when calling functions and methods using JScript .NET. However, JScript .NET functions and methods can be called from other languages (such as Visual Basic .NET) that support named parameters. For more information, see Argument Passing by Position and by Name.

To correct this error

- Provide a parameter name for each named parameter name.

See Also

Troubleshooting Your Scripts | JScript Reference

JS5038: Too few arguments specified

Too few arguments specified. The number of parameter names cannot exceed the number of arguments passed in.

Named parameters call a JScript function or method, but the number of arguments passed in exceeds the number of arguments specified by the function or method. This is not allowed since at least one of the arguments passed in must be discarded.

> **Note** When calling functions and methods using JScript .NET, named parameters cannot be used. However, JScript .NET functions and methods can be called from other languages (such as Visual Basic .NET) that support named parameters. For more information, see Argument Passing by Position and by Name.

To correct this error

- Make sure that the number of arguments passed in does not exceed the number of parameter names.

See Also

Troubleshooting Your Scripts | JScript Reference

JS5039: The expression cannot be evaluated in the debugger

While debugging a JScript .NET program, an expression was entered into the Command Window that could not be evaluated.

To correct this error

- Make sure that only valid JScript .NET expressions are entered in the Command Window.

See Also

Troubleshooting Your Scripts | JScript Reference | Writing, Compiling, and Debugging JScript Code

JS5040: Assignment to read-only field or property

The code assigns a value to a read-only identifier. This is not allowed because code cannot write to read-only identifiers.

The **const** statement defines a read-only field or a constant. A **function get** statement without a matching **function set** statement defines a read-only property.

To correct this error

- Make sure the code does not assign values to read-only identifiers.
- Define the field or variable with the **var** statement to make it assignable.
- Add a matching **function set** statement to the property to make it assignable.

See Also

Troubleshooting Your Scripts | JScript Reference | Class-based Objects | const Statement | function get Statement | function set Statement

JS5041: The property can only be assigned to

The code reads the value of a write-only property. This is not allowed because code cannot read values from write-only identifiers.

A **function set** statement without a matching **function get** statement defines a write-only property.

To correct this error

- Make sure the code does not read from a write-only property.
- Add a matching **function get** statement to the property to make it readable.

See Also

Troubleshooting Your Scripts | JScript Reference | Class-based Objects | function get Statement | function set Statement

JS5042: The number of indices does not match the dimension of the array

The code accesses an element of an array, but the number of indices does not match the number of dimensions of the array.

To correct this error

- Make sure the number of indices used to access an array element matches the number of dimensions specified when defining the array.

See Also

Typed Arrays | Troubleshooting Your Scripts | JScript Reference

JS5043: Methods with ref parameters cannot be called in the debugger

While debugging a JScript .NET program, a call to a method that takes parameters by reference was entered into the Command Window. This is not allowed since by reference parameters can change the values of variables.

To correct this error

- Make sure that only methods that do not take parameters by reference are called from the Command Window.

See Also

Troubleshooting Your Scripts | JScript Reference | Writing, Compiling, and Debugging JScript Code

JS5044: The Deny, PermitOnly and Assert security methods cannot be called using late binding

The code calls the **PermitOnly**, **Assert**, or **Deny** method of a late-bound **CodeAccessPermission** object. This is not allowed for security reasons. To use the **PermitOnly**, **Assert**, and **Deny** methods, the variable that stores the **CodeAccessPermission** object must be explicitly typed (early-bound) to store only **CodeAccessPermission** objects.

To correct this error

- Use type annotation when defining the variable that stored the **CodeAccessPermission** object.

See Also

Troubleshooting Your Scripts | JScript Reference | CodeAccessPermission Class

JS5045: JScript .NET does not support declarative security attributes

A custom attribute that inherits from **CodeAccessSecurityAttribute** is applied to the definition of a method, class, or assembly. This is not allowed. Dynamic security must be used instead of declarative security attributes to control access to a portion of code.

> **Note** Only early-bound code can make calls to the **Assert**, **Deny** or **PermitOnly** security methods within the .NET Framework. This means that type-annotated variables must be used to store permission objects, since type annotation allows the compiler to generate early-bound code. Moreover, code generated at runtime (with the **eval** method or with a **Function** object created with the **new** operator) is late-bound code, which prevents it from making calls to the **Assert**, **Deny** or **PermitOnly** methods.

In the following example, dynamic security is used to deny access to a particular file by a method.

```
import System;
import System.IO;
import System.Security;
import System.Security.Permissions;
class Alpha{
   function Bravo() {
      var fileioperm : FileIOPermission;
      fileioperm = new FileIOPermission(FileIOPermissionAccess.AllAccess, 'd:\\temp\\myfile.txt');
      fileioperm.Deny();
      // Any additional code in this method will be
      // denied access to d:\temp\myfile.txt.
   }
}
```

To correct this error

- Use dynamic security instead of declarative security to declare a secure method or assembly.

See Also

Troubleshooting Your Scripts | JScript Reference | CodeAccessSecurityAttribute Class | FileIOPermission Class

JScript Syntax Errors

JScript syntax errors result when the structure of JScript statements violates one or more of the grammatical rules of the JScript scripting language. JScript syntax errors occur during the program compilation stage before the program starts execution.

There are several ways to get help for a particular error message:

- Look for the error number and message in the table of contents contained in the JScript Syntax Errors node.
- Type the error number in the **Look for** box in the **Index**. The error number is in the format JSxxxx, where xxxx is the four-digit error code.
- Type the error message in the **Look for** box in the **Search**. Remember that some error messages include words in single quotes. The quoted words refer to identifiers from your code and are not part of the error message. You should not include the quoted words as part of your search.

See Also

JScript Run-time Errors

JS0005: Invalid procedure call or argument

The code calls a procedure, but there is no procedure defined which matches the call.

To correct this error

- Make sure that the data types passed to the procedure match the data types that the procedure is defined to take.
- Make sure that the number of arguments passed to the procedure matches the number that the procedure expects.

See Also

Troubleshooting Your Scripts | JScript Reference

JS0007: Out of memory

The program has consumed all available memory, which might occur when dealing with large amounts of data. One way to avoid this error is to make efficient use of the available memory. This can be accomplished by making sure the program does not reserve memory (in the form of arrays, objects, and so on) that is not needed.

Another way to reduce the memory used by a program is to assist the garbage collection routine to dynamically free memory. The garbage collection routine used by JScript takes care of freeing unused memory. The routine frees data that can no longer be accessed by the program. Data may become inaccessible when the data in a variable is replaced with new data or when the scope changes and a variable is no longer accessible.

To free memory in your program, set variables that consume large amounts of memory (such as large arrays or other objects) to **null** once they are no longer needed. This allows the garbage collector to free the memory.

To correct this error

- Make sure that your code uses memory efficiently.
- Declare objects that consume large amounts of memory immediately before they are needed.
- Set variables to **null** when they are no longer needed.

See Also

Troubleshooting Your Scripts | JScript Reference

JS0013: Type mismatch

The code is attempting to use data of one data type in a context that expects an incompatible data type. This can happen when assigning a value to a variable or when passing an argument to a function that has type-annotated parameters.

To correct this error

- Make sure that the code passes data of a data type that can be coerced to the expected data type.

See Also

Troubleshooting Your Scripts | JScript Reference

JS0028: Out of stack space

The program has consumed all of the available stack space. This can occur when a recursive function never explicitly terminates.

To correct this error

- Make sure that recursive functions explicitly terminate.

See Also

Troubleshooting Your Scripts | JScript Reference | Recursion

JS0051: Internal error

The code has revealed an internal error in the script engine.

To correct this error

- Rewrite your code to produce the intended results in a different but equivalent way.
- Click the "Microsoft Product Support Knowledge Base Link" at the bottom of this page to find advice for workarounds.

See Also

Troubleshooting Your Scripts | JScript Reference

JS0053: File not found

The program cannot find the file that it attempted to access.

To correct this error

- Make sure that the path and file name are correct.
- Make sure that the file exists.

See Also

Troubleshooting Your Scripts | JScript Reference

JS0424: Object required

The program is attempting to use something that is not an object in a context where an object is required.

To correct this error

- Make sure that an object exists in this context.

See Also

Troubleshooting Your Scripts | JScript Reference | Object Object | ActiveXObject Object

JS0429: Can't create object

Although the program is attempting to create a new object, it cannot create the object. This may occur if the application that provides the object is unavailable or if the application does not provide that particular object.

To correct this error

- Make sure that the application is available.
- Make sure the appropriate application is used to provide the object.

See Also

Troubleshooting Your Scripts | JScript Reference | ActiveXObject Object

JS0438: Object doesn't support this property or method

A property or method that the program is attempting to access is not a member of an object.

To correct this error

- Make sure that only valid properties and methods of an object are accessed.

See Also

Troubleshooting Your Scripts | JScript Reference | ActiveXObject Object

JS0445: Object doesn't support this action

An object is used for an action that the object does not support.

To correct this error

- Make sure that only the valid actions for the object are used.

See Also

Troubleshooting Your Scripts | JScript Reference | ActiveXObject Object

JS0451: Object is not a collection

The program is attempting to create a new **Enumerator** object, but the argument passed to the constructor is not a collection.

> **Note** Elements of a collection can be accessed directly in JScript .NET. For more information, see Enumerator Object.

To correct this error

- Make sure that only collections are used to construct **Enumerator** objects.

See Also

Troubleshooting Your Scripts | JScript Reference | Enumerator Object

JS1002: Syntax error

A JScript statement violates one or more of JScript's grammatical rules.

To correct this error

- Double-check the program's syntax on the line number indicated.
- Look for misdirected parentheses or braces.

See Also

Error Object | JScript Reference

JS1003: Expected ':'

An expression uses the ternary conditional operator but does not include the colon between the second and third operands. The ternary (three operands) conditional operator requires a colon between the second (true) and third (false) operands.

To correct this error

- Insert a colon between the second and third operands.

See Also

Conditional (Ternary) Operator (?:) | Operators | JScript Reference

JS1004: Expected ';'

Either more than one statement is on a line and the statements are not separated by semicolons, or a **for** statement is on a line without semicolons separating the initialization, test, and increment in the header.

Semicolons are used to terminate statements. Although several statements may be on a single line, each statement must be delimited from the next with a semicolon.

Semicolons are also used to separate the initialization, test, and increment expressions in the header of a **for** loop.

To correct this error

- Mark the end of each statement with a semicolon.
- Make sure function calls use parentheses properly.
- Make sure there are semicolons inside the header of a **for** loop.
- Make sure there is an = in an assignment.

See Also

JScript Language Features | JScript Reference | for Statement

JS1005: Expected '('

An expression that is intended to be enclosed within a set of parentheses is missing the opening parenthesis. Some expressions must be enclosed within a set of opening and closing parentheses. For example, the following **for** statement has correct parenthesis placement:

```
for (initialize; test; increment) {
   statement;
}
```

To correct this error

- Add the left parenthesis to the evaluation expression.

See Also

JScript Reference

JS1006: Expected ')'

An expression that is intended to be enclosed within a set of parentheses is missing the closing parenthesis. Some expressions must be enclosed within a set of opening and closing parentheses. For example, the following **for** statement has correct parenthesis placement:

```
for (initialize; test; increment) {
   statement;
}
```

To correct this error

- Add the right parenthesis to the evaluation expression.

See Also

JScript Reference

JS1007: Expected ']'

A reference to an array element does not include the right bracket. Any expression that refers to an array element must include both opening and closing brackets.

To correct this error

- Add the right bracket to the expression that refers to the array element.

See Also

Using Arrays | Array Object

JS1008: Expected '{'

The left brace that marks the beginning of the function body, class member block, interface member block, or enumeration block is missing. Code that comprises the body of a function, even if it is a single line, must be contained within left and right braces.

Notice that the use of braces in loops is less strict than for function bodies and member blocks.

To correct this error

- Add the left brace that marks the beginning of the function body.

See Also

Function Object | function Statement | class Statement | interface Statement | enum Statement | JScript Reference

JS1009: Expected '}'

The right brace that marks the end of the function body, class member block, interface member block, enumeration block, loop, block of code, or object initializer is missing. An example of this error would be a **for** loop with only the left brace marking the body of the loop.

To correct this error

- Add the right brace that marks the end of the function, class, interface, enumeration, loop, block, or object initializer.

See Also

Function Object | function Statement | class Statement | interface Statement | enum Statement | Controlling JScript Program Flow | JScript Reference

JS1010: Expected identifier

An identifier is missing in a context where one is required. An identifier can be:

- a variable,
- a property,
- an array,
- a function name.

To correct this error

- Change the expression so an identifier appears to the left of the equal sign.

See Also

JScript Reference | Using Arrays

JS1011: Expected '='

A variable that is to be used with conditional compilation statements does not include an equal sign between the variable and the assigned value.

To correct this error

- Add an equal sign. For example:

```
@set @myvar1 = 1
```

See Also

Conditional Compilation | Conditional Compilation Variables

JS1014: Invalid character

An identifier includes a character (or characters) not recognized as valid by the JScript compiler. Valid characters use the following rules:

- The first character must be an ASCII letter (either uppercase or lowercase), an underscore (_), or a dollar sign ($).
- Subsequent characters can be ASCII letters, numbers, underscores, or dollar signs.
- The identifier name cannot be a reserved word.

To correct this error

- Avoid using characters that are not part of the JScript language definition.

See Also

JScript Variables and Constants | String Data | Data Types

JS1015: Unterminated string constant

A string constant is missing a closing quotation mark. String constants must be enclosed within a pair of quotation marks.

> **Note** You can use matching pairs of single or double quotation marks. Double quotation marks can be contained within strings surrounded by single quotation marks, and single quotation marks can be contained within strings surrounded by double quotation marks.

To correct this error

- Add the closing quotation mark to the end of the string.

See Also

String Object | toString Method

JS1016: Unterminated comment

A multi-line comment block is not properly terminated. Multi-line comments must begin with a "/*" combination and end with the reverse "*/" combination.

Following is an example of correct multi-line comment usage:

```
/* This is a comment
This is another part of the same comment.*/
```

To correct this error

- Be sure to terminate multi-line comments with "*/".

See Also

Comment Statements

JS1018: 'return' statement outside of function

A **return** statement is within the global scope of the code or from the body of a class or package. The **return** statement should only appear within the body of a function.

To correct this error

- Remove the **return** statement.

See Also

return Statement | Function Object | caller Property

JS1019: Can't have 'break' outside of loop

The **break** keyword appears outside a loop. The **break** keyword is used to terminate a loop or **switch** statement. It must be embedded in the body of a loop or **switch** statement.

To correct this error

- Make sure the **break** keyword appears inside an enclosing loop or switch statement.

See Also

break Statement | Controlling JScript Program Flow | Troubleshooting Your Scripts | JScript Reference

JS1020: Can't have 'continue' outside of loop

The **continue** statement is outside a loop. The **continue** statement can be used only within the body of a:

- **do-while** loop,
- **while** loop,
- **for** loop,
- **for/in** loop.

To correct this error

- Make sure the **continue** statement appears within the body of a:
    - **do-while** loop,
    - **while** loop,
    - **for** loop,
    - **for/in** loop.

See Also

continue Statement | Controlling JScript Program Flow | Troubleshooting Your Scripts | JScript Reference

JS1023: Expected hexadecimal digit

Code includes an incorrect Unicode escape sequence, or a non-hexadecimal character is the first character in a hexadecimal literal.

Unicode escape sequences begin with \u, followed by exactly four hexadecimal digits. Hexadecimal literals begin with 0x, followed by any number of hexadecimal digits. Hexadecimal digits can contain only the numbers 0-9, the uppercase letters A-F, and the lowercase letters a-f. The following example demonstrates a correctly formed Unicode escape sequence.

```
z = "\u1A5F";
```

The following example demonstrates a correctly formed hexadecimal literal.

```
k = 0x3E8;
```

To correct this error

- Be sure that hexadecimal numbers contain only the numbers 0-9, the uppercase letters A-F, and the lowercase letters a-f.
- Be sure that the Unicode escape sequence contains four digits.

Note If you want to use the literal text \u in a string, then use two backslashes - (\\u) — one to escape the first backslash.

See Also

JScript Reference | Data Types

JS1024: Expected 'while'

A **do while** loop does not include the **while** condition. A **do** statement must have a corresponding **while** test at the end of the code block.

To correct this error

- Include the **while** test statement after the closing curly brace.

See Also

while Statement | Controlling JScript Program Flow | JScript Reference

JS1025: Label redefined

A new label uses the name of an existing label. Within a specified scope, labels must be unique.

To correct this error

- Ensure that all label names are unique within their respective scopes.

See Also

Labeled Statement | switch Statement | break Statement | continue Statement

JS1026: Label not found

Code references a label that does not exist. Within a specified scope, labels must be unique.

To correct this error

- Check the spelling of label names.
- Ensure that all label references are made to labels that have been defined in the current scope (this includes forward definitions).

See Also

Labeled Statement | switch Statement | break Statement | continue Statement

JS1027: 'default' can only appear once in a 'switch' statement

A switch statement uses a **default** case statement more than once. The default case must always be the last case statement in a switch statement (it is the fall-through case).

To correct this error

- Use only one default case statement in your switch statement.

See Also

switch Statement | Controlling JScript Program Flow | JScript Reserved Words

JS1028: Expected identifier or string

An incorrect literal syntax is used to declare an object literal. The properties of an object literal must be either an identifier or a string. An object literal (also called an "object initializer") consists of a comma-separated list of property:value pairs, all enclosed within brackets. For example:

```
var point = {x:1.2, y:-3.4};
```

To correct this error

- Ensure that the literal syntax is correct.

See Also

Comma Operator (,)

JS1029: Expected '@end'

A conditionally compiled block of code does not end with an **@end** statement. JScript statements can be conditionally compiled by enclosing them within an **@if/@end** block.

To correct this error

- Add the corresponding **@end** statement.

See Also

Conditional Compilation | Conditional Compilation Variables| @if Statement

JS1030: Conditional compilation is turned off

Code uses a conditional compilation variable, but conditional compilation is not turned on. Turning on conditional compilation tells the JScript compiler to interpret identifiers beginning with @ as conditional compilation variables. You do this by beginning your conditional code with the statement:

```
/*@cc_on @*/
```

To correct this error

- Add the following statement to the beginning of your conditional code:

  ```
  /*@cc_on @*/
  ```

See Also

Conditional Compilation | Conditional Compilation Variables | @cc_on Statement | @if Statement | @set Statement

JS1031: Expected constant

A variable is in an **@if** (conditional compilation) test statement. Only literals and conditional compilation variables (both of which are constant at the time of compilation) are allowed in a conditional compilation test statement.

To correct this error

- Replace the variable with a literal.
- Replace the variable with a conditional compilation variable.

See Also

Conditional Compilation | Conditional Compilation Variables | @cc_on Statement | @if Statement | @set Statement

JS1032: Expected '@'

A variable that is intended to be used with conditional compilation statements uses the **@set** statement but does not have an at sign "**@**" before the variable name.

To correct this error

- Add an at sign "**@**" immediately before the variable name. For example:

  ```
  @set @myvar = 1
  ```

See Also

@set Statement | Conditional Compilation | Conditional Compilation Variables

JS1033: Expected 'catch'

An exception handling **try** block does not include the associated **catch** statement. To function properly, the exception handling mechanism requires that the code that might fail, along with the code that will not execute if an exception occurs, be wrapped inside a **try** block. Exceptions are thrown from within the **try** block using the **throw** statement and are caught outside the **try** block with one or more **catch** statements.

To correct this error

- Add the associated **catch** block.
- Try using a **finally** block instead of a **catch** block.

See Also

try...catch...finally Statement | Error Object

JS1034: Unmatched 'else'; no 'if' defined

An **else** statement is missing a matching **if** statement. The **if** statement is followed by a statement or compound statement and subsequently followed by the optional **else** statement. That is the only context in which the else statement can appear.

A compound statement is explicitly surrounded with braces. The compiler ignores tabbing conventions, which help improve the readability of the code.

To correct this error

- Enclose the code that follows the **if** statement in braces.
- Add an **if** statement before the **else** statement.

See Also

if...else Statement | Troubleshooting Your Scripts | JScript Reference

JS1100: Expected ','

A required comma is missing between parameters in a function declaration.

To correct this error

- Use a comma to separate each parameter in the function declaration.

See Also

Troubleshooting Your Scripts | JScript Reference | function Statement

JS1101: Visibility modifier already defined

A visibility modifier is applied more than once to an expression. Additional applications of the modifier have no effect.

To correct this error

- Reduce the number of times the visibility modifier is applied to one.

See Also

Troubleshooting Your Scripts | JScript Reference | JScript Modifiers

JS1102: Invalid visibility modifier

A visibility modifier exists in an inappropriate context. This can occur when applying a modifier to an expression that cannot take the modifier or when using a modifier for a member of a class or interface that already has an incompatible modifier.

To correct this error

- Remove the visibility modifier or use an alternative modifier.
- Make sure that visibility modifiers applied to class members match the visibility of the class.

See Also

Troubleshooting Your Scripts | JScript Reference | JScript Modifiers

JS1103: Missing 'case' or 'default' statement

The first line of code inside a **switch** statement block does not begin with a **case** or **default** keyword.

To correct this error

- Add a **case** or **default** keyword after the **switch** statement.

See Also

switch Statement | Troubleshooting Your Scripts | JScript Reference

JS1104: Unmatched '@end'; no '@if' defined

An **@end** statement is missing a matching **@if** statement. Since the **@end** statement terminates an **@if**...**@end** block, every **@if** statement must have a matching **@end** statement, and vice versa.

To correct this error

- Make sure an **@if** statement precedes each **@end** statement.

See Also

@if...@elif...@else...@end Statement | Troubleshooting Your Scripts | JScript Reference

JS1105: Unmatched '@else'; no '@if' defined

An **@else** statement is missing a matching **@if** statement. The **@else** statement must appear only inside an **@if**...**@end** block.

To correct this error

- Make sure the **@else** statement appears only inside an **@if**...**@end** block.

See Also

@if...@elif...@else...@end Statement | Troubleshooting Your Scripts | JScript Reference

JS1106: Unmatched '@elif'; no '@if' defined

An **@elif** statement is missing a matching **@if** statement. The **@elif** statement must appear only inside an **@if**...**@end** block.

To correct this error

- Make sure the **@elif** statement appears only inside an **@if**...**@end** block.

See Also

@if...@elif...@else...@end Statement | Troubleshooting Your Scripts | JScript Reference

JS1107: Expecting more source characters

A line of code ends before an expression is closed. This error will occur if you are required to have both the start and end of an expression on the same line and you have split the line into two or more lines.

To correct this error

- Provide the matching end for the expression on the same line as it starts.

See Also

Troubleshooting Your Scripts | JScript Reference

JS1108: Incompatible visibility modifier

A visibility modifier is applied to a class or interface that cannot take that modifier or to a member of a class or interface defined with an incompatible overall modifier.

To correct this error

- Remove the visibility modifier or use an alternative modifier.
- Make sure that visibility modifiers applied to class members match the visibility of the class.

See Also

Troubleshooting Your Scripts | JScript Reference | JScript Modifiers | Class-based Objects

JS1109: Class definition not allowed in this context

A class declaration exists in an inappropriate context. Class declarations are allowed only in the main program block, inside other classes, inside packages, or in functions.

To correct this error

- Define the class in the main program block, inside another class, inside a package, or in a function.

See Also

Troubleshooting Your Scripts | JScript Reference | class Statement

JS1110: Expression must be a compile time constant

An expression that is undefined at compile time exists in a context that allows only compile-time constants.

To correct this error

- Use only compile-time constants in this context.

See Also

Troubleshooting Your Scripts | JScript Reference

JS1111: Identifier already in use

The name of an identifier repeats in an expression.

To correct this error

- If both instances refer to the same identifier, remove the second definition of the identifier.
- If the identifiers should be different, give them unique names.

See Also

JScript Variables and Constants | Troubleshooting Your Scripts | JScript Reference

JS1112: Type name expected

A required type name is misspelled or omitted. Type names are required following a colon in the definition of a variable, constant, function, or parameter.

To correct this error

- Make sure that a valid data type identifier is used where a type name is expected.

See Also

Type Annotation | Troubleshooting Your Scripts | JScript Reference

JS1113: Only valid inside a class definition

A directive or keyword that is only valid inside a class definition exists outside a class definition.

To correct this error

- Remove the term.
- Make sure the code is inside a class definition.

See Also

Class-based Objects | Troubleshooting Your Scripts | JScript Reference

JS1114: Unknown position directive

The code passed an invalid argument to the position directive. Valid arguments are **end**, **file=**, **line=,** and **column=.**

To correct this error

- Make sure that only valid arguments are used with the position directive.

See Also

Troubleshooting Your Scripts | JScript Reference | Directives

JS1115: Directive may not be followed by other code on the same line

Extra code follows a directive.

To correct this error

- Split the line in two lines after the directive.

See Also

Troubleshooting Your Scripts | JScript Reference | Directives

JS1118: Wrong debugger directive or wrong position for the directive

Code passed an invalid argument to the **@debug** directive. Valid arguments are **on**, and **off**.

- or -

Code contains a disallowed directive.

To correct this error

- Make sure the argument passed to the **@debug** directive is either **on** or **off**.
- - or -
- Move the directive to another location.

See Also

Troubleshooting Your Scripts | JScript Reference | Directives

JS1119: Position directives cannot be nested

Code contains nested **@position** directives.

To correct this error

- Close one **@position** directive before starting another one.

See Also

Troubleshooting Your Scripts | JScript Reference | @position Directive

JS1120: Circular definition

The code defines a class or interface in terms of itself. This can happen if the code contains two classes, each one extending the other.

To correct this error

- Make sure when extending a class or interface that the base class or interface does not depend on the class or interface being defined.

See Also

Class-based Objects | Troubleshooting Your Scripts | JScript Reference

JS1121: Deprecated

The code contains a deprecated expression. A preferred alternative expression performs the same task. Use the new approach, since the support for deprecated expressions may be dropped in later versions of the language.

To correct this error

- Use the encodeURI Method instead of the escape Method.
- Use the getFullYear Method instead of the getYear Method.
- Use the setFullYear Method instead of the setYear Method.
- Use the substring Method instead of the substr Method.
- Use the toUTCString Method instead of the toGMTString Method.
- Use the decodeURI Method instead of the unescape Method.

See Also

Troubleshooting Your Scripts | JScript Reference

JS1122: It is invalid to use 'this' in current context

The code uses **this** in a **static** class or member function.

To correct this error

- Replace **this** with a qualified reference to a particular instance of the class.
- Remove the **static** modifier.

See Also

Class-based Objects | Troubleshooting Your Scripts | JScript Reference

JS1123: Not accessible from this scope

A statement is attempting to access a member of an object, but that member has visibility modifiers that prevent it from being accessed from the current scope. For example, a private field cannot be accessed from outside a class.

To correct this error

- Access the member by other means, such as though a public method.
- Change the modifiers of the member that the code accesses.

See Also

Class-based Objects | Troubleshooting Your Scripts | JScript Reference

JS1124: Only a constructor function can have the same name as the class it appears in

A method that is not a constructor has the same name as its class.

To correct this error

- Make the method a constructor by eliminating the return type and the **return** statements.
- Rename the method.

See Also

Class-based Objects | Troubleshooting Your Scripts | JScript Reference

JS1128: Class must provide implementation

A **final** method or a method in a **final** class does not have an associated body.

To correct this error

- Provide a body for the method.
- Remove the **final** modifier.

See Also

Class-based Objects | JScript Modifiers | Troubleshooting Your Scripts | JScript Reference

JS1129: Interface name expected

A defined class implements a non-existent interface.

To correct this error

- Provide the name of a valid interface following the **implements** keyword in the class definition.

See Also

Class-based Objects | Troubleshooting Your Scripts | JScript Reference

JS1133: Catch clause will never be reached

A **catch** block follows another **catch** block that catches all errors. A **catch** statement will catch all errors when the argument of the statement does not have a specific type.

To correct this error

- Examine all **catch** statements to make sure that **catch** statements only catch each type once and that the last **catch** statement catches untyped errors.

See Also

try...catch...finally Statement | Troubleshooting Your Scripts | JScript Reference

JS1134: Type cannot be extended

An expression attempts to extend a type or class that has the **final** modifier.

To correct this error

- Do not attempt to extend the type or class.
- Remove the **final** modifier from the class.

See Also

Class-based Objects | JScript Modifiers | Troubleshooting Your Scripts | JScript Reference

JS1135: Variable has not been declared

An expression includes a variable that has not been defined with a **var** statement or that has a misspelled name, and the code is compiled in fast mode. Programs compiled in fast mode must have all variables explicitly declared.

Fast mode can be turned off for programs compiled with the command-line compiler.

To correct this error

- Make sure to define all variables.
- Make sure all identifiers are spelled correctly.
- Compile the program with the **/fast-** option to turn fast mode off. (For command-line compilation only.)

See Also

JScript Variables and Constants | Troubleshooting Your Scripts | JScript Reference | /fast

JS1136: Leaving variables uninitialized is dangerous and makes them slow to use

Leaving variable uninitialized is dangerous and makes them slow to use. Did you intend to leave this variable uninitialized?

A variable that is defined with the **var** statement does not have a specified type or has not been initialized before using it.

To correct this error

- Use type annotation for the variables.
- Initialize all variables before use.

See Also

JScript Variables and Constants | Troubleshooting Your Scripts | JScript Reference

JS1137: This is a new reserved word and should not be used as an identifier

The code uses a new reserved word as the name of an identifier.

To correct this error

- Change the name of the identifier to exclude reserved words.

See Also

JScript Reserved Words | Troubleshooting Your Scripts | JScript Reference

JS1140: Not allowed in a call to a base class constructor

The code attempted to pass a property of the current class to the base class constructor. This is not allowed because the properties of the current class do not exist until after the base class is constructed.

To correct this error

- Make sure no properties of the class that is being constructed are passed to the base class constructor.

See Also

Class-based Objects | Troubleshooting Your Scripts | JScript Reference

JS1141: This constructor or property getter/setter method is not meant to be called directly

An expression called the constructor method of a class directly.

- - or -

An expression called the getter or setter method of a property directly.

- Neither method can be called directly.

To correct this error

- Do not call the constructor method.

  - or -

- Access the property using the "." syntax.

See Also

Troubleshooting Your Scripts | JScript Reference | function get Statement | function set Statement | Class-based Objects

JS1142: The get and set methods of this property do not match each other

The code defines **get** and **set** accessors for a property. However, the return data type of the **get** accessor is not the same as the parameter type of the **set** accessor.

To correct this error

- Make sure that the return type of the **get** accessor matches the argument type of the **set** accessor.

See Also

function get Statement | function set Statement | Troubleshooting Your Scripts | JScript Reference

JS1143: A custom attribute class must derive from System.Attribute

A class that is used as a custom attribute is not derived from **System.Attribute**. Only classes that have **System.Attribute** as a base class can be used as custom attributes.

To correct this error

- Make sure that the custom attribute class has **System.Attribute** as a base class.

See Also

Troubleshooting Your Scripts | JScript Reference

JS1144: Only primitive types are allowed in a custom attribute

An expression attempted to pass something other than a primitive type to a custom attribute constructor, or an expression is using something that is not an attribute where an attribute is expected.

To correct this error

- Make sure that the expression is using an attribute and that it is passing only primitive types to the custom attribute constructor.

See Also

Writing Custom Attributes | Troubleshooting Your Scripts | JScript Reference

JS1146: Unknown custom attribute class or constructor

The code uses something that is not an attribute or custom attribute constructor where an attribute is expected.

To correct this error

- Make sure the code uses an attribute or custom attribute constructor in this context.

See Also

Writing Custom Attributes | Troubleshooting Your Scripts | JScript Reference

JS1148: There are too many arguments

There are too many arguments. The extra arguments will be ignored.

An expression passes more arguments to a function or method than were specified in the definition of the function or method.

To correct this error

- Use the correct number of arguments for a function or method.
- Check that an expression does not have extra arguments.

See Also

function Statement | Troubleshooting Your Scripts | JScript Reference

JS1149: The with statement has made the use of this name ambiguous

An expression uses the **with** statement to access a class that has a member with the same name as an identifier in the current scope. The compiler cannot distinguish which identifier to access.

To correct this error

- Rename the member of the class or the identifier in the current scope.
- Avoid using the **with** statement.

See Also

with Statement | Troubleshooting Your Scripts | JScript Reference

JS1150: The presence of eval has made the use of this name ambiguous

The program uses an **eval** statement and is compiled with fast mode turned off.

When fast mode is turned off, the **eval** statement allows new variables to be declared at runtime with local scope. These new variables can shadow the global variables, which makes any reference to a variable not explicitly defined in the local scope potentially ambiguous.

To correct this error

- Compile with the **fast** option on.
- Avoid using the **eval** statement.
- Use only the variables available in the current local scope.

See Also

Troubleshooting Your Scripts | JScript Reference | eval Method | /fast

JS1151: Object does not have such a member

An expression references a member of a class-based object, but the object does not have a member with that name.

If the code is contained in an ASP.NET page, the code might define a constructor function within a **<script runat="server">** block that uses the **this** statement. The **expando** modifier should be applied to every constructor function definition in a **<script runat="server">** block.

For example, the following code for an ASP.NET page uses a constructor function marked with the **expando** modifier.

```
<script runat="server">
expando function Person(name) {
   // If the expando modifier was not applied to the definition of Person,
   // the this statment in the following line of code would generate error
   // JS1151

   this.name = name;
}
</script>

<%
var fred = new Person("Fred");
Response.Write(fred.name);
%>
```

To correct this error

- Make sure that the expression references an existing member of the class-based object and that the member name is correctly spelled.
- Make sure to apply the **expando** modifier to each constructor function declaration in a **<script runat="server">** block.

See Also

Troubleshooting Your Scripts | JScript Reference | Class-based Objects | this Statement | expando Modifier

JS1152: Cannot define the property Item on an Expando class

Cannot define the property Item on an Expando class. Item is reserved for the expando fields.

A member of an expando class is named **Item**. This is not allowed because it leads to a conflict with the **Item** property that is implicitly defined for expando classes.

To correct this error

- Rename the class member named **Item**.

See Also

Troubleshooting Your Scripts | JScript Reference | Class-based Objects | expando Modifier

JS1153: Cannot define get_Item or set_Item on an Expando class

Cannot define get_Item or set_Item on an Expando class. Methods reserved for the expando fields.

A member of an expando class is named **get_Item** or **set_Item**. This is not allowed because it leads to a conflict with the **get_Item** or **set_Item** properties that are implicitly defined for expando classes.

To correct this error

- Rename the class member named **get_Item** or **set_Item**.

See Also

Troubleshooting Your Scripts | JScript Reference | Class-based Objects | expando Modifier

JS1155: Base class defines get_Item or set_Item, cannot create expando class

Base class defines get_Item or set_Item, cannot create expando class. Methods reserved for the expando fields.

An expando class extends a base class with a member named **get_Item** or **set_Item**. This is not allowed because it leads to a conflict with the **get_Item** or **set_Item** properties implicitly defined for expando classes.

To correct this error

- Rename the base class member named **get_Item** or **set_Item**.
- Do not inherit from the base class.
- Remove the **expando** modifier from the class definition.

See Also

Troubleshooting Your Scripts | JScript Reference | Class-based Objects | expando Modifier

JS1156: A base class is already marked expando; current specification will be ignored

A class marked with the **expando** modifier extends an expando base class. However, classes derived from an expando base class are automatically expando; you do not need to add the **expand** modifier explicitly.

To correct this error

- Remove the **expando** modifier from the derived class.

See Also

Troubleshooting Your Scripts | JScript Reference | Class-based Objects | expando Modifier

JS1157: An abstract method cannot be private

A method has both **abstract** and **private** modifiers. This is not allowed since private methods are accessible from within the class, but abstract methods are inherited from outside the class.

To correct this error

- Remove either the **abstract** or **private** modifier from the method declaration.

See Also

Troubleshooting Your Scripts | JScript Reference | Class-based Objects | abstract Modifier | private Modifier

JS1158: Objects of this type are not indexable

The code attempted to index an element of an object, but the data type of the object does not support indexing.

Elements of indexable objects, such as arrays and JScript objects, are accessed using the [] notation.

To correct this error

- Change the data type of the object.
- Remove the index accessor.

See Also

Troubleshooting Your Scripts | JScript Reference | JScript Objects | Typed Arrays

JS1159: Syntax error. Use 'static classname {...}' to define a class initializer

A code block immediately follows a modifier inside a class. JScript modifiers can only be applied to class members. Two likely scenarios might produce this result:

- You meant to define a class initializer but left out the class name.
- You meant to define a method or property accessor but left out the **function**, **function get**, or **function set** statement.

To correct this error

- If you meant to define a class initializer, use the correct syntax for **static** statement.
- If you meant to define a method or property accessor, use the correct syntax for the **function**, **function get**, or **function set** statement.

See Also

Troubleshooting Your Scripts | JScript Reference | static Statement | JScript Modifiers | function Statement | function get Statement | function set Statement

JS1160: The list of attributes does not apply to the current context

The code specifies a list of attributes that does not apply in the current context.

To correct this error

- Make sure that you only use attributes that apply in the current context.

See Also

Troubleshooting Your Scripts | JScript Reference

JS1161: Only classes are allowed inside a package

The code declares a function, variable, or constant inside a package. Only classes and interfaces can be defined in a package.

To correct this error

- Remove all definitions in the package that are not for a class or interface.

See Also

Troubleshooting Your Scripts | JScript Reference | package Statement | JScript Objects

JS1162: Expando classes should not implement IEnumerable

Expando classes should not implement IEnumerable. The interface is implicitly defined on expando classes.

An expando class implements **IEnumerable** explicitly. This is not necessary, since classes with the **expando** modifier implicitly implement **IEnumerable**.

To correct this error

- Do not implement **IEnumerable** explicitly.

See Also

Troubleshooting Your Scripts | JScript Reference | expando Modifier | Class-based Objects | IEnumerable Interface

JS1163: The specified member is not CLS compliant

The program contains the **CLSCompliantAttribute** attribute, and the compiler has detected a class member that is not Common Language Specification (CLS) compliant. Some possible causes of this error are:

- The member name is not CLS compliant. CLS compliant names cannot start with an underscore (_), contain a dollar sign ($), or differ only in capitalization from the name of another public member.
- If the member is a public method, data types that are not available in the common language runtime are used to type annotate the parameters or return type.
- If the member is a field or property, data types that are not available in the common language runtime are used to type annotate field or property.

There are several reasons a data type may not to be available in the common language runtime.

- The type is defined within the class but it is not publicly accessible.
- The type is defined but not marked as CLS compliant.
- The type is a primitive type that is not CLS compliant. For example, **uint** is a primitive type that is not CLS compliant. The corresponding CLS compliant system type is **System.UInt32**.
- The type is an intrinsic JScript object, none of which are CLS compliant. The commonly used JScript objects, **Array**, **Date**, **Error**, **RegExp**, and **Function**, correspond to the CLS compliant system types, **System.Array**, **System.DateTime**, **System.Exception**, **System.Text.RegularExpressions.RegEx**, and **System.EventHandler**.

To correct this error

- Make sure that the member name does not start with an underscore (_), contain a dollar sign ($), or differ only in capitalization from the name of another member.
- Make sure that the parameters or return types for public methods and the types of public fields and properties are common language runtime data types or publicly accessible classes that have been marked as CLS compliant.

See Also

Troubleshooting Your Scripts | JScript Reference | CLSCompliantAttribute Class | Writing CLS-Compliant Code |
What is the Common Language Specification?

JS1164: Member is not deleteable

The code attempted to **delete** a member of an object that cannot be deleted. Only expando properties (properties that have been dynamically added to an object) can be deleted.

To correct this error

- Do not attempt to **delete** the object member.

See Also

Troubleshooting Your Scripts | JScript Reference | delete Operator | JScript Object Object | expando Modifier

JS1165: Package name expected

A valid package name does not follow an **import** statement.

To correct this error

- Include a valid package name after the **import** statement.

See Also

Troubleshooting Your Scripts | JScript Reference | import Statement | package Statement

JS1169: Expression has no effect

An expression returns a value that is never used.

To correct this error

- Remove the expression.
- Use the value of the expression as the argument for a function or operator.

See Also

Troubleshooting Your Scripts | JScript Reference | JScript Expressions

JS1170: Hides another member declared in the base class

A member in a derived class hides (redefines the meaning of) a field, class, interface, or enumeration defined in a base class. This is not allowed; only methods and properties may be hidden.

To correct this error

- Make sure that no derived class member hides a base-class field, class, interface, or enumeration definition.

See Also

Troubleshooting Your Scripts | JScript Reference | Class-based Objects

JS1171: Cannot change visibility specification of a base method

A method in a derived class overrides a base-class method, and the visibility modifiers of the two methods are different. This is not allowed because the visibility of a base-class member cannot be changed.

To correct this error

- Change the visibility modifier of the derived-class method to match the visibility of the base class method.

See Also

Troubleshooting Your Scripts | JScript Reference | JScript Modifiers

JS1172: Method hides abstract method in a base class

A method in a derived class defined with the **hide** modifier has the same name as an abstract method in the base class. This is not allowed because an abstract method requires an implementation from the derived class, and hiding a method prevents the implementation.

To correct this error

- Remove the **abstract** modifier from the base class method and provide the method with an implementation.
- Remove the **hide** modifier from the derived class method.

See Also

Troubleshooting Your Scripts | JScript Reference | JScript Modifiers

JS1173: Method matches a method in a base class

Method matches a method in a base class. Specify 'override' or 'hide' to suppress this message.

A method in a derived class that was defined without a version-safe modifier matches a base-class method. Furthermore, the program was compiled with the **/versionsafe** option. When compiling with the **/versionsafe** option, every method that matches a base-class method must use a version-safe modifier (either **hide** or **override**).

To correct this error

- Apply the appropriate version-safe modifier to the method declaration.

See Also

Troubleshooting Your Scripts | JScript Reference | /versionsafe | JScript Modifiers | hide Modifier | override Modifier

JS1174: Method matches a non-overridable method in a base class

Method matches a non-overridable method in a base class. Specify 'hide' to suppress this message.

A base-class method with the **final** modifier matches a method in a derived class. In addition, the derived-class method has the **override** modifier or the code is being compiled with the **/versionsafe** option. A final method cannot be overridden, and you must explicitly specify the **hide** modifier for the derived-class method if you are using the **/versionsafe** option.

To correct this error

- Use the **hide** modifier for the method in the derived class.

See Also

Troubleshooting Your Scripts | JScript Reference | /versionsafe | JScript Modifiers | hide Modifier | override Modifier | final Modifier

JS1175: There is no member in a base class to hide

A derived class method has the **hide** modifier, but there is no matching method in the base class. You cannot hide a method that does not exist.

To correct this error

- Remove the **hide** modifier from the method declaration.

  - or -

- Add a matching method to the base class.

See Also

Troubleshooting Your Scripts | JScript Reference | hide Modifier

JS1176: Method in base has a different return type

A derived class implements an interface or extends a base class. The derived class has a method that has the same name and parameter list as a method in the interface or base class, but the return type is different. Two methods cannot have the same name and parameter list but different return types.

To correct this error

- Rename the function in the derived class.
- Change the return types or the methods so they match in the derived class and interface or base class.

See Also

Troubleshooting Your Scripts | JScript Reference | Class-based Objects

JS1177: Clashes with property

A field or method defined in a derived class has the same name as a property in the base class. This creates an ambiguity when using the name to refer to a member of the derived class and is not allowed.

To correct this error

- Rename either the base class property or the derived class member.

See Also

Troubleshooting Your Scripts | JScript Reference | function set Statement | function get Statement

JS1178: Cannot use 'override' and 'hide' together

A method is defined with both the **override** and **hide** modifiers. However, each class member can have only one version-safe modifier.

To correct this error

- Remove either **override** or **hide** from the method declaration.

See Also

Troubleshooting Your Scripts | JScript Reference | JScript Modifiers | hide Modifier | override Modifier

JS1179: Invalid Option

An expression that uses the **@option** directive does not include a valid option. The **@option** directive is not currently supported.

To correct this error

- Do not use the **@option** directive.

See Also

Troubleshooting Your Scripts | JScript Reference

JS1180: There is no matching method in a base class to override

A derived-class method that uses the **override** modifier has no matching method in the base class. You cannot override a method that does not exist.

To correct this error

- Remove the **override** modifier from the method declaration.

  - or -

- Add a matching method to the base class.

See Also

Troubleshooting Your Scripts | JScript Reference | override Modifier | Class-based Objects

JS1181: Not valid for a constructor

A class constructor uses an invalid modifier. Only the visibility modifiers (**public**, **private**, **protected**, and **internal**), the version-safe modifiers (**hide** and **override**), or the **final** and **expando** modifiers can be applied to a constructor.

To correct this error

- Use a valid modifier for the constructor.

See Also

Troubleshooting Your Scripts | JScript Reference | Class-based Objects | JScript Modifiers

JS1182: Cannot return a value from a constructor or void function

An expression attempted to return a value from a constructor or a **void** function. Constructor functions automatically return a pointer to the constructed function; they do not return a value. Functions that are defined with the **void** return type cannot return a value.

To correct this error

- Remove the return statement.
- Specify a non-**void** return type for the function.

See Also

Troubleshooting Your Scripts | JScript Reference | function Statement | return Statement

JS1183: More than one method or property matches this argument list

An expression that is calling an overloaded method or property does not find an exact match to the types of arguments that are passed. In this situation, the compiler attempts to determine which overloaded function requires the least number of data type coercions of the arguments. This error indicates that the compiler found more than one function that matches the arguments with the same number of data type coercions.

To correct this error

- Check the data types accepted by the overloaded function and make sure the data types of the arguments match only one overloaded function.

See Also

Troubleshooting Your Scripts | JScript Reference

JS1184: More than one constructor matches this argument list

An expression that is calling an overloaded constructor does not find an exact match to the types of arguments that are passed. In this situation, the compiler attempts to determine which overloaded constructor requires the least number of data type coercions of the arguments. This error indicates that the compiler found more than one constructor that matches the arguments with the same number of data type coercions.

To correct this error

- Check the data types accepted by the overloaded constructor and make sure the data types of the arguments match only one overloaded constructor.

See Also

Troubleshooting Your Scripts | JScript Reference JScript Reference

JS1185: Base class constructor is not accessible from this scope

A class extends a base class, and the constructor for the base class is not accessible from the current scope. This can happen if visibility modifiers are used for the base class or the constructor of the base class.

To correct this error

- Use a different visibility modifier for the base class constructor.
- If the base class is defined with the **internal** modifier, define the derived class in the same package as the base class.

See Also

Troubleshooting Your Scripts | JScript Reference | JScript Modifiers

JS1186: Octal literals are deprecated

An octal literal represents a number in the code. In an octal literal, one or more zeros (0) precede a whole number. Decimal or hexadecimal literals should be used instead of octal literals.

To correct this error

- Remove the leading zeros from the number if the number is a decimal number.
- Convert the octal number to a decimal or hexadecimal number.

See Also

Troubleshooting Your Scripts | JScript Reference | Numeric Data

JS1187: Variable might not be initialized

An expression is accessing the value of a variable that was neither initialized nor defined as a specific data type.

To correct this error

- Initialize the variable before using it.
- Declare the variable using type annotation.

See Also

Troubleshooting Your Scripts | JScript Reference | var Statement

JS1188: It is not valid to call a base class constructor from this location

An expression is calling the base class constructor, **super**, from a location other than the first line inside a constructor definition.

To correct this error

- Make sure that the base class constructor is called only from the first line within a constructor declaration.

See Also

Troubleshooting Your Scripts | JScript Reference | super Statement

JS1189: It is not valid to use the super keyword in this way

An expression is using the **super** statement in a static class member. This is not allowed because static members are associated with the class itself, while the **super** statement is used to access base-class members of the current instance of the class.

To correct this error

- Replace **super** with a qualified reference to a particular instance of the base class.
- Remove the **static** modifier.

See Also

Troubleshooting Your Scripts | JScript Reference super Statement | static Modifier

JS1190: It is slow and potentially confusing to leave a finally block this way

It is slow and potentially confusing to leave a finally block this way. Is this intentional?

A statement (either **return** or **break**) causes control of the program to leave the **finally** block. This may produce unintended consequences if there is a **return** or **break** statement in either of the **try** or **catch** blocks.

The code in a **finally** block is always run after the code in the **try** block and (if there is an error) after the code in the **catch** block. For example, if a **return** statement is encountered in a **try** block, the **finally** block is executed before the **return** statement is executed. If there is another **return** statement in the finally block, it will be executed, and the original return statement will not be executed. To avoid this potentially confusing situation, do not use a **return** statement in a **finally** block.

```
function test() {
   try {
      return(5);   // Attempt to return 5.
   } catch(e) {
      print(e);
   } finally {
      return(10);  // This gets run first, returning 10 instead of 5.
   }
}
print(test());     // Prints 10, not 5.
```

To correct this error

- Make sure that the **return** and **break** statements are not used in a **finally** block.
- Move the **return** or **break** statements to immediately follow the **finally** block if the intention is to execute them after the **try** and **catch** blocks.

See Also

Troubleshooting Your Scripts | JScript Reference | try...catch...finally Statement

JS1191: Expected ','. Write 'identifier : Type' rather than 'Type identifier' to declare a typed parameter

A function declaration includes parameters that are not separated with commas or a type annotated parameter that is specified as `Type identifier` instead of `identifier : Type`.

To correct this error

- Make sure all parameters are separated with commas.
- Specify type-annotated parameters with the `identifier : Type` syntax.

See Also

Troubleshooting Your Scripts | JScript Reference | function Statement | JScript Functions

JS1192: Abstract function cannot have body

A function body is associated with a method or property, but the method or property is marked with the **abstract** modifier or is in an interface.

To correct this error

- Remove the function body.
- Change the modifiers.
- Use a class instead of an interface.

See Also

class Statement | interface Statement | JScript Modifiers | Troubleshooting Your Scripts | JScript Reference

JS1193: Expected ',' or ')'

A call to a function, method, or constructor is missing a comma or closing parenthesis.

To correct this error

- Add a comma or closing parenthesis.

See Also

Troubleshooting Your Scripts | JScript Reference | JScript Functions | JScript Objects

JS1194: Expected ',' or ']'

A reference to an array element is missing a comma or closing square bracket.

To correct this error

- Add a comma or closing square bracket.

See Also

Troubleshooting Your Scripts | JScript Reference | JScript Arrays

JS1195: Expected expression

In this context, a value or a reference (the result of an expression) is expected, but an expression has not been provided. The code may include a statement that does not return a value.

To correct this error

- Make sure that an expression is used.

See Also

Troubleshooting Your Scripts | JScript Reference | JScript Expressions

JS1196: Unexpected ';'

A semicolon is inappropriate in this context, or the statement terminated by the semicolon has an error.

To correct this error

- Remove the semicolon.
- Make sure that the statement terminated by the semicolon has no other errors.

See Also

Troubleshooting Your Scripts | JScript Reference | JScript Statements

JS1197: Too many errors

Too many errors. The file might not be a JScript .NET file.

The code has generated too many errors. The most common cause is trying to compile a file that is not a JScript .NET file. Alternatively, the code may simply have a few errors from which the compiler cannot recover, which causes many other errors.

To correct this error

- Make sure the file that the compiler is compiling is a JScript .NET file.
- Fix the first few errors, then recompile.

See Also

Troubleshooting Your Scripts | JScript Reference

JS1198: Syntax error. Write 'var identifier : Type' rather than 'Type identifier' to declare a typed variable

Your code has what appears to be a C-style field declaration of the form `Type identifier`. To declare a field in JScript, use the `var identifier : Type` syntax.

To correct this error

- Declare fields using the `var identifier : Type` syntax.

See Also

Troubleshooting Your Scripts | JScript Reference | var Statement | Class-based Objects

JS1199: Syntax error. Write 'function identifier(...) : Type{' rather than 'Type identifier(...){' to declare a typed function

The code has what appears to be a C-style method declaration of the form `Type identifier(...)`. To declare a method in JScript, use the `function identifier(...) : Type` syntax.

To correct this error

- Declare methods using the `function identifier(...) : Type` syntax.

See Also

Troubleshooting Your Scripts | JScript Reference | function Statement | Class-based Objects

JS1200: Invalid property declaration

Invalid property declaration. The getter must not have arguments and the setter must have one argument.

The code defines a property getter function with one or more parameters, or defines a property setter function with no parameters or more than one parameter. The definition of a getter function must have no parameters, while the setter function must have exactly one parameter.

To correct this error

- Define property getter functions with no parameters.
- Define property setter functions with exactly one parameter.

See Also

Troubleshooting Your Scripts | JScript Reference | function get Statement | function set Statement | Class-based Objects

JS1203: Expression does not have an address

An ampersand (**&**) in the code is followed by an expression that does not have an address. The ampersand should precede only a variable name (which has an address) and should only be used to pass the variable by reference to a function that accepts the parameter by reference.

Passing variables by reference allows the function to change the value of the variable.

> **Note** JScript .NET does not allow functions to be defined with reference parameters. JScript .NET provides the ampersand to allow calls to external objects that take reference parameters.

To correct this error

- Make sure that the ampersand (**&**) precedes a variable name in a call to a function and that the function accepts the parameter by reference.

See Also

Troubleshooting Your Scripts | JScript Reference

JS1204: Not all required arguments have been supplied

The code calls a function or method with fewer arguments than the function or method is defined to accept. Although missing arguments will be given default values by the function or method, it is a good idea to provide values for all expected arguments.

To correct this error

- Make sure that all required arguments are passed to a function or method.

See Also

Troubleshooting Your Scripts | JScript Reference

JS1205: Assignment creates an expando property that is immediately thrown away

The code has an assignment to a non-existent property of an object, and the object does not support expando properties. The compiler attempts to create an expando property, but the properties cannot be added to the object and so the property is discarded.

To correct this error

- Use an object that supports expando properties.
- Do not attempt to add expando properties to objects that do not support them.

See Also

Troubleshooting Your Scripts | JScript Reference | Advanced Class Creation | JScript Object Object

JS1206: Did you intend to write an assignment here?

The code has an assignment operator as the conditional expression for a conditional statement. You may have intended to use an equality or strict equality operator.

To correct this error

- Change the assignment operator (**=**) to the equality operator (**==**) or the strict equality operator (**===**).
- Move the assignment to immediately precede the conditional statement, and then use the left operand of the assignment operator as the conditional expression.

See Also

Troubleshooting Your Scripts | JScript Reference | JScript Conditional Structures

JS1207: Did you intend to have an empty statement for this branch of the if statement?

The code has an **if** statement followed by a semicolon. The semicolon is interpreted as the terminator for an empty statement that is executed when the conditional expression in the **if** statement is true.

To correct this error

- Remove the semicolon.
- Follow the **if** statement with an empty block ({}).

See Also

Troubleshooting Your Scripts | JScript Reference | if...else Statement

JS1208: The specified conversion or coercion is not possible

The code has a type conversion or coercion that cannot be performed. This indicates that the data in the original data type does not have an obvious analogue in the target conversion type.

To correct this error

- Make sure that the data provided is compatible with the data type that is converted or the data type to which it is coerced.

See Also

Troubleshooting Your Scripts | JScript Reference | Type Conversion | Coercion in JScript

JS1209: final and abstract cannot be used together

The code has a class or class member marked with both the **final** and **abstract** modifiers. These modifiers cannot be combined.

To correct this error

- Use either the **final** or **abstract** modifier.

See Also

Troubleshooting Your Scripts | JScript Reference | JScript Modifiers

JS1210: Must be an instance

The code is attempting to access a non-static class member by using the class name. Only static class members are associated with the class itself; non-static members are associated with and accessed through a particular class instance.

To correct this error

- Make sure that non-static members are accessed with a class instance.

See Also

Troubleshooting Your Scripts | JScript Reference | Class-based Objects

JS1212: Cannot be abstract unless the declaring class is marked as abstract

The code has a member that is marked with the **abstract** modifier, but the class of which it is a member is not marked as **abstract**. A class must be marked as **abstract** if at least one member is **abstract**.

To correct this error

- Make sure all classes with **abstract** members are marked as **abstract**.

See Also

Troubleshooting Your Scripts | JScript Reference | JScript Modifiers

JS1213: The base type of an enum must be a primitive integral type

The code has an enumeration that is declared to have an underlying base type that is not a primitive integral type. Valid base types for an enumeration are the integral data types: **int**, **short**, **long**, **byte**, **uint**, **ushort**, **ulong**, and **sbyte**.

To correct this error

- Make sure that the base type for each enumeration is a valid integral data type.

See Also

Troubleshooting Your Scripts | JScript Reference | enum Statement

JS1214: It is not possible to construct an instance of an abstract class

The code attempts to construct an instance of an abstract class with the **new** operator. Classes marked with the **abstract** modifier cannot be instantiated.

To correct this error

- Remove the **abstract** modifier from the class.
- Define a class that extends the abstract class and provides an implementation for each of the abstract methods and properties.
- Do not attempt to instantiate an abstract class.

See Also

Troubleshooting Your Scripts | JScript Reference

JS1215: Converting a JScript Array to a System.Array results in a memory allocation and an array copy

The code converts a JScript **Array** object to a typed array (a **System.Array**).

> **Note** This is accomplished by allocating enough memory to store a copy of the typed array and by copying the elements of the JScript array into the typed array.

Consequently, modifications to the typed array will not be reflected in the JScript array unless the code copies the typed array back to the JScript array after making the modifications.

To correct this error

- Use explicit type conversion to convert the JScript array to the typed array.

See Also

Troubleshooting Your Scripts | JScript Reference | JScript Arrays | Type Conversion

JS1216: Static methods cannot be abstract

A method has both **static** and **abstract** modifiers. This is not allowed since static methods are associated with the class itself, but abstract methods are inherited from outside the class.

To correct this error

- Remove either the **static** or **abstract** modifier from the method declaration.

See Also

Troubleshooting Your Scripts | JScript Reference | JScript Modifiers

JS1217: Static methods cannot be final

A method has both **static** and **final** modifiers. This is not allowed since static methods are already final. Static methods are associated with the class itself, and therefore cannot be overridden.

To correct this error

- Remove either the **static** or **final** modifier from the method declaration.

See Also

Troubleshooting Your Scripts | JScript Reference | JScript Modifiers

JS1218: Static methods cannot override base class methods

A method has both **static** and **override** modifiers. This is not allowed since the static method is associated with the current class, and overriding only works with class instances.

To correct this error

- Remove either the **static** or **override** modifier from the method declaration.

See Also

Troubleshooting Your Scripts | JScript Reference | JScript Modifiers

JS1219: Static methods cannot hide base class methods

A method has both **static** and **hide** modifiers. This is not allowed since the static method is associated with the current class, and **hide** only applies to members associated with class instance.

To correct this error

- Remove either the **static** or **hide** modifier from the method declaration.

See Also

Troubleshooting Your Scripts | JScript Reference | JScript Modifiers

JS1220: Expando methods cannot override base class methods

A method has both **expando** and **override** modifiers. This is not allowed since the expando method is associated with the current class, and **override** only applies to members associated with a class instance.

To correct this error

- Remove either the **expando** or **override** modifier from the method declaration.

See Also

Troubleshooting Your Scripts | JScript Reference | JScript Modifiers

JS1221: A variable argument list must be of an array type

The definition of the function has a parameter array (or variable argument list) that is not type annotated as a typed array. A parameter array must be the last parameter in the function declaration, preceded with an ellipsis (...), and type annotated as a typed array. A parameter array cannot be a JScript **Array** object.

To correct this error

- Type annotate the variable argument list as a typed array.

See Also

Troubleshooting Your Scripts | JScript Reference | function Statement

JS1222: Expando methods cannot be abstract

A method has both **expando** and **abstract** modifiers. Expando methods can never be abstract.

To correct this error

- Remove either the **expando** or **abstract** modifier from the method declaration.

See Also

Troubleshooting Your Scripts | JScript Reference | JScript Modifiers

JS1223: A function without a body should be abstract

A function inside a class (either a method or property) does not have a body and is not marked with the **abstract** modifier. Functions marked with the **abstract** modifier must not have a body, while functions with a body must not be marked with the **abstract** modifier.

To correct this error

- Mark the function with the **abstract** modifier.
- Add a body to the function.

See Also

Troubleshooting Your Scripts | JScript Reference | JScript Modifiers

JS1224: This modifier cannot be used on an interface member

An interface member has a modifier that is not allowed. Only the **public** modifier is allowed for interface members.

To correct this error

- Make sure that only the modifier that is applied to interface members is the **public** modifier.

See Also

Troubleshooting Your Scripts | JScript Reference | JScript Modifiers | interface Statement

JS1226: Variables cannot be declared in an interface

A variable is declared in an interface, which is not allowed. Variables may be declared in classes to create fields.

To correct this error

- Make sure variable declarations do not appear in interface declarations.

See Also

Troubleshooting Your Scripts | JScript Reference | interface Statement

JS1227: Interfaces cannot be declared in an interface

An interface declaration is nested within an interface declaration, which is not allowed. Interfaces declarations are only allowed within the global scope or within a package.

To correct this error

- Make sure nested interface declarations do not occur.

See Also

Troubleshooting Your Scripts | JScript Reference | interface Statement

JS1228: Enum member declarations should not use the 'var' keyword

An enumeration declaration contains the **var** keyword, but variable declarations are not allowed within an enumeration declaration.

To correct this error

- Remove the **var** keyword or the variable declaration.

See Also

Troubleshooting Your Scripts | JScript Reference | enum Statement | var Statement

JS1229: The import statement is not valid in this context

An **import** statement that appears in the code is not in the global scope. The import statement can only appear in the global scope.

To correct this error

- Move the **import** statement from the current location to the main program block (the global scope.)

See Also

Troubleshooting Your Scripts | JScript Reference | import Statement

JS1230: Enum declaration not allowed in this context

An enumeration declaration exists in an inappropriate context. Enumeration declarations are allowed only in the main program block, inside classes, inside packages, or in functions.

To correct this error

- Define the enumeration in the main program block, inside a class, inside a package, or in a function.

See Also

Troubleshooting Your Scripts | JScript Reference | enum Statement

JS1231: Attribute not valid for this type of declaration

The code applies an attribute to a declaration that cannot accept an attribute.

To correct this error

- Make sure that the attribute is not applied to this type of declaration.

See Also

Troubleshooting Your Scripts | JScript Reference

JS1232: Package declaration not allowed in this context

A package declaration appears in a context other than the global scope. This is not allowed; packages can only be declared in the global scope.

To correct this error

- Make sure all packages are declared in the global scope.

See Also

Troubleshooting Your Scripts | JScript Reference | package Statement

JS1233: A constructor function may not have a return type

The constructor function of a class has a return type specified. However, a constructor function automatically returns a reference to the constructed class instance; it does not return a value.

To correct this error

- Remove the return type specification from the constructor.
- Change the constructor to a method by renaming the method with a name that is different from the class name.

See Also

Troubleshooting Your Scripts | JScript Reference | Class-based Objects

JS1234: Only type and package definitions are allowed inside a library

The code is being compiled to create a library, but it contains declarations that are not allowed. Code that is used to create libraries can contain only classes, interfaces, and packages.

To correct this error

- Make sure that the code contains only classes, interfaces, and packages.

See Also

Troubleshooting Your Scripts | JScript Reference | /target:library | class Statement | interface Statement | package Statement

JS1235: Invalid debug directive

The **@debug** directive is used with an invalid option. The valid options are `on` and `off`.

To correct this error

- Use only `on` and `off` as the options for the **@debug** directive.

See Also

Troubleshooting Your Scripts | JScript Reference | @debug Directive

JS1236: This type of attribute must be unique

The code applies an attribute more than one time to an identifier, but the attribute can only be applied once.

To correct this error

- Make sure that the attribute is only applied one time for each identifier.

See Also

Troubleshooting Your Scripts | JScript Reference

JS1237: A non-static nested type can only be extended by non-static type nested in the same class

A class contains a nested class definition, and the nested class is not marked with the **static** modifier. Another defined class extends the nested class, but the extending class either does not have the correct modifiers or is not defined within the same class as the nested class. Only another non-static class that is nested within the same class can extend a non-static nested class.

To correct this error

- Make sure that only non-static nested classes extend non-static classes that are nested within the same class.
- Apply the **static** modifier to the nested class that is to be extended. This allows non-nested classes and static nested classes to extend the nested class.

See Also

Troubleshooting Your Scripts | JScript Reference | Class-based Objects | static Modifier

JS1238: An attribute that targets the property must be specified on the property getter, if present

A property that is defined with a setter (a **function set** declaration) and getter (a **function get** declaration) has an attribute applied to the setter, which is not allowed. All attributes must be applied explicitly to the getter, if a getter is present. The compiler applies the attributes implicitly to the setter.

To correct this error

- Apply the attribute the to property getter.

See Also

Troubleshooting Your Scripts | JScript Reference | function set Statement | function get Statement

JS1239: A throw must have an argument when not inside the catch block of a try statement

A **throw** statement is used without an argument outside a **catch** block. The only place where a **throw** statement can be used without an argument is within a **catch** block, in which case it rethrows the error that was caught.

To correct this error

- Pass an argument to the **throw** statement.
- Move the **throw** statement to a catch block.

See Also

Troubleshooting Your Scripts | JScript Reference | try...catch...finally Statement | throw Statement

JS1240: A variable argument list must be the last argument

The definition of the function has a parameter array (or variable argument list) that is followed by another parameter. This is not allowed because the parameter array must be the last parameter.

To correct this error

- Make sure the parameter array is the last parameter in the definition of a function.

See Also

Troubleshooting Your Scripts | JScript Reference | function Statement

JS1241: Type could not be found, is an assembly reference missing?

A qualified reference to a type uses a qualifier that looks like a package name. The type cannot be found in the package, or the package cannot be found. This can occur when the assembly that provides the type for the package is not referenced.

To correct this error

- Make sure the type exists in the package provided.
- Make sure that the **/autoref** option is turned on or that assemblies are referenced explicitly using the **/reference** option.

See Also

Troubleshooting Your Scripts | JScript Reference | import Statement | /reference | /autoref

JS1242: Malformed octal literal treated as decimal literal

A literal number begins with a leading zero and has no decimal point, which indicates that it is an octal (base 8) literal. However, it also contains the digits 8 or 9, which are not octal. The compiler will interpret the number as a decimal (base 10) number.

To correct this error

- If the literal should be a decimal literal, remove the leading zero.
- If the literal should be an octal literal, make sure it uses only the digits zero through seven.

See Also

Troubleshooting Your Scripts | JScript Reference | Numeric Data

JS1243: A non-static member is not accessible from a static scope

A static method or property accessed a non-static member of the class. Static class members are associated with the class itself and do have information about the members of a particular instance, while non-static members are associated with a particular instance. This means that static methods and properties cannot access non-static members.

Non-static members can be accessed indirectly by a static method when an instance of the class is passed to the method as an argument. The static method can access all the members of the class instance, including members marked with the **private** modifier.

To correct this error

- Change the modifiers so that both the accessed member and the member that accesses are either static or non-static.
- Pass an instance of the class to the static method.

See Also

Troubleshooting Your Scripts | JScript Reference | static Modifier | Class-based Objects

JS1244: A static member must be accessed with the class name

The code is attempting to access a static class member through a class instance. Static class members are associated with the class itself and cannot be accessed from a class instance; they must be accessed directly with the class name as the qualifier.

To correct this error

- Make sure that static members are accessed with the class name.

See Also

Troubleshooting Your Scripts | JScript Reference | static Modifier | Class-based Objects

JS1245: A non-static member cannot be accessed with the class name

The code is attempting to access a non-static class member by using the class name. Only static class members are associated with the class itself; non-static members are associated with and accessed through a particular class instance.

To correct this error

- Make sure that non-static members are accessed with a class instance.

See Also

Troubleshooting Your Scripts | JScript Reference | static Modifier | Class-based Objects

JS1246: Type does not have such a static member

The code is attempting to access a member by using the class name (which accesses only static class members) but no matching static member is found.

To correct this error

- When accessing a member by using the class name, make sure the member is static.

See Also

Troubleshooting Your Scripts | JScript Reference | static Modifier | Class-based Objects

JS1247: The loop condition is a function reference

The loop condition is a function reference. Did you intend to call the function?

In the conditional expression part of a loop statement, a function name is not followed by a set of parentheses that enclose the function arguments. The function name by itself refers to the **Function** object for the function; it does not refer to value that the function returns.

Although using a **Function** object as the loop condition may be useful in certain circumstances, such as when the function itself is changing, this is most likely an error.

To correct this error

- Use the function-calling syntax with parentheses enclosing the function arguments to evaluate the function value.

See Also

Troubleshooting Your Scripts | JScript Reference

JS1248: Expected 'assembly'

The code appears to define global attributes for an assembly, but the assembly identifier is not used. The correct syntax to define an assembly attribute is:

```
[assembly: attribute]
```

The `attribute` should be a valid global attribute for an assembly, which are provided by the **System.Reflection** namespace. For more information, see System.Reflection Namespace.

To correct this error

- Make sure to use the correct syntax to declare global attributes.

See Also

Troubleshooting Your Scripts | JScript Reference

JS1249: Assembly custom attributes may not be part of another construct

Assembly custom attributes can only be used in the global scope.

To correct this error

- Make sure that assembly custom attributes are used in the global scope.

See Also

Troubleshooting Your Scripts | JScript Reference | Scope of Variables and Constants

JS1250: Expando methods cannot be static

A method has both **expando** and **static** modifiers. This is not allowed.

To correct this error

- Remove either the **expando** or **static** modifier from the method declaration.

See Also

Troubleshooting Your Scripts | JScript Reference | JScript Modifiers

JS1251: This method has the same name and parameter types as another method in this class

There are multiple methods in the class with the same name and parameter types. This is not allowed because there is no way to distinguish the different methods when calling them.

To correct this error

- If there are duplicate methods, remove the redundant ones.
- Change the name or the parameter types of one or more of the methods.

See Also

Troubleshooting Your Scripts | JScript Reference | Class-based Objects

JS1252: Class members used as constructors should be marked as expando functions

The **new** operator was applied to a class member. This is allowed only when the class member is a method or property marked with the **expando** modifier, which allows it to be used a constructor.

To correct this error

- Apply the **expando** modifier to the definition of the class member.
- Do not use the **new** operator on non-expando class members.

See Also

Troubleshooting Your Scripts | JScript Reference | expando Modifier | new Operator

JS1253: Not a valid version string

Not a valid version string. Expected format is major.minor[.build[.revision]]

The code defines a global **AssemblyVersion** attribute for the assembly, but the version string passed to the attribute does not have the correct form. The version string must have the format "major.minor[.build[.revision]]".

To correct this error

- Make sure the version string has the format "major.minor[.build[.revision]]".

See Also

Troubleshooting Your Scripts | JScript Reference | AssemblyVersionAttribute Class

JS1254: Executables cannot be localized, Culture should always be empty

The code defines a global **AssemblyCulture** attribute for the assembly, which is not allowed since executables cannot be localized.

To correct this error

- Make sure the **AssemblyCulture** attribute is not specified for an executable file.

See Also

Troubleshooting Your Scripts | JScript Reference | AssemblyCultureAttribute Class

JS1255: The plus operator is a slow way to concatenate strings

The plus operator is a slow way to concatenate strings. Consider using System.Text.StringBuilder instead.

The plus (**+**) operator concatenates strings. In many circumstances, such as appending many small strings to another string, **System.Text.StringBuilder** produces code that runs much faster.

For example, consider the following code that builds the string "0123456789". It generates this warning when compiled.

```
var a : String = "";
for(var i=0; i<10; i++)
   a += i;
print(a);
```

When run, this displays the string "0123456789".

When the previous example uses **System.Text.StringBuilder**, the program runs faster and does not generate the warning.

```
var b : System.Text.StringBuilder;
b = new System.Text.StringBuilder();
for(var i=0; i<10; i++)
   b.Append(i);
print(b);
```

Like the program before, this also displays "0123456789".

Another way to prevent the warning from being displayed is to use an untyped variable to hold the string to which other strings are appended.

To correct this error

- Use **System.Text.StringBuilder** for the type of the string to which other strings are appended, and rewrite the statements with the **+=** operations to use the **Append** method instead.
- Use an untyped variable for the string to which other strings are appended. (This solution will not make the code run faster, but it will suppress the warning.)

See Also

Troubleshooting Your Scripts | JScript Reference | StringBuilder Class

JS1256: Conditional compilation directives and variables cannot be used in the debugger

While debugging a JScript .NET program, a conditional compilation directive or variable was entered into the Command Window. Conditional compilation directives and variables are only used when the program is being compiled; they are not available after the compilation is completed.

To correct this error

- Make sure that conditional compilation directives and variables are not entered in the Command Window.

See Also

Troubleshooting Your Scripts | JScript Reference | Writing, Compiling, and Debugging JScript Code

JS1257: Expando methods must be public

A method that has an **expando** modifier also has a visibility modifier that is not **public**. This is not allowed.

To correct this error

- Remove either the **expando** or visibility modifier from the method declaration.

See Also

Troubleshooting Your Scripts | JScript Reference | JScript Modifiers

JS1258: Delegates should not be explicitly constructed, simply use the method name

The code is constructing a delegate from a function, which is unnecessary. The function name by itself refers to a delegate.

To correct this error

- Use the function name without parentheses to refer to a delegate.

See Also

Troubleshooting Your Scripts | JScript Reference | JScript Functions

JS1259: A referenced assembly depends on another assembly that is not referenced or could not be found

The program imports an assembly (either implicitly with the **import** statement or explicitly with the **/reference** option) that depends on another assembly. The other assembly cannot be found because it is either not referenced or it does not exist in the specified location.

One possible cause of this error is moving one assembly to a new location without moving the assemblies on which it depends. Another cause is not referencing assemblies on which other assemblies depend.

To correct this error

- Make sure that the assemblies required by the program exist.
- Check that the location and name is correctly specified for each required assembly.
- Make sure to explicitly reference assemblies that are required by other assemblies but are not imported by the program.

See Also

Troubleshooting Your Scripts | JScript Reference | import Statement | /reference

JS1260: This conversion may fail at runtime

The code has an implicit type conversion that may fail at runtime. This indicates that some values of data in the original data type do not have an obvious analogue in the target conversion type.

Using explicit type conversions, which allow for lossy conversions, makes code more reliable and much less likely to fail at runtime.

To correct this error

- Make sure that the data provided is compatible with the data type to which it is converted.
- Use explicit type conversion when converting data from one type to another.

See Also

Troubleshooting Your Scripts | JScript Reference | Type Conversion

JS1261: Converting a string to a number or boolean is slow and may fail at runtime

The code has an implicit type conversion that may fail at runtime. This means that some string values do not have an obvious analogue as a number or Boolean.

Using explicit type conversions, which allow for lossy conversions, makes code more reliable and much less likely to fail at runtime.

To correct this error

- Make sure that the string provided is compatible with the data type to which it is converted.
- Use explicit type conversion when converting data from one type to another.

See Also

Troubleshooting Your Scripts | JScript Reference | Type Conversion

JS1262: Not a valid .resources file

The program is compiled with the **/resource** option, but the resource file specified does not have the correct format. The file may be corrupt, or it might not be a resource file.

To correct this error

- Make sure that the file specified with the **/resource** option is a valid resource file.

See Also

Troubleshooting Your Scripts | JScript Reference | /resource

JS1263: The & operator can only be used in a list of arguments

An ampersand (**&**) in the code is used outside a function call. The ampersand should precede only a variable name (which has an address), and it should only be used to pass the variable by reference to a function that accepts the parameter by reference.

Passing variables by reference allows the function to change the value of the variable.

> **Note** JScript .NET does not allow functions to be defined with reference parameters. JScript .NET provides the ampersand to allow calls to external objects that take reference parameters.

To correct this error

- Make sure that the ampersand (**&**) precedes a variable name in a call to a function and that the function accepts the parameter by reference.

See Also

Troubleshooting Your Scripts | JScript Reference

JS1264: The specified type is not CLS compliant

The program contains the **CLSCompliantAttribute** attribute, and the compiler has detected a data type definition that is not Common Language Specification (CLS) compliant. Some possible causes of this error are:

- The type name is not CLS compliant. CLS compliant names cannot start with an underscore (_), contain a dollar sign ($), or differ only in capitalization from the name of another public member.
- An enumeration is defined to have an underlying type that is not a CLS compliant type. For example, an enumeration may be based on the primitive type **uint** (which is not CLS compliant) instead of the corresponding CLS compliant system type **System.UInt32**.

To correct this error

- Make sure that the data type name does not start with an underscore (_), contain a dollar sign ($), or differ only in capitalization from the name of another member.
- Make sure that only CLS compliant data types are used as the underlying types for enumerations.

See Also

Troubleshooting Your Scripts | JScript Reference | CLSCompliantAttribute Class | Writing CLS-Compliant Code |
What is the Common Language Specification?

JS1265: Class member cannot be marked CLS compliant because the class is not marked as CLS compliant

The class contains a member marked with the **CLSCompliantAttribute** attribute, but the class itself is not marked with the **CLSCompliantAttribute** attribute. A class must be marked as CLS compliant if any class member is marked as CLS compliant.

To correct this error

- Make sure to apply the **CLSCompliantAttribute** attribute to each class that has members marked with the **CLSCompliantAttribute** attribute.

See Also

Troubleshooting Your Scripts | JScript Reference | CLSCompliantAttribute Class | Writing CLS-Compliant Code | What is the Common Language Specification?

JS1266: Type cannot be marked CLS compliant because the assembly is not marked as CLS compliant

A data type is marked with the **CLSCompliantAttribute** attribute, but the assembly that contains the data type is not marked with the **CLSCompliantAttribute** attribute. The assembly must be marked as CLS compliant if it contains any data types marked as CLS compliant.

To correct this error

- Make sure to apply the **CLSCompliantAttribute** attribute to the assembly if any data types are marked with the **CLSCompliantAttribute** attribute.

See Also

Troubleshooting Your Scripts | JScript Reference | CLSCompliantAttribute Class | Writing CLS-Compliant Code | What is the Common Language Specification?

Functions

These functions are built into JScript return values and other functions can use them in subsequent operations.

In This Section

GetObject Function
Returns a reference to an Automation object from a file.

ScriptEngine Function
Returns a string representing the scripting language in use.

ScriptEngineBuildVersion Function
Returns the build version number of the scripting engine in use.

ScriptEngineMajorVersion Function
Returns the major version number of the scripting engine in use.

ScriptEngineMinorVersion Function
Returns the minor version number of the scripting engine in use.

Related Sections

JScript Reference
Lists elements that comprise JScript Language Reference and links to topics that explain the details behind the proper use of language elements.

.NET Framework Reference
Lists links to topics that explain the syntax and structure of the .NET Framework class library and other essential elements.

GetObject Function

Returns a reference to an Automation object from a file. There are two forms of the function.

```
function GetObject(class : String)
function GetObject(pathname : String [, class : String])
```

Arguments

class
Required. A string of the form "*appName.objectType*", where *appName* is the name of the application providing the object, and *objectType* is the type or class of object to create.

pathname
Required. Full path and name of the file containing the object to retrieve. If *pathname* is omitted, *class* is required.

Remarks

Use the **GetObject** function to access an Automation object from a file. Assign the object returned by **GetObject** to the object variable. For example:

```
var CADObject;
CADObject = GetObject("C:\\CAD\\SCHEMA.CAD");
```

When this code is executed, the application associated with the specified *pathname* is started, and the object in the specified file is activated. If *pathname* is a zero-length string (""), **GetObject** returns a new object instance of the specified type. If the *pathname* argument is omitted, **GetObject** returns a currently active object of the specified type. If no object of the specified type exists, an error occurs.

Some applications allow you to activate part of a file. To do so, add an exclamation point (!) to the end of the file name and follow it with a string that identifies the part of the file you want to activate. For information on how to create this string, see the documentation for the application that created the object.

For example, in a drawing application you might have multiple layers to a drawing stored in a file. You could use the following code to activate a layer within a drawing called `SCHEMA.CAD`:

```
var LayerObject = GetObject("C:\\CAD\\SCHEMA.CAD!Layer3");
```

If you don not specify the object's class, Automation determines which application to start and which object to activate, based on the file name you provide. Some files, however, may support more than one class of object. For example, a drawing might support three different types of objects: an Application object, a Drawing object, and a Toolbar object, all of which are part of the same file. To specify which object in a file you want to activate, use the optional *class* argument. For example:

```
var MyObject;
MyObject = GetObject("C:\\DRAWINGS\\SAMPLE.DRW", "FIGMENT.DRAWING");
```

In the preceding example, `FIGMENT` is the name of a drawing application and `DRAWING` is one of the object types it supports. Once an object is activated, you reference it in code using the object variable you defined. In the preceding example, you access properties and methods of the new object using the object variable `MyObject`. For example:

```
MyObject.Line(9, 90);
MyObject.InsertText(9, 100, "Hello, world.");
MyObject.SaveAs("C:\\DRAWINGS\\SAMPLE.DRW");
```

> **Note** Use the **GetObject** function when there is a current instance of the object, or if you want to create the object with a file already loaded. If there is no current instance, and you don't want the object started with a file loaded, use the **ActiveXObject** object.

If an object has registered itself as a single-instance object, only one instance of the object is created, no matter how many times **ActiveXObject** is executed. With a single-instance object, **GetObject** always returns the same instance when called with the zero-length string ("") syntax, and it causes an error if the *pathname* argument is omitted.

Requirements

Version 5

See Also

ActiveXObject Object

ScriptEngine Function

Returns a string representing the scripting language in use.

```
function ScriptEngine() : String
```

Remarks

The **ScriptEngine** function can return any of the following strings:

String	Description
JScript	Indicates that Microsoft JScript is the current scripting engine.
VBA	Indicates that Microsoft Visual Basic for Applications is the current scripting engine.
VBScript	Indicates that Microsoft Visual Basic Scripting Edition is the current scripting engine.

Example

The following example illustrates the use of the **ScriptEngine** function:

```
function GetScriptEngineInfo(){
   var s;
   s = ""; // Build string with necessary info.
   s += ScriptEngine() + " Version ";
   s += ScriptEngineMajorVersion() + ".";
   s += ScriptEngineMinorVersion() + ".";
   s += ScriptEngineBuildVersion();
   return(s);
}
```

Requirements

Version 2

See Also

ScriptEngineBuildVersion Function | ScriptEngineMajorVersion Function | ScriptEngineMinorVersion Function

ScriptEngineBuildVersion Function

Returns the build version number of the scripting engine in use.

```
function ScriptEngineBuildVersion() : int
```

Remarks

The return value corresponds directly to the version information contained in the dynamic-link library (DLL) for the scripting language in use.

Example

The following example illustrates the use of the **ScriptEngineBuildVersion** function:

```
function GetScriptEngineInfo(){
   var s;
   s = ""; // Build string with necessary info.
   s += ScriptEngine() + " Version ";
   s += ScriptEngineMajorVersion() + ".";
   s += ScriptEngineMinorVersion() + ".";
   s += ScriptEngineBuildVersion();
   return(s);
}
```

Requirements

Version 2

See Also

ScriptEngine Function | ScriptEngineMajorVersion Function | ScriptEngineMinorVersion Function

ScriptEngineMajorVersion Function

Returns the major version number of the scripting engine in use.

```
function ScriptEngineMajorVersion() : int
```

Remarks

The return value corresponds directly to the version information contained in the dynamic-link library (DLL) for the scripting language in use.

Example

The following example illustrates the use of the **ScriptEngineMajorVersion** function:

```
function GetScriptEngineInfo(){
   var s;
   s = ""; // Build string with necessary info.
   s += ScriptEngine() + " Version ";
   s += ScriptEngineMajorVersion() + ".";
   s += ScriptEngineMinorVersion() + ".";
   s += ScriptEngineBuildVersion();
   return(s);
}
```

Requirements

Version 2

See Also

ScriptEngine Function | ScriptEngineBuildVersion Function | ScriptEngineMinorVersion Function

ScriptEngineMinorVersion Function

Returns the minor version number of the scripting engine in use.

```
function ScriptEngineMinorVersion() : int
```

Remarks

The return value corresponds directly to the version information contained in the dynamic-link library (DLL) for the scripting language in use.

Example

The following example illustrates the use of the **ScriptEngineMinorVersion** function.

```
function GetScriptEngineInfo(){
   var s;
   s = ""; // Build string with necessary info.
   s += ScriptEngine() + " Version ";
   s += ScriptEngineMajorVersion() + ".";
   s += ScriptEngineMinorVersion() + ".";
   s += ScriptEngineBuildVersion();
   return(s);
}
```

Requirements

Version 2

See Also

ScriptEngine Function | ScriptEngineBuildVersion Function | ScriptEngineMajorVersion Function

Literals

Literals are invariant program elements that have special meaning within the context of JScript code.

In This Section

false Literal

null Literal

true Literal

Related Sections

JScript Reference
: Lists elements that comprise JScript Language Reference and links to topics that explain the details behind the proper use of language elements.

.NET Framework Reference
: Lists links to topics that explain the syntax and structure of the .NET Framework class library and other essential elements.

false Literal

A Boolean value that represents false.

Remarks

A Boolean value is either **true** or **false**. The opposite of **false**, or not **false**, is **true**.

Requirements

Version 1

See Also

boolean Data Type | Boolean Object | true Literal | JScript Data Types

null Literal

An object that represents "no object".

Remarks

You can erase the contents of a variable (without deleting the variable) by assigning it the **null** value.

> **Note** In JScript, **null** is not the same as 0 (as it is in C and C++). Also the **typeof** operator in JScript reports null values as being of type **Object**, not of type **null**. This potentially confusing behavior is maintained for backwards compatibility.

Requirements

Version 1

See Also

Object Object | JScript Data Types

true Literal

A Boolean value that represents true.

Remarks

A Boolean value is either **true** or **false**. The opposite of **true**, or not **true**, is **false**.

Requirements

Version 1

See Also

boolean Data Type | Boolean Object | false Literal | JScript Data Types

Methods

A method is a function that is a member of an object. The many methods in JScript are categorized alphabetically according to method name.

In This Section

Methods (A-E)

Methods (F-I)

Methods (J-R)

Methods (S)

Methods (T-Z)

Related Sections

JScript Reference
Lists elements that comprise JScript Language Reference and links to topics that explain the details behind the proper use of language elements.

Objects
Explains the concept of objects in JScript, explains how objects are related to properties and methods, and links to topics that provide more detail about the objects that JScript supports.

.NET Framework Reference
Lists links to topics that explain the syntax and structure of the .NET Framework class library and other essential elements.

Methods (A-E)

A method is a function that is a member of an object. Following are methods whose names begin with letters a through e.

In This Section

abs Method
  Returns the absolute value of a number.
acos Method
  Returns the arccosine of a number.
anchor Method
  Places an HTML anchor with a NAME attribute around specified text in the object.
apply Method
  Returns a method of an object, substituting another object for the current object.
asin Method
  Returns the arcsine of a number.
atan Method
  Returns the arctangent of a number.
atan2 Method
  Returns the angle (in radians) from the X-axis to a point (x,y).
atEnd Method
  Returns a Boolean value indicating if the enumerator is at the end of the collection.
big Method
  Places HTML <BIG> tags around text in a **String** object.
blink Method
  Places HTML <BLINK> tags around text in a **String** object.
bold Method
  Places HTML <B> tags around text in a **String** object.
call Method
  Calls a method of an object, substituting another object for the current object.
ceil Method
  Returns the smallest integer greater than or equal to its numeric argument.
charAt Method
  Returns the character at the specified index.
charCodeAt Method
  Returns the Unicode encoding of the specified character.
compile Method
  Compiles a regular expression into an internal format.
concat Method (Array)
  Returns a new array consisting of a combination of two arrays.
concat Method (String)
  Returns a **String** object containing the concatenation of two supplied strings.
cos Method
  Returns the cosine of a number.
decodeURI Method
  Returns the unencoded version of an encoded Uniform Resource Identifier (URI).
decodeURIComponent Method
  Returns the unencoded version of an encoded component of a Uniform Resource Identifier (URI).
dimensions Method
  Returns the number of dimensions in a VBArray.
encodeURI Method
  Encodes a text string as a valid Uniform Resource Identifier (URI).
encodeURIComponent Method
  Encodes a text string as a valid component of a Uniform Resource Identifier (URI).
escape Method
  Encodes **String** objects so they can be read on all computers.
eval Method
  Evaluates JScript code and executes it.
exec Method
  Executes a search for a match in a specified string.
exp Method

Returns *e* (the base of natural logarithms) raised to a power.

Related Sections

JScript Reference
Lists elements that comprise JScript Language Reference and links to topics that explain the details behind the proper use of language elements.

Methods
Lists the methods, classified alphabetically, available in JScript, and links to each category of methods.

Objects
Explains the concept of objects in JScript, how objects are related to properties and methods, and links to topics that provide more detail about the objects that JScript supports.

abs Method

Returns the absolute value of a number.

```
function abs( number : Number ) : Number
```

Arguments

number
Required. A numeric expression.

Remarks

The return value is the absolute value of the *number* argument.

Example

The following example illustrates the use of the **abs** method.

```
function ComparePosNegVal(n) {
   var s = "";
   var v1 = Math.abs(n);
   var v2 = Math.abs(-n);
   if (v1 == v2) {
      s = "The absolute values of " + n + " and "
      s += -n + " are identical.";
   }
   return(s);
}
```

Requirements

Version 1

See Also

Methods

Applies To: Math Object

acos Method

Returns the arccosine of a number.

```
function acos( number : Number ) : Number
```

Arguments

number
Required. A numeric expression.

Remarks

The return value is between 0 and *pi*, representing the arccosine of the *number* argument. If *number* is less than -1 or *number* is greater than +1, the return value is **NaN**.

Requirements

Version 1

See Also

asin Method | atan Method | cos Method | sin Method | tan Method | PI Property

Applies To: Math Object

anchor Method

Returns a string with an HTML anchor with a NAME attribute around specified text in the object.

```
function anchor(anchorString : String ) : String
```

Arguments

anchorString
Required. Text you want to place in the NAME attribute of an HTML anchor.

Remarks

Call the **anchor** method to create a named anchor out of a **String** object.

No checking is done to see if the tag has already been applied to the string.

Example

The following example demonstrates how the **anchor** method accomplishes this:

```
var strVariable = "This is an anchor";
strVariable = strVariable.anchor("Anchor1");
```

The value of `strVariable` after the last statement is:

```
<A NAME="Anchor1">This is an anchor</A>
```

Requirements

Version 1

See Also

link Method

Applies To: String Object

apply Method

Returns a method of an object, substituting another object for the current object.

```
function apply([thisObj : Object [,argArray : { Array | arguments }]])
```

Arguments

thisObj
Optional. The object to be used as the current object.
argArray
Optional. Array of arguments or an **arguments** object to be passed to the function.

Remarks

If *argArray* is not a valid array or is not the **arguments** object, then a **TypeError** results.

If neither *argArray* nor *thisObj* are supplied, the **global** object is used as *thisObj* and is passed no arguments.

Requirements

Version 5.5

See Also

Methods

Applies To: Function Object

asin Method

Returns the arcsine of a number.

```
function asin(number : Number) : Number
```

Arguments

number
Required. A numeric expression.

Remarks

The return value is between *-pi/2* and *pi/2*, representing the arcsine of the *number* argument. If *number* is less than -1 or *number* is greater than +1, the return value is **NaN**.

Requirements

Version 1

See Also

acos Method | atan Method | cos Method | sin Method | tan Method | PI Property

Applies To: Math Object

atan Method

Returns the arctangent of a number.

```
function atan(number : Number ) Number
```

Arguments

number
Required. A numeric expression.

Remarks

The return value is between *-pi/2* and *pi/2*, representing the arctangent of the *number* argument.

Requirements

Version 1

See Also

acos Method | asin Method | atan2 Method | cos Method | sin Method | tan Method | PI Property

Applies To: Math Object

atan2 Method

Returns the angle (in radians) from the X axis to a point (*x*,*y*).

```
function atan2(y : Number , x : Number ) : Number
```

Arguments

x
Required. A numeric expression representing the Cartesian x-coordinate of a point.
y
Required. A numeric expression representing the Cartesian y-coordinate of a point.

Remarks

The return value is between *-pi* and *pi*, representing the angle of the supplied (*x*,*y*) point.

Requirements

Version 1

See Also

atan Method | tan Method | PI Property

Applies To: Math Object

atEnd Method

Returns a Boolean value indicating if the enumerator is at the end of the collection.

```
function atEnd() : Boolean
```

Remarks

The **atEnd** method returns **true** if the current item is the last one in the collection, the collection is empty, or the current item is undefined. Otherwise, it returns **false**.

Example

In following code, the **atEnd** method is used to determine if the end of a list of drives has been reached:

```
function ShowDriveList(){
   var fso, s, n, e, x;
   fso = new ActiveXObject("Scripting.FileSystemObject");
   e = new Enumerator(fso.Drives);
   s = "";
   for (; !e.atEnd(); e.moveNext())
   {
      x = e.item();
      s = s + x.DriveLetter;
      s += " - ";
      if (x.DriveType == 3)
         n = x.ShareName;
      else if (x.IsReady)
         n = x.VolumeName;
      else
         n = "[Drive not ready]";
      s +=   n + "<br>";
   }
   return(s);
}
```

Requirements

Version 2

See Also

item Method | moveFirst Method | moveNext Method

Applies To: Enumerator Object

big Method

Returns a string with HTML <BIG> tags around the text in a **String** object.

```
function big() : String
```

Remarks

No checking is done to see if the tag has already been applied to the string.

Example

The example that follows shows how the **big** method works:

```
var strVariable = "This is a string object";
strVariable = strVariable.big();
```

The value of `strVariable` after the last statement is:

```
<BIG>This is a string object</BIG>
```

Requirements

Version 1

See Also

small Method

Applies To: String Object

blink Method

Returns a string with HTML <BLINK> tags around the text in a **String** object.

```
function blink() : String
```

Remarks

No checking is done to see if the tag has already been applied to the string.

The <BLINK> tag is not supported in Microsoft Internet Explorer.

Example

The following example demonstrates how the **blink** method works:

```
var strVariable = "This is a string object";
strVariable = strVariable.blink();
```

The value of `strVariable` after the last statement is:

```
<BLINK>This is a string object</BLINK>
```

Requirements

Version 1

See Also

Methods

Applies To: String Object

bold Method

Returns a string with HTML <B> tags around the text in a **String** object.

```
function bold() : String
```

Remarks

No checking is done to see if the tag has already been applied to the string.

Example

The following example demonstrates how the **bold** method works:

```
var strVariable = "This is a string object";
strVariable = strVariable.bold();
```

The value of `strVariable` after the last statement is:

```
<B>This is a string object</B>
```

Requirements

Version 1

See Also

italics Method

Applies To: String Object

call Method

Calls a method of an object, substituting another object for the current object.

```
function call([thisObj : Object [, arg1[, arg2[, ... [, argN]]]]])
```

Arguments

thisObj
Optional. The object to be used as the current object.
arg1, *arg2*, ..., *argN*
Optional. List of arguments to be passed to the method.

Remarks

The **call** method is used to call a method on behalf of another object. The **call** method allows you to change the object context of a function from the original context to the new object specified by *thisObj*.

If *thisObj* is not supplied, the **global** object is used as *thisObj*.

Requirements

Version 5.5

See Also

Methods

Applies To: Function Object

ceil Method

Returns the smallest integer greater than or equal to its numeric argument.

```
function ceil(number : Number ) : Number
```

Arguments

number
Required. A numeric expression.

Remarks

The return value is an integer value equal to the smallest integer greater than or equal to its numeric argument.

Requirements

Version 1

See Also

floor Method

Applies To: Math Object

charAt Method

Returns the character at the specified index of a **String** object.

```
function charAt(index : Number) : String
```

Arguments

index

Required. Zero-based index of the desired character. Valid values are between 0 and the length of the string minus 1.

Remarks

The **charAt** method returns a character value equal to the character at the specified *index*. The first character in a string is at index 0, the second is at index 1, and so forth. Values of *index* out of valid range return an empty string.

Example

The following example illustrates the use of the **charAt** method:

```
function charAtTest(n){
   var str = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"; // Initialize variable.
   var s;                              // Declare variable.
   s = str.charAt(n - 1);              // Get correct character
                                       // from position n – 1.
   return(s);                          // Return character.
}
```

Requirements

Version 1

See Also

Methods

Applies To: String Object

charCodeAt Method

Returns an integer representing the Unicode encoding of the character at the specified location in a **String** object.

```
function charCodeAt(index : Number) : String
```

Arguments

index

Required. Zero-based index of the desired character. Valid values are between 0 and the length of the string minus 1.

Remarks

The first character in a string is at index 0, the second is at index 1, and so forth.

If there is no character at the specified *index*, **NaN** is returned.

Example

The following example illustrates the use of the **charCodeAt** method.

```
function charCodeAtTest(n){
  var str = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"; //Initialize variable.
  var n;                              //Declare variable.
  n = str.charCodeAt(n - 1);          //Get the Unicode value of the
                                      // character at position n.
  return(n);                          //Return the value.
}
```

Requirements

Version 5.5

See Also

fromCharCode Method

Applies To: String Object

compile Method

Compiles a regular expression into an internal format for faster execution.

```
function compile(pattern : String [, flags : String] )
```

Arguments

pattern
Required. A string expression containing a regular expression pattern to be compiled
flags
Optional. Available flags, which may be combined, are:

- g (global search for all occurrences of *pattern*)
- i (ignore case)
- m (multiline search)

Remarks

The **compile** method converts *pattern* into an internal format for faster execution. This allows for more efficient use of regular expressions in loops, for example. A compiled regular expression speeds things up when reusing the same expression repeatedly. No advantage is gained, however, if the regular expression changes.

Example

The following example illustrates the use of the **compile** method:

```
function CompileDemo(){
   var rs;
   var s = "AaBbCcDdEeFfGgHhIiJjKkLlMmNnOoPp"
   // Create regular expression for uppercase only.
   var r = new RegExp("[A-Z]", "g");
   var a1 = s.match(r)              // Find matches.
   // Compile the regular expression for lowercase only.
   r.compile("[a-z]", "g");
   var a2 = s.match(r)              // Find matches.
   return(a1 + "\n" + a2);
}
```

Requirements

Version 3

See Also

Regular Expression Syntax

Applies To: Regular Expression Object

concat Method (Array)

Returns a new array consisting of a combination of the current array and any additional items.

```
function concat([item1 : { Object | Array } [, ... [, itemN : { Object | Array }]]]]) : Array
```

Arguments

item1, item2, ..., itemN
Optional. Additional items to add to the end of the current array.

Remarks

The **concat** method returns an **Array** object containing the concatenation of the current array and any other supplied items.

The items to be added (*item1 ... itemN*) to the array are added, in order, from left to right. If one of the items is an array, its contents are added to the end of the current array. If the item is anything other than an array, it is added to the end of the array as a single array element.

Elements of source arrays are copied to the resulting array as follows:

- For an object reference copied from any of the arrays being concatenated to the new array, the object reference continues to point to the same object. A change in either the new array or the original array will result in a change to the other.
- For a numeric or string value being concatenated to the new array, only the value is copied. Changes in a value in one array do not affect the value in the other.

Example

The following example illustrates the use of the **concat** method when used with an array:

```
function ConcatArrayDemo(){
   var a, b, c, d;
   a = new Array(1,2,3);
   b = "JScript";
   c = new Array(42, "VBScript");
   d = a.concat(b, c);
   //Returns the array [1, 2, 3, "JScript", 42, "VBScript"]
   return(d);
}
```

Requirements

Version 3

See Also

concat Method (String) | join Method | String Object

Applies To: Array Object

concat Method (String)

Returns a string value containing the concatenation of the current string with any supplied strings.

```
function concat([string1 : String [, ... [, stringN : String]]]]) : String
```

Arguments

string1, ... , stringN
Optional. **String** objects or literals to concatenate to the end of the current string.

Remarks

The result of the **concat** method is equivalent to: *result* = *curstring* + *string1* + ... + *stringN*. The *curstring* refers the string stored in the object that supplies the **concat** method. A change of value in either a source or result string does not affect the value in the other string. If any of the arguments are not strings, they are first converted to strings before being concatenated to *curstring*.

Example

The following example illustrates the use of the **concat** method when used with a string:

```
function concatDemo(){
   var str1 = "ABCDEFGHIJKLM"
   var str2 = "NOPQRSTUVWXYZ";
   var s = str1.concat(str2);
   // Return concatenated string.
   return(s);
}
```

Requirements

Version 3

See Also

Addition Operator (+) | Array Object | concat Method (Array)

Applies To: String Object

cos Method

Returns the cosine of a number.

```
function cos(number : Number) : Number
```

Arguments

number
Required. A numeric expression.

Remarks

The return value is the cosine of its numeric argument.

Requirements

Version 1

See Also

acos Method | asin Method | atan Method | sin Method | tan Method

Applies To: Math Object

decodeURI Method

Returns the unencoded version of an encoded Uniform Resource Identifier (URI).

```
function decodeURI(URIstring : String) : String
```

Arguments

URIstring
Required. A string representing an encoded URI.

Remarks

Use the **decodeURI** method instead of the obsolete **unescape** method.

The **decodeURI** method returns a string value.

If the *URIString* is not valid, a **URIError** occurs.

Requirements

Version 5.5

See Also

decodeURIComponent Method | encodeURI Method

Applies To: Global Object

decodeURIComponent Method

Returns the unencoded version of an encoded component of a Uniform Resource Identifier (URI).

```
function decodeURIComponent(encodedURIString : String) : String
```

The required *encodedURIString* argument is a value representing an encoded URI component.

Remarks

A URIComponent is part of a complete URI.

If the *encodedURIString* is not valid, a **URIError** occurs.

Requirements

Version 5.5

See Also

decodeURI Method | encodeURI Method

Applies To: Global Object

dimensions Method

Returns the number of dimensions in a VBArray.

```
function dimensions() : Number
```

Remarks

The **dimensions** method provides a way to retrieve the number of dimensions in a specified VBArray.

The following example consists of three parts. The first part is VBScript code to create a Visual Basic safe array. The second part is JScript code that determines the number of dimensions in the safe array and the upper bound of each dimension. Both of these parts go into the <HEAD> section of an HTML page. The third part is the JScript code that goes in the <BODY> section to run the other two parts.

```
<HEAD>
<SCRIPT LANGUAGE="VBScript">
<!--
Function CreateVBArray()
   Dim i, j, k
   Dim a(2, 2)
   k = 1
   For i = 0 To 2
      For j = 0 To 2
         a(j, i) = k
         k = k + 1
      Next
   Next
   CreateVBArray = a
End Function
-->
</SCRIPT>

<SCRIPT LANGUAGE="JScript">
<!--
function VBArrayTest(vba)
{
   var i, s;
   var a = new VBArray(vba);
   for (i = 1; i <= a.dimensions(); i++)
   {
      s = "The upper bound of dimension ";
      s += i + " is ";
      s += a.ubound(i)+ ".<BR>";
   }
   return(s);
}
-->
</SCRIPT>
</HEAD>

<BODY>
<SCRIPT language="jscript">
   document.write(VBArrayTest(CreateVBArray()));
</SCRIPT>
</BODY>
```

Requirements

Version 3

See Also

getItem Method | lbound Method | toArray Method | ubound Method

Applies To: VBArray Object

encodeURI Method

Returns a string encoded as a valid Uniform Resource Identifier (URI)

```
function encodeURI(URIString : String) : String
```

Arguments

URIString
Required. A string representing an encoded URI.

Remarks

The **encodeURI** method returns an encoded URI. If you pass the result to **decodeURI**, the original string is returned. The **encodeURI** method does not encode the following characters: ":", "/", ";", and "?". Use **encodeURIComponent** to encode these characters.

Requirements

Version 5.5

See Also

decodeURI Method | decodeURIComponent Method

Applies To: Global Object

encodeURIComponent Method

Returns as string encoded as a valid component of a Uniform Resource Identifier (URI).

```
function encodeURIComponent(encodedURIString : String) : String
```

Arguments

encodedURIString
Required. A string representing an encoded URI component.

Remarks

The **encodeURIComponent** method returns an encoded URI. If you pass the result to **decodeURIComponent**, the original string is returned. Because the **encodeURIComponent** method encodes all characters, be careful if the string represents a path such as */folder1/folder2/default.html*. The slash characters will be encoded and will not be valid if sent as a request to a Web server. Use the **encodeURI** method if the string contains more than a single URI component.

Requirements

Version 5.5

See Also

decodeURI Method | decodeURIComponent Method

Applies To: Global Object

escape Method

Returns an encoded **String** object that can be read on all computers.

```
function escape(charString : String) : String
```

Arguments

charString
Required. Any **String** object or literal to be encoded.

Remarks

The **escape** method returns a string value (in Unicode format) that contains the contents of *charstring*. All spaces, punctuation, accented characters, and any other non-ASCII characters are replaced with **%***xx* encoding, where *xx* is equivalent to the hexadecimal number representing the character. For example, a space is returned as "%20."

Characters with a value greater than 255 are stored using the **%u***xxxx* format.

> **Note**   The **escape** method should not be used to encode Uniform Resource Identifiers (URI). Use **encodeURI** and **encodeURIComponent** methods instead.

Requirements

Version 1

See Also

encodeURI Method | encodeURIComponent Method | String Object | unescape Method

Applies To: Global Object

eval Method

Evaluates JScript code and executes it.

```
function eval(codeString : String [, override : String])
```

Arguments

codeString
Required. A string that contains valid JScript code.
override
Optional. A string that determines which security permissions to apply to the code in *codeString*.

Remarks

The **eval** function allows dynamic execution of JScript source code.

The code passed to the **eval** method is executed in the same context as the call to the **eval** method. Note that new variables or types defined in the **eval** statement are not visible to the enclosing program.

The code passed to the **eval** method is executed in a restricted security context, unless the string "unsafe" is passed as the second parameter. The restricted security context forbids access to system resources, such as the file system, the network, or the user interface. A security exception is generated if the code attempts to access those resources.

When the second parameter of eval is the string "unsafe", the code passed to the **eval** method is executed in the same security context as the calling code. The second parameter is case sensitive, so the strings "Unsafe" or "UnSAfE" will not override the restricted security context.

> **Security Note** Use **eval** in unsafe mode only to execute code strings obtained from trustworthy sources.

Example

For example, the following code initializes the variable `mydate` to a test date or the current date, depending on the value of the `doTest` variable:

```
var doTest : boolean = true;
var dateFn : String;
if(doTest)
   dateFn = "Date(1971,3,8)";
else
   dateFn = "Date()";

var mydate : Date;
eval("mydate = new "+dateFn+";");
print(mydate);
```

The output of this program is:

```
Thu Apr 8 00:00:00 PDT 1971
```

Requirements

Version 1

See Also

String Object

Applies To: Global Object

exec Method

Executes a search on a string using a regular expression pattern, and returns an array containing the results of that search.

```
function exec(str : String) : Array
```

Arguments

str

Required. The **String** object or string literal on which to perform the search.

Remarks

If the **exec** method does not find a match, it returns **null**. If it finds a match, **exec** returns an array, and the properties of the global **RegExp** object are updated to reflect the results of the match. Element zero of the array contains the entire match, while elements 1 – *n* contain any submatches that have occurred within the match. This behavior is identical to the behavior of the **match** method without the global flag (**g**) set.

If the global flag is set for a regular expression, **exec** searches the string beginning at the position indicated by the value of **lastIndex**. If the global flag is not set, **exec** ignores the value of **lastIndex** and searches from the beginning of the string.

The array returned by the **exec** method has three properties, **input**, **index** and **lastIndex.** The **input** property contains the entire searched string. The **index** property contains the position of the matched substring within the complete searched string. The **lastIndex** property contains the position following the last character in the match.

Example

The following example illustrates the use of the **exec** method:

```
function RegExpTest() {
   var s = "";
   var src = "The rain in Spain falls mainly in the plain.";
   // Create regular expression pattern for matching a word.
   var re = /\w+/g;
   var arr;
   // Loop over all the regular expression matches in the string.
   while ((arr = re.exec(src)) != null)
      s += arr.index + "-" + arr.lastIndex + "\t" + arr + "\n";
   return s;
}
```

Requirements

Version 3

See Also

match Method | RegExp Object | Regular Expression Syntax | search method | test Method

Applies To: Regular Expression Object

exp Method

Returns e (the base of natural logarithms) raised to a power.

```
function exp(number : Number) : Number
```

Arguments

number
Required. A numeric expression.

Remarks

The return value is e^{number}. The constant *e* is the base of natural logarithms, approximately equal to 2.178 and *number* is the supplied argument.

Requirements

Version 1

See Also

E Property

Applies To: Math Object

Methods (F-I)

A method is a function that is a member of an object. Following are methods whose names begin with letters f through i.

In This Section

fixed Method
  Places HTML <TT> tags around text in a **String** object.
floor Method
  Returns the greatest integer less than or equal to its numeric argument.
fontcolor Method
  Places an HTML <FONT> tag with the COLOR attribute around the text in a **String** object.
fontsize Method
  Places an HTML <FONT> tag with the SIZE attribute around the text in a **String** object.
fromCharCode Method
  Returns a string from a number of Unicode character values.
getDate Method
  Returns the day of the month value in a **Date** object using local time.
getDay Method
  Returns the day of the week value in a **Date** object using local time.
getFullYear Method
  Returns the year value in the **Date** object using local time.
getHours Method
  Returns the hours value in a **Date** object using local time.
getItem Method
  Returns the item at the specified location.
getMilliseconds Method
  Returns the milliseconds value in a **Date** object using local time.
getMinutes Method
  Returns the minutes value stored in a **Date** object using local time.
getMonth Method
  Returns the month value in the **Date** object using local time.
getSeconds Method
  Returns seconds value stored in a **Date** object using local time.
getTime Method
  Returns the time value in a **Date** object.
getTimezoneOffset Method
  Returns the difference in minutes between the time on the host computer and Coordinated Universal Time (UTC).
getUTCDate Method
  Returns the date value in a **Date** object using Coordinated Universal Time (UTC).
getUTCDay Method
  Returns the day of the week value in a **Date** object using Coordinated Universal Time (UTC).
getUTCFullYear Method
  Returns the year value in a **Date** object using Coordinated Universal Time (UTC).
getUTCHours Method
  Returns the hours value in a **Date** object using Coordinated Universal Time (UTC).
getUTCMilliseconds Method
  Returns the milliseconds value in a **Date** object using Coordinated Universal Time (UTC).
getUTCMinutes Method
  Returns the minutes value in a **Date** object using Coordinated Universal Time (UTC).
getUTCMonth Method
  Returns the month value in a **Date** object using Coordinated Universal Time (UTC).
getUTCSeconds Method
  Returns the seconds value in a **Date** object using Coordinated Universal Time (UTC).
getVarDate Method
  Returns the VT_DATE value in a **Date** object.
getYear Method
  Returns the year value in a **Date** object. (This method is obsolete; use the **getFullYear** method instead.)
hasOwnProperty Method
  Returns a Boolean value indicating whether an object has a property with the specified name.
indexOf Method

Returns the character position where the first occurrence of a substring occurs within a **String** object.

isFinite Method
Returns a Boolean value that indicates if a supplied number is finite.

isNaN Method
Returns a Boolean value that indicates whether a value is the reserved value **NaN** (not a number).

isPrototypeOf Method
Returns a Boolean value indicating whether an object exists in another object's prototype chain.

italics Method
Places HTML <I> tags around text in a **String** object.

item Method
Returns the current item in the collection.

Related Sections

JScript Reference
Lists elements that comprise JScript Language Reference and links to topics that explain the details behind the proper use of language elements.

Methods
Lists the methods, classified alphabetically, available in JScript, and links to each category of methods.

Objects
Explains the concept of objects in JScript, how objects are related to properties and methods, and links to topics that provide more detail about the objects that JScript supports.

fixed Method

Returns a string with HTML <TT> tags around text in a **String** object.

```
function fixed() : String
```

Remarks

No checking is done to see if the tag has already been applied to the string.

Example

The following example demonstrates how the **fixed** method works:

```
var strVariable = "This is a string object";
strVariable = strVariable.fixed();
```

The value of `strVariable` after the last statement is:

```
<TT>This is a string object</TT>
```

Requirements

Version 1

See Also

Methods

Applies To: String Object

floor Method

Returns the greatest integer less than or equal to its numeric argument.

```
function floor(number : Number) : Number
```

Arguments

number
Required. A numeric expression.

Remarks

The return value is an integer value equal to the greatest integer less than or equal to its numeric argument.

Requirements

Version 1

See Also

ceil Method

Applies To: Math Object

fontcolor Method

Returns a string with an HTML <FONT> tag with the COLOR attribute around the text in a **String** object.

```
function fontcolor(colorVal : String) : String
```

Arguments

colorVal
Required. String value containing a color value. This can either be the hexadecimal value for a color, or the predefined name for a color.

Remarks

Valid predefined color names depend on your JScript host (browser, server, and so forth). They may also vary from version to version of your host. Check your host documentation for more information.

No checking is done to see if the tag has already been applied to the string.

Example

The following example demonstrates the **fontcolor** method:

```
var strVariable = "This is a string";
strVariable = strVariable.fontcolor("red");
```

The value of `strVariable` after the last statement is:

```
<FONT COLOR="RED">This is a string</FONT>
```

Requirements

Version 1

See Also

fontsize Method

Applies To: String Object

fontsize Method

Returns a string with an HTML <FONT> tag with the SIZE attribute around the text in a **String** object.

```
function fontsize(intSize : Number) : String
```

Arguments

intSize

Required. Integer value that specifies the size of the text.

Remarks

Valid integer values depend on your Microsoft JScript host. See your host documentation for more information.

No checking is done to see if the tag has already been applied to the string.

Example

The following example demonstrates the **fontsize** method:

```
var strVariable = "This is a string";
strVariable = strVariable.fontsize(-1);
```

The value of `strVariable` after the last statement is:

```
<FONT SIZE="-1">This is a string</FONT>
```

Requirements

Version 1

See Also

fontcolor Method

Applies To: String Object

fromCharCode Method

Returns a string from a number of Unicode character values.

```
function fromCharCode([code1 : Number [, ... [, codeN : Number]]]]) : String
```

Arguments

code1, ... , *codeN*
  Optional. A series of Unicode character values to convert to a string. If no arguments are supplied, the result is the empty string.

Remarks

The **fromCharCode** method is called from the global String object.

Example

In the following example, `test` is assigned the string "plain":

```
var test = String.fromCharCode(112, 108, 97, 105, 110);
```

Requirements

Version 3

See Also

charCodeAt Method

Applies To: String Object

getDate Method

Returns the day of the month value in a **Date** object using local time.

```
function getDate() : Number
```

Remarks

To get the date value using Coordinated Universal Time (UTC), use the **getUTCDate** method.

The return value is an integer between 1 and 31 that represents the date value in the **Date** object.

Example

The following example illustrates the use of the **getDate** method.

```
function DateDemo(){
   var d, s = "Today's date is: ";
   d = new Date();
   s += (d.getMonth() + 1) + "/";
   s += d.getDate() + "/";
   s += d.getYear();
   return(s);
}
```

Requirements

Version 1

See Also

getUTCDate Method | setDate Method | setUTCDate Method

Applies To: Date Object

getDay Method

Returns the day of the week value in a **Date** object using local time.

```
function getDay() : Number
```

Remarks

To get the day using Coordinated Universal Time (UTC), use the **getUTCDay** method.

The value returned from the **getDay** method is an integer between 0 and 6 representing the day of the week and corresponds to a day of the week as follows:

Value	Day of the Week
0	Sunday
1	Monday
2	Tuesday
3	Wednesday
4	Thursday
5	Friday
6	Saturday

The following example illustrates the use of the **getDay** method.

```
function DateDemo(){
   var d, day, x, s = "Today is: ";
   var x = new Array("Sunday", "Monday", "Tuesday");
   var x = x.concat("Wednesday","Thursday", "Friday");
   var x = x.concat("Saturday");
   d = new Date();
   day = d.getDay();
   return(s += x[day]);
}
```

Requirements

Version 1

See Also

getUTCDay Method

Applies To: Date Object

getFullYear Method

Returns the year value in the **Date** object using local time.

```
function .getFullYear() : Number
```

Remarks

To get the year using Coordinated Universal Time (UTC), use the **getUTCFullYear** method.

The **getFullYear** method returns the year as an absolute number. For example, the year 1976 is returned as 1976. This avoids the year 2000 problem where dates beginning with January 1, 2000 are confused with those beginning with January 1, 1900.

The following example illustrates the use of the **getFullYear** method.

```
function DateDemo(){
   var d, s = "Today's date is: ";
   d = new Date();
   s += (d.getMonth() + 1) + "/";
   s += d.getDate() + "/";
   s += d.getFullYear();
   return(s);
}
```

Requirements

Version 3

See Also

getUTCFullYear Method | setFullYear Method | setUTCFullYear Method

Applies To: Date Object

getHours Method

Returns the hours value in a **Date** object using local time.

```
function getHours() : Number
```

Remarks

To get the hours value using Coordinated Universal Time (UTC), use the **getUTCHours** method.

The **getHours** method returns an integer between 0 and 23, indicating the number of hours since midnight. A zero occurs in two situations: the time is before 1:00:00 am, or the time was not stored in the **Date** object when the object was created. The only way to determine which situation you have is to also check the minutes and seconds for zero values. If they are all zeroes, it is nearly certain that the time was not stored in the **Date** object.

The following example illustrates the use of the **getHours** method.

```
function TimeDemo(){
   var d, s = "The current local time is: ";
   var c = ":";
   d = new Date();
   s += d.getHours() + c;
   s += d.getMinutes() + c;
   s += d.getSeconds() + c;
   s += d.getMilliseconds();
   return(s);
}
```

Requirements

Version 1

See Also

getUTCHours Method | setHours Method | setUTCHours Method

Applies To: Date Object

getItem Method

Returns the item at the specified location in a VBArray object.

```
function getItem(dimension1 : Number [, ...], dimensionN : Number) : Object
```

Arguments

dimension1, ..., dimensionN
Specifies the exact location of the desired element of the VBArray. The number of arguments must match the number of dimensions in the VBArray.

Example

The following example consists of three parts. The first part is VBScript code to create a Visual Basic safe array. The second part is JScript code that iterates the Visual Basic safe array and prints out the contents of each element. Both of these parts go into the <HEAD> section of an HTML page. The third part is the JScript code that goes in the <BODY> section to run the other two parts.

```
<HEAD>
<SCRIPT LANGUAGE="VBScript">
<!--
Function CreateVBArray()
   Dim i, j, k
   Dim a(2, 2)
   k = 1
   For i = 0 To 2
      For j = 0 To 2
         a(i, j) = k
         document.writeln(k)
         k = k + 1
      Next
      document.writeln("<BR>")
   Next
   CreateVBArray = a
End Function
-->
</SCRIPT>
<SCRIPT LANGUAGE="JScript">
<!--
function GetItemTest(vbarray)
{
   var i, j;
   var a = new VBArray(vbarray);
   for (i = 0; i <= 2; i++)
   {
      for (j =0; j <= 2; j++)
      {
         document.writeln(a.getItem(i, j));
      }
   }
}-->
</SCRIPT>
</HEAD>
&ltBODY>
<SCRIPT LANGUAGE="JScript">
<!--
   GetItemTest(CreateVBArray());
-->
</SCRIPT>
</BODY>
```

Requirements

Version 1

See Also

dimensions Method | lbound Method | toArray Method | ubound Method

Applies To: VBArray Object

getMilliseconds Method

Returns the milliseconds value in a **Date** object using local time.

```
function getMilliseconds() : Number
```

Remarks

To get the number of milliseconds in Coordinated Universal Time (UTC), use the **getUTCMilliseconds** method.

The millisecond value returned can range from 0-999.

Example

The following example illustrates the use of the **getMilliseconds** method.

```
function TimeDemo(){
   var d, s = "The current local time is: ";
   var c = ":";
   d = new Date();
   s += d.getHours() + c;
   s += d.getMinutes() + c;
   s += d.getSeconds() + c;
   s += d.getMilliseconds();
   return(s);
}
```

Requirements

Version 3

See Also

getUTCMilliseconds Method | setMilliseconds Method | setUTCMilliseconds Method

Applies To: Date Object

getMinutes Method

Returns the minutes value in a **Date** object using local time.

```
function getMinutes() : Number
```

Remarks

To get the minutes value using Coordinated Universal Time (UTC), use the **getUTCMinutes** method.

The **getMinutes** method returns an integer between 0 and 59 equal to the minute's value stored in the **Date** object. A zero is returned in two situations: when the time is less than one minute after the hour, or when the time was not stored in the **Date** object when the object was created. The only way to determine which situation you have is to also check the hours and seconds for zero values. If they are all zeroes, it is nearly certain that the time was not stored in the **Date** object.

Example

The following example illustrates the use of the **getMinutes** method.

```
function TimeDemo(){
   var d, s = "The current local time is: ";
   var c = ":";
   d = new Date();
   s += d.getHours() + c;
   s += d.getMinutes() + c;
   s += d.getSeconds() + c;
   s += d.getMilliseconds();
   return(s);
}
```

Requirements

Version 3

See Also

getUTCMinutes Method | setMinutes Method | setUTCMinutes Method

Applies To: Date Object

getMonth Method

Returns the month value in the **Date** object using local time.

```
function getMonth() : Number
```

Remarks

To get the month value using Coordinated Universal Time (UTC), use the **getUTCMonth** method.

The **getMonth** method returns an integer between 0 and 11 indicating the month value in the **Date** object. The integer returned is not the traditional number used to indicate the month. It is one less. If "Jan 5, 1996 08:47:00" is stored in a **Date** object, **getMonth** returns 0.

Example

The following example illustrates the use of the **getMonth** method.

```
function DateDemo(){
   var d, s = "Today's date is: ";
   d = new Date();
   s += (d.getMonth() + 1) + "/";
   s += d.getDate() + "/";
   s += d.getYear();
   return(s);
}
```

Requirements

Version 1

See Also

getUTCMonth Method | setMonth Method | setUTCMonth Method

Applies To: Date Object

getSeconds Method

Returns the seconds value in a **Date** object using local time.

```
function getSeconds() : Number
```

Remarks

To get the seconds value using Coordinated Universal Time (UTC), use the **getUTCSeconds** method.

The **getSeconds** method returns an integer between 0 and 59 indicating the second's value of the indicated **Date** object. A zero is returned in two situations. One occurs when the time is less than one second into the current minute. The other occurs when the time was not stored in the **Date** object when the object was created. The only way to determine which situation you have is to also check the hours and minutes for zero values. If they are all zeroes, it is nearly certain that the time was not stored in the **Date** object.

Example

The following example illustrates the use of the **getSeconds** method.

```
function TimeDemo(){
   var d, s = "The current local time is: ";
   var c = ":";
   d = new Date();
   s += d.getHours() + c;
   s += d.getMinutes() + c;
   s += d.getSeconds() + c;
   s += d.getMilliseconds();
   return(s);
}
```

Requirements

Version 1

See Also

getUTCSeconds Method | setSeconds Method | setUTCSeconds Method

Applies To: Date Object

getTime Method

Returns the time value in a **Date** object.

```
function getTime() : Number
```

Remarks

The **getTime** method returns an integer value representing the number of milliseconds between midnight, January 1, 1970 and the time value in the **Date** object. The range of dates is approximately 285,616 years from either side of midnight, January 1, 1970. Negative numbers indicate dates prior to 1970.

When doing multiple date and time calculations, it is frequently useful to define variables equal to the number of milliseconds in a day, hour, or minute. For example:

```
var MinMilli = 1000 * 60
var HrMilli = MinMilli * 60
var DyMilli = HrMilli * 24
```

Example

The following example illustrates the use of the **getTime** method.

```
function GetTimeTest(){
   var d, s, t;
   var MinMilli = 1000 * 60;
   var HrMilli = MinMilli * 60;
   var DyMilli = HrMilli * 24;
   d = new Date();
   t = d.getTime();
   s = "It's been "
   s += Math.round(t / DyMilli) + " days since 1/1/70";
   return(s);
}
```

Requirements

Version 1

See Also

setTime Method

Applies To: Date Object

getTimezoneOffset Method

Returns the difference in minutes between the time on the host computer and Coordinated Universal Time (UTC).

```
function getTimezoneOffset() : Number
```

Remarks

The **getTimezoneOffset** method returns an integer value representing the number of minutes between the time on the current machine and UTC. These values are appropriate to the computer the script is executed on. If it is called from a server script, the return value is appropriate to the server. If it is called from a client script, the return value is appropriate to the client.

This number will be positive if you are behind UTC (for example, Pacific Daylight Time), and negative if you are ahead of UTC (for example, Japan).

For example, suppose a client in Los Angeles contacts a server in New York City on December 1. **getTimezoneOffset** returns 480 if executed on the client, or 300 if executed on the server.

Example

The following example illustrates the use of the **getTimezoneOffset** method.

```
function TZDemo(){
   var d, tz, s = "The current local time is ";
   d = new Date();
   tz = d.getTimezoneOffset();
   if (tz < 0)
      s += tz / 60 + " hours before UTC";
   else if (tz == 0)
      s += "UTC";
   else
      s += tz / 60 + " hours after UTC";
   return(s);
}
```

Requirements

Version 1

See Also

Methods

Applies To: Date Object

getUTCDate Method

Returns the date in a **Date** object using Coordinated Universal Time (UTC).

```
function getUTCDate() : Number
```

Remarks

To get the date using local time, use the **getDate** method.

The return value is an integer between 1 and 31 that represents the date value in the **Date** object.

Example

The following example illustrates the use of the **getUTCDate** method.

```
function UTCDateDemo(){
   var d, s = "Today's UTC date is: ";
   d = new Date();
   s += (d.getUTCMonth() + 1) + "/";
   s += d.getUTCDate() + "/";
   s += d.getUTCFullYear();
   return(s);
}
```

Requirements

Version 3

See Also

getDate Method | setDate Method | setUTCDate Method

Applies To: Date Object

getUTCDay Method

Returns the day of the week value in a **Date** object using Coordinated Universal Time (UTC).

```
function getUTCDay() : Number
```

Remarks

To get the day of the week using local time, use the **getDate** method.

The value returned by the **getUTCDay** method is an integer between 0 and 6 representing the day of the week and corresponds to a day of the week as follows:

Value	Day of the Week
0	Sunday
1	Monday
2	Tuesday
3	Wednesday
4	Thursday
5	Friday
6	Saturday

Example

The following example illustrates the use of the **getUTCDay** method.

```
function DateDemo(){
   var d, day, x, s = "Today is ";
   var x = new Array("Sunday", "Monday", "Tuesday");
   x = x.concat("Wednesday","Thursday", "Friday");
   x = x.concat("Saturday");
   d = new Date();
   day = d.getUTCDay();
   return(s += x[day] + " in UTC");
}
```

Requirements

Version 3

See Also

getDay Method

Applies To: Date Object

getUTCFullYear Method

Returns the year value in a **Date** object using Coordinated Universal Time (UTC).

```
function getUTCFullYear() : Number
```

Remarks

To get the year using local time, use the **getFullYear** method.

The **getUTCFullYear** method returns the year as an absolute number. This avoids the year 2000 problem where dates beginning with January 1, 2000 are confused with those beginning with January 1, 1900.

Example

The following example illustrates the use of the **getUTCFullYear** method.

```
function UTCDateDemo(){
   var d, s = "Today's UTC date is: ";
   d = new Date();
   s += (d.getUTCMonth() + 1) + "/";
   s += d.getUTCDate() + "/";
   s += d.getUTCFullYear();
   return(s);
}
```

Requirements

Version 3

See Also

getFullYear Method | setFullYear Method | setUTCFullYear Method

Applies To: Date Object

getUTCHours Method

Returns the hours value in a **Date** object using Coordinated Universal Time (UTC).

```
function getUTCHours() : Number
```

Remarks

To get the number of hours elapsed since midnight using local time, use the **getHours** method.

The **getUTCHours** method returns an integer between 0 and 23 indicating the number of hours since midnight. A zero occurs in two situations: the time is before 1:00:00 A.M., or a time was not stored in the **Date** object when the object was created. The only way to determine which situation you have is to also check the minutes and seconds for zero values. If they are all zeroes, it is nearly certain that the time was not stored in the **Date** object.

Example

The following example illustrates the use of the **getUTCHours** method.

```
function UTCTimeDemo(){
   var d, s = "Current Coordinated Universal Time (UTC) is: ";
   var c = ":";
   d = new Date();
   s += d.getUTCHours() + c;
   s += d.getUTCMinutes() + c;
   s += d.getUTCSeconds() + c;
   s += d.getUTCMilliseconds();
   return(s);
}
```

Requirements

Version 3

See Also

getHours Method | setHours Method | setUTCHours Method

Applies To: Date Object

getUTCMilliseconds Method

Returns the milliseconds value in a **Date** object using Coordinated Universal Time (UTC).

```
function getUTCMilliseconds() : Number
```

Remarks

To get the number of milliseconds in local time, use the **getMilliseconds** method.

The millisecond value returned can range from 0-999.

Example

The following example illustrates the use of the **getUTCMilliseconds** method.

```
function UTCTimeDemo(){
   var d, s = "Current Coordinated Universal Time (UTC) is: ";
   var c = ":";
   d = new Date();
   s += d.getUTCHours() + c;
   s += d.getUTCMinutes() + c;
   s += d.getUTCSeconds() + c;
   s += d.getUTCMilliseconds();
   return(s);
}
```

Requirements

Version 3

See Also

getMilliseconds Method | setMilliseconds Method | setUTCMilliseconds Method

Applies To: Date Object

getUTCMinutes Method

Returns the minutes value in a **Date** object using Coordinated Universal Time (UTC).

```
function getUTCMinutes() : Number
```

Remarks

To get the number of minutes stored using local time, use the **getMinutes** method.

The **getUTCMinutes** method returns an integer between 0 and 59 equal to the number of minute's value in the **Date** object. A zero occurs in two situations: the time is less than one minute after the hour, or a time was not stored in the **Date** object when the object was created. The only way to determine which situation you have is to also check the hours and seconds for zero values. If they are all zeroes, it is nearly certain that the time was not stored in the **Date** object.

Example

The following example illustrates the use of the **getUTCMinutes** method.

```
function UTCTimeDemo()
{
   var d, s = "Current Coordinated Universal Time (UTC) is: ";
   var c = ":";
   d = new Date();
   s += d.getUTCHours() + c;
   s += d.getUTCMinutes() + c;
   s += d.getUTCSeconds() + c;
   s += d.getUTCMilliseconds();
   return(s);
}
```

Requirements

Version 3

See Also

getMinutes Method | setMinutes Method | setUTCMinutes Method

Applies To: Date Object

getUTCMonth Method

Returns the month value in a **Date** object using Coordinated Universal Time (UTC).

```
function getUTCMonth(): Number
```

Remarks

To get the month in local time, use the **getMonth** method.

The **getUTCMonth** method returns an integer between 0 and 11 indicating the month value in the Date object. The integer returned is not the traditional number used to indicate the month. It is one less. If "Jan 5, 1996 08:47:00.0" is stored in a **Date** object, **getUTCMonth** returns 0.

Example

The following example illustrates the use of the **getUTCMonth** method.

```
function UTCDateDemo(){
   var d, s = "Today's UTC date is: ";
   d = new Date();
   s += (d.getUTCMonth() + 1) + "/";
   s += d.getUTCDate() + "/";
   s += d.getUTCFullYear();
   return(s);
}
```

Requirements

Version 3

See Also

getMonth Method | setMonth Method | setUTCMonth Method

Applies To: Date Object

getUTCSeconds Method

Returns the seconds value in a **Date** object using Coordinated Universal Time (UTC).

```
function getUTCSeconds(): Number
```

Remarks

To get the number of seconds in local time, use the **getSeconds** method.

The **getUTCSeconds** method returns an integer between 0 and 59 indicating the second's value of the indicated **Date** object. A zero occurs in two situations: the time is less than one second into the current minute, or a time was not stored in the **Date** object when the object was created. The only way to determine which situation you have is to also check the minutes and hours for zero values. If they are all zeroes, it is nearly certain that the time was not stored in the **Date** object.

Example

The following example illustrates the use of the **getUTCSeconds** method.

```
function UTCTimeDemo(){
   var d, s = "Current Coordinated Universal Time (UTC) is: ";
   var c = ":";
   d = new Date();
   s += d.getUTCHours() + c;
   s += d.getUTCMinutes() + c;
   s += d.getUTCSeconds() + c;
   s += d.getUTCMilliseconds();
   return(s);
}
```

Requirements

Version 3

See Also

getSeconds Method | setSeconds Method | setUTCSeconds Method

Applies To: Date Object

getVarDate Method

Returns the VT_DATE value in a **Date** object.

```
function getVarDate(): System.DateTime
```

Remarks

The **getVarDate** method is used when interacting with COM objects, ActiveX® objects or other objects that accept and return date values in VT_DATE format, such as Visual Basic and VBScript. The actual format is dependent on regional settings and should not be relied upon within JScript.

Requirements

Version 3

See Also

getDate Method | parse Method

Applies To: Date Object

getYear Method

Returns the year value in a **Date** object.

```
function getYear(): Number
```

Remarks

This method is obsolete, and is provided for backwards compatibility only. Use the **getFullYear** method instead.

For the years 1900 though 1999, the year is a 2-digit integer value returned as the difference between the stored year and 1900. For dates outside that period, the 4-digit year is returned. For example, 1996 is returned as 96, but 1825 and 2025 are returned as-is.

> **Note** For JScript version 1.0, **getYear** returns a value that is the result of the subtraction of 1900 from the year value in the provided **Date** object, regardless of the value of the year. For example, the year 1899 is returned as -1 and the year 2000 is returned as 100.

Example

The following example illustrates the use of the **getYear** method:

```
function DateDemo(){
   var d, s = "Today's date is: ";
   d = new Date();
   s += (d.getMonth() + 1) + "/";
   s += d.getDate() + "/";
   s += d.getYear();
   return(s);
}
```

Requirements

Version 1

See Also

getFullYear Method | getUTCFullYear Method | setFullYear Method | setUTCFullYear Method | setYear Method

Applies To: Date Object

hasOwnProperty Method

Returns a Boolean value indicating whether an object has a property with the specified name.

```
function hasOwnProperty(proName : String) : Boolean
```

Arguments

proName
Required. String value of a property name.

Remarks

The **hasOwnProperty** method returns **true** if the object has a property of the specified name, **false** if it does not. This method does not check if the property exists in the object's prototype chain; the property must be a member of the object itself.

Example

In the following example, all **String** objects share a common split method.

```
var s = new String("JScript");
print(s.hasOwnProperty("split"));
print(String.prototype.hasOwnProperty("split"));
```

The output of this program is:

```
false
true
```

Requirements

Version 5.5

See Also

in Operator

Applies To: Object Object

indexOf Method

Returns the character position where the first occurrence of a substring occurs within a **String** object.

```
function indexOf(subString : String [, startIndex : Number]) : Number
```

Arguments

subString
  Required. Substring to search for within the **String** object.
startIndex
  Optional. Integer value specifying the index to begin searching within the **String** object. If omitted, searching starts at the beginning of the string.

Remarks

The **indexOf** method returns an integer value indicating the beginning of the substring within the **String** object. If the substring is not found, a -1 is returned.

If *startIndex* is negative, *startIndex* is treated as zero. If it is larger than the greatest character position index, it is treated as the largest possible index.

Searching is performed from left to right. Otherwise, this method is identical to **lastIndexOf**.

Example

The following example illustrates the use of the **indexOf** method.

```
function IndexDemo(str2){
   var str1 = "BABEBIBOBUBABEBIBOBU"
   var s = str1.indexOf(str2);
   return(s);
}
```

Requirements

Version 1

See Also

lastIndexOf Method

Applies To: String Object

isFinite Method

Returns a Boolean value that indicates if a supplied number is finite.

```
function isFinite(number : Number) : Boolean
```

Arguments

number
Required. A numeric value.

Remarks

The **isFinite** method returns **true** if *number* is any value other than **NaN**, negative infinity, or positive infinity. In those three cases, it returns **false**.

Requirements

Version 3

See Also

isNaN Method

Applies To: Global Object

isNaN Method

Returns a Boolean value that indicates whether a value is the reserved value **NaN** (not a number).

```
function isNaN(number : Number) : Boolean
```

Arguments

number
Required. A numeric value.

Remarks

The **isNaN** function returns **true** if the value is **NaN**, and **false** otherwise. You typically use this function to test return values from the **parseInt** and **parseFloat** methods.

Alternatively, a variable could be compared to itself. If it compares as unequal, it is **NaN**. This is because **NaN** is the only value that is not equal to itself.

Requirements

Version 1

See Also

isFinite Method | NaN Property (Global) | parseFloat Method | parseInt Method

Applies To: Global Object

isPrototypeOf Method

Returns a Boolean value indicating whether an object exists in the prototype chain of another object.

```
function isPrototypeOf(obj : Object) : Boolean
```

Arguments

obj
Required. An object whose prototype chain is to be checked.

Remarks

The **isPrototypeOf** method returns **true** if *obj* has the current object in its prototype chain. The prototype chain is used to share functionality between instances of the same object type. The **isPrototypeOf** method returns **false** when *obj* is not an object or when the current object does not appear in the prototype chain of the *obj*.

Example

The following example illustrates the use of the **isPrototypeof** method.

```
function test(){
   var re = new RegExp();                          //Initialize variable.
   return (RegExp.prototype.isPrototypeOf(re));  //Return true.
}
```

Requirements

Version 5.5

See Also

Methods

Applies To: Object Object

italics Method

Returns a string with HTML <I> tags around text in a string object.

```
function italics() : String
```

Remarks

No checking is done to see if the tag has already been applied to the string.

Example

The following example demonstrates how the **italics** method works:

```
var strVariable = "This is a string";
strVariable = strVariable.italics();
```

The value of `strVariable` after the last statement is:

```
<I>This is a string</I>
```

Requirements

Version 1

See Also

bold Method

Applies To: String Object

item Method

Returns the current item in the collection.

```
function item() : Number
```

Remarks

The **item** method returns the current item in an **Enumerator** object. If the collection is empty or the current item is undefined, it returns **undefined**.

Example

In following code, the **item** method is used to return a member of the **Drives** collection.

```
function ShowDriveList(){
   var fso, s, n, e, x;
   fso = new ActiveXObject("Scripting.FileSystemObject");
   e = new Enumerator(fso.Drives);
   s = "";
   for (; !e.atEnd(); e.moveNext())
   {
      x = e.item();
      s = s + x.DriveLetter;
      s += " - ";
      if (x.DriveType == 3)
         n = x.ShareName;
      else if (x.IsReady)
         n = x.VolumeName;
      else
         n = "[Drive not ready]";
      s +=  n + "<br>";
   }
   return(s);
}
```

Requirements

Version 3

See Also

atEnd Method | moveFirst Method | moveNext Method

Applies To: Enumerator Object

Methods (J-R)

A method is a function that is a member of an object. Following are methods whose names begin with letters j through r.

In This Section

join Method
: Returns a **String** object consisting of all the concatenated elements of an array.

lastIndexOf Method
: Returns the last occurrence of a substring within a **String** object.

lbound Method
: Returns the lowest index value used in the specified dimension of a VBArray.

link Method
: Places an HTML anchor with an HREF attribute around the text in a **String** object.

localeCompare Method
: Returns a value indicating whether two strings are equivalent in the current locale.

log Method
: Returns the natural logarithm of a number.

match Method
: Returns, as an array, the results of a search on a string using a supplied **Regular Expression** object.

max Method
: Returns the greater of two supplied numeric expressions

min Method
: Returns the lesser of two supplied numbers.

moveFirst Method
: Resets the current item in the collection to the first item.

moveNext Method
: Moves the current item to the next item in the collection.

parse Method
: Parses a string containing a date, and returns the number of milliseconds between that date and midnight, January 1, 1970.

parseFloat Method
: Returns a floating-point number converted from a string.

parseInt Method
: Returns an integer converted from a string.

pop Method
: Removes the last element from an array and returns it.

pow Method
: Returns the value of a base expression raised to a specified power.

push Method
: Appends new elements to an array and returns the new length of the array.

random Method
: Returns a pseudorandom number between 0 and 1.

replace Method
: Returns a copy of a string with text replaced using a regular expression.

reverse Method
: Returns an **Array** object with the elements reversed.

round Method
: Returns a specified numeric expression rounded to the nearest integer.

Related Sections

JScript Reference
: Lists elements that comprise JScript Language Reference and links to topics that explain the details behind the proper use of language elements.

Methods
: Lists the methods, classified alphabetically, available in JScript, and links to each category of methods.

Objects
: Explains the concept of objects in JScript, how objects are related to properties and methods, and links to topics that provide more detail about the objects that JScript supports.

join Method

Returns a string value consisting of all the elements of an array concatenated together and separated by the specified separator character.

```
function join(separator : String) : String
```

Arguments

separator

Required. A string that is used to separate one element of an array from the next in the resulting **String** object. If omitted, the array elements are separated with a comma.

Remarks

If any element of the array is **undefined** or **null**, it is treated as an empty string.

Example

The following example illustrates the use of the **join** method.

```
function JoinDemo(){
   var a, b;
   a = new Array(0,1,2,3,4);
   b = a.join("-");
   return(b);
}
```

Requirements

Version 2

See Also

String Object

Applies To: Array Object

lastIndexOf Method

Returns the index of the last occurrence of a substring within a **String** object.

```
function lastIndexOf(substring : String [, startindex : Number ]) : Number
```

Arguments

substring
Required. The substring to search for within the **String** object.

startindex
Optional. Integer value specifying the index to begin searching within the **String** object. If omitted, searching begins at the end of the string.

Remarks

The **lastIndexOf** method returns an integer value indicating the beginning of the substring within the **String** object. If the substring is not found, a -1 is returned.

If *startindex* is negative, *startindex* is treated as zero. If it is larger than the greatest character position index, it is treated as the largest possible index.

Searching is performed right to left. Otherwise, this method is identical to **indexOf**.

Example

The following example illustrates the use of the **lastIndexOf** method.

```
function lastIndexDemo(str2) {
   var str1 = "BABEBIBOBUBABEBIBOBU"
   var s = str1.lastIndexOf(str2);
   return(s);
}
```

Requirements

Version 1

See Also

indexOf Method

Applies To: String Object

lbound Method

Returns the lowest index value used in the specified dimension of a VBArray.

```
function lbound([dimension : Number]) : Object
```

Arguments

dimension
Optional. The dimension of the VBArray for which the lower bound index is wanted. If omitted, **lbound** behaves as if a 1 was passed.

Remarks

If the VBArray is empty, the **lbound** method returns undefined. If *dimension* is greater than the number of dimensions in the VBArray, or is negative, the method generates a "Subscript out of range" error.

Example

The following example consists of three parts. The first part is VBScript code to create a Visual Basic safe array. The second part is JScript code that determines the number of dimensions in the safe array and the lower bound of each dimension. Since the safe array is created in VBScript rather than Visual Basic, the lower bound will always be zero. Both of these parts go into the <HEAD> section of an HTML page. The third part is the JScript code that goes in the <BODY> section to run the other two parts.

```
<HEAD>
<SCRIPT LANGUAGE="VBScript">
<!--
Function CreateVBArray()
   Dim i, j, k
   Dim a(2, 2)
   k = 1
   For i = 0 To 2
      For j = 0 To 2
         a(j, i) = k
         k = k + 1
      Next
   Next
   CreateVBArray = a
End Function
-->
</SCRIPT>

<SCRIPT LANGUAGE="JScript">
<!--
function VBArrayTest(vba){
   var i, s;
   var a = new VBArray(vba);
   for (i = 1; i <= a.dimensions(); i++)
   {
      s = "The lower bound of dimension ";
      s += i + " is ";
      s += a.lbound(i)+ ".<BR>";
      return(s);
   }
}
-->
</SCRIPT>
</HEAD>

<BODY>
<SCRIPT language="jscript">
   document.write(VBArrayTest(CreateVBArray()));
</SCRIPT>
```

```
</BODY>
```

Requirements

Version 3

See Also

dimensions Method | getItem Method | toArray Method | ubound Method

Applies To: VBArray Object

link Method

Returns as string with an HTML anchor and an HREF attribute around the text in a **String** object.

```
function link(linkstring : String) : String
```

Remarks

Call the **link** method to create a hyperlink out of a **String** object.

No checking is done to see if the tag has already been applied to the string.

Example

The following is an example of how the method accomplishes this:

```
var strVariable = "This is a hyperlink";
strVariable = strVariable.link("http://www.microsoft.com");
```

The value of `strVariable` after the last statement is:

```
<A HREF="http://www.microsoft.com">This is a hyperlink</A>
```

Requirements

Version 1

See Also

anchor Method

Applies To: String Object

localeCompare Method

Returns a value indicating whether two strings are equivalent in the current locale.

```
function localeCompare(stringExp : String) : Number
```

Arguments

stringExp
Required. String to compare against the current string object.

Remarks

The **localeCompare** performs a locale-sensitive string comparison of the current string object and the *stringExp* and returns -1, 0, or +1, depending on the sort order of the system default locale.

If the current string object sorts before *stringExp*, **localeCompare** returns –1; if the current string sorts after *stringExp*, +1 is returned. A return value of zero means that the two strings are equivalent.

Requirements

Version 5.5

See Also

ToLocaleString Method

Applies To: String Object

log Method

Returns the natural logarithm of a number.

```
function log(number : Number) : Number
```

Arguments

number
Required. A numeric value.

Remarks

The return value is the natural logarithm of *number*. The base is *e*.

Requirements

Version 1

See Also

Methods

Applies To: Math Object

match Method

Executes a search on a string using a regular expression pattern, and returns an array containing the results of that search.

```
function match(rgExp : RegExp) : Array
```

Arguments

rgExp
Required. An instance of a **Regular Expression** object containing the regular expression pattern and applicable flags. Can also be a variable name or string literal containing the regular expression pattern and flags.

Remarks

If the **match** method does not find a match, it returns **null**. If it finds a match, **match** returns an array, and the properties of the global **RegExp** object are updated to reflect the results of the match.

The array returned by the **match** method has three properties, **input**, **index** and **lastIndex.** The **input** property contains the entire searched string. The **index** property contains the position of the matched substring within the complete searched string. The **lastIndex** property contains the position following the last character in the last match.

If the global flag (**g**) is not set, Element zero of the array contains the entire match, while elements 1 – *n* contain any submatches that have occurred within the match. This behavior is identical to the behavior of the **exec** method without the global flag set. If the global flag is set, elements 0 - *n* contain all matches that occurred.

Example

The following example illustrates the use of the **match** method.

```
function MatchDemo(){
   var r, re;         //Declare variables.
   var s = "The rain in Spain falls mainly in the plain";
   re = /ain/i;    //Create regular expression pattern.
   r = s.match(re);   //Attempt match on search string.
   return(r);         //Return first occurrence of "ain".
}
```

This example illustrates the use of the match method with the **g** flag set.

```
function MatchDemo(){
   var r, re;         //Declare variables.
   var s = "The rain in Spain falls mainly in the plain";
   re = /ain/ig;      //Create regular expression pattern.
   r = s.match(re);   //Attempt match on search string.
   return(r);         //Return array containing all four
                      // occurrences of "ain".
}
```

The following lines of code illustrate the use of a string literal with the **match** method.

```
var r, re = "Spain";
r = "The rain in Spain".replace(re, "Canada");
```

Requirements

Version 3

See Also

exec Method | RegExp Object | replace Method | search Method | test Method

Applies To: String Object

max Method

Returns the greater of zero or more supplied numeric expressions.

```
function max([number1 : Number [, ... [, numberN : Number]]]) : Number
```

Arguments

number1, ... , numberN
Required. Numeric expressions.

Remarks

If no arguments are provided, the return value is equal to **NEGATIVE_INFINITY**. If any argument is **NaN**, the return value is also **NaN**.

Requirements

Version 1

See Also

min Method| NEGATIVE_INFINITY Property

Applies To: Math Object

min Method

Returns the lesser of zero or more supplied numeric expressions.

```
function min([number1 : Number [, ... [, numberN : Number]]]) : Number
```

Arguments

number1, ... , numberN
Required. Numeric expressions.

Remarks

If no arguments are provided, the return value is equal to **POSITIVE_INFINITY**. If any argument is **NaN**, the return value is also **NaN**.

Requirements

Version 1

See Also

max Method | POSITIVE_INFINITY Property

Applies To: Math Object

moveFirst Method

Resets the current item in an **Enumerator** object to the first item.

```
function moveFirst()
```

Remarks

If there are no items in the collection, the current item is set to undefined.

Example

In following example, the **moveFirst** method is used to evaluate members of the **Drives** collection from the beginning of the list:

```
function ShowFirstAvailableDrive(){
   var fso, s, e, x;                //Declare variables.
   fso = new ActiveXObject("Scripting.FileSystemObject");
   e = new Enumerator(fso.Drives);  //Create Enumerator object.
   e.moveFirst();                   //Move to first drive.
   s = "";                          //Initialize s.
   do
   {
      x = e.item();                 //Test for existence of drive.
      if (x.IsReady)                //See if it's ready.
      {
         s = x.DriveLetter + ":";   //Assign 1st drive letter to s.
         break;
      }
      else
         if (e.atEnd())             //See if at the end of the collection.
         {
            s = "No drives are available";
            break;
         }
      e.moveNext();                 //Move to the next drive.
   }
   while (!e.atEnd());              //Do while not at collection end.
   return(s);                       //Return list of available drives.
}
```

Requirements

Version 3

See Also

atEnd Method | item Method | moveNext Method

Applies To: Enumerator Object

moveNext Method

Moves the current item to the next item in the **Enumerator** object.

```
function moveNext()
```

Remarks

If the enumerator is at the end of the collection or the collection is empty, the current item is set to undefined.

In following example, the **moveNext** method is used to move to the next drive in the **Drives** collection:

```
function ShowDriveList(){
   var fso, s, n, e, x;                 //Declare variables.
   fso = new ActiveXObject("Scripting.FileSystemObject");
   e = new Enumerator(fso.Drives);      //Create Enumerator object.
   s = "";                              //Initialize s.
   for (; !e.atEnd(); e.moveNext())
   {
      x = e.item();
      s = s + x.DriveLetter;            //Add drive letter
      s += " - ";                       //Add "-" character.
      if (x.DriveType == 3)
         n = x.ShareName;               //Add share name.
      else if (x.IsReady)
         n = x.VolumeName;              //Add volume name.
      else
         n = "[Drive not ready]";       //Indicate drive not ready.
      s +=   n + "\n";
   }
   return(s);                           //Return drive status.
}
```

Requirements

Version 3

See Also

atEnd Method | item Method | moveFirst Method

Applies To: Enumerator Object

parse Method

Parses a string containing a date, and returns the number of milliseconds between that date and midnight, January 1, 1970.

```
function parse(dateVal : {String | System.DateTime} ) : Number
```

Arguments

dateVal
Required. Either a string containing a date in a format such as "Jan 5, 1996 08:47:00" or a VT_DATE value retrieved from an ActiveX® object or other object.

Remarks

The **parse** method returns an integer value representing the number of milliseconds between midnight, January 1, 1970 and the date supplied in *dateVal*.

The **parse** method is a static method of the **Date** object. Because it is a static method, it is invoked as shown in the following example, rather than invoked as a method of a created **Date** object.

```
var datestring = "November 1, 1997 10:15 AM";
Date.parse(datestring)
```

The following rules govern what the **parse** method can successfully parse:

- Short dates can use either a "/" or "-" date separator, but must follow the month/day/year format, for example "7/20/96".
- Long dates of the form "July 10 1995" can be given with the year, month, and day in any order, and the year in 2-digit or 4-digit form. If you use the 2-digit form, the year must be greater than or equal to 70.
- Any text inside parentheses is treated as a comment. These parentheses may be nested.
- Both commas and spaces are treated as delimiters. Multiple delimiters are permitted.
- Month and day names must have two or more characters. Two character names that are not unique are resolved as the last match. For example, "Ju" is resolved as July, not June.
- The stated day of the week is ignored if it is incorrect given the remainder of the supplied date. For example, "Tuesday November 9 1996" is accepted and parsed even though that date actually falls on a Friday. The resulting **Date** object contains "Friday November 9 1996".
- JScript handles all standard time zones, as well as Universal Coordinated Time (UTC) and Greenwich Mean Time (GMT).
- Colons separate hours, minutes, and seconds, although all need not be specified. "10:", "10:11", and "10:11:12" are all valid.
- If the 24-hour clock is used, it is an error to specify "PM" for times later than 12 noon. For example, "23:15 PM" is an error.
- A string containing an invalid date is an error. For example, a string containing two years or two months is an error.

Example

The following example illustrates the use of the **parse** method. Provide the function with a date and the function will return the difference between the date provided and 1/1/1970:

```
function GetTimeTest(testdate){
   var s, t;                    //Declare variables.
   var MinMilli = 1000 * 60;       //Initialize variables.
   var HrMilli = MinMilli * 60;
   var DyMilli = HrMilli * 24;
   t = Date.parse(testdate);       //Parse testdate.
   s = "There are "                //Create return string.
   s += Math.round(Math.abs(t / DyMilli)) + " days "
   s += "between " + testdate + " and 1/1/70";
   return(s);                      //Return results.
}
```

Requirements

Version 1

See Also

Methods

Applies To: Date Object

parseFloat Method

Returns a floating-point number converted from a string.

```
function parseFloat(numString : String) : Number
```

Arguments

numString
Required. A string that represents a floating-point number.

Remarks

The **parseFloat** method returns a numerical value equal to the number contained in *numString*. If no prefix of *numString* can be successfully parsed into a floating-point number, **NaN** (not a number) is returned.

You can test for **NaN** using the **isNaN** method.

Example

In following example, the **parseFloat** method is used to convert two strings to numbers.

```
parseFloat("abc")      // Returns NaN.
parseFloat("1.2abc")   // Returns 1.2.
```

Requirements

Version 1

See Also

isNaN Method | parseInt Method | String Object

Applies To: Global Object

parseInt Method

Returns an integer converted from a string.

```
function parseInt(numString : String [, radix : Number]) : Number
```

Arguments

numString
Required. A string to convert into a number.

radix
Optional. A value between 2 and 36 indicating the base of the number contained in *numString*. If not supplied, strings with a prefix of '0x' are considered hexadecimal and strings with a prefix of '0' are considered octal. All other strings are considered decimal.

Remarks

The **parseInt** method returns a whole number value equal to the number contained in *numString*. If no prefix of *numString* can be successfully parsed into an integer, **NaN** (not a number) is returned.

You can test for **NaN** using the **isNaN** method.

Example

In following example, the **parseInt** method is used to convert two strings to numbers.

```
parseInt("abc")     // Returns NaN.
parseInt("12abc")   // Returns 12.
```

Requirements

Version 1

See Also

isNaN Method | parseFloat Method | String Object | valueOf Method

Applies To: Global Object

pop Method

Removes the last element from an array and returns it.

```
function pop() : Object
```

Remarks

If the array is empty, **undefined** is returned.

Requirements

Version 5.5

See Also

push Method

Applies To: Array Object

pow Method

Returns the value of a base expression taken to a specified power.

```
function pow(base : Number, exponent : Number) : Number
```

Arguments

base
Required. The base value of the expression.

exponent
Required. The exponent value of the expression.

Remarks

The pow method returns a numeric expression equal to $\text{base}^{\text{exponent}}$.

Example

The following example illustrates the use of the **pow** method.

```
var x = Math.pow(10,3); // x is assigned the value 1000.
```

Requirements

Version 1

See Also

Methods

Applies To: Math Object

push Method

Appends new elements to an array, and returns the new length of the array.

```
function push([item1 : Object [, ... [, itemN : Object]]]) : Number
```

Arguments

item1, ... , itemN
Optional. New elements of the **Array**.

Remarks

The **push** method appends elements in the order in which they appear. If one of the arguments is an array, it is added as a single element. Use the **concat** method to join the elements from two or more arrays.

Requirements

Version 5.5

See Also

concat Method (Array) | pop Method

Applies To: Array Object

random Method

Returns a pseudorandom number between 0 and 1.

```
function random() : Number
```

Remarks

The pseudorandom number generated is from 0 (inclusive) to 1 (exclusive), that is, the returned number can be zero, but it will always be less than one. The random number generator is seeded automatically when JScript is first loaded.

Requirements

Version 1

See Also

Methods

Applies To: Math Object

replace Method

Returns a copy of a string with text replaced using a regular expression or search string.

```
function replace(rgExp : RegExp, replaceText : String) : String
```

Arguments

rgExp
Required. An instance of a **Regular Expression** object containing the regular expression pattern and applicable flags. Can also be a **String** object or literal. If *rgExp* is not an instance of a **Regular Expression** object, it is converted to a string, and an exact search is made for the results; no attempt is made to convert the string into a regular expression.

replaceText
Required. A **String** object or string literal containing the text to replace for every successful match of *rgExp* in the current string object. In JScript 5.5 or later, the *replaceText* argument can also be a function that returns the replacement text.

Remarks

The result of the **replace** method is a copy of the current string object after the specified replacements have been made.

Any of the following match variables can be used to identify the most recent match and the string from which it came. The match variables can be used in text replacement where the replacement string has to be determined dynamically.

Characters	Meaning
$$	**$** (JScript 5.5 or later)
$&	Specifies that portion of the current string object that the entire pattern matched. (JScript 5.5 or later)
$`	Specifies that portion of the current string object that precedes the match described by **$&**. (JScript 5.5 or later)
$'	Specifies that portion of the current string object that follows the match described by **$&**. (JScript 5.5 or later)
$*n*	The *n*th captured submatch, where *n* is a single decimal digit from 1 through 9. (JScript 5.5 or later)
$*nn*	The *nn*th captured submatch, where *nn* is a two-digit decimal number from 01 through 99. (JScript 5.5 or later)

If *replaceText* is a function, for each matched substring the function is called with the following *m* + 3 arguments where *m* is the number of left capturing parentheses in the *rgExp*. The first argument is the substring that matched. The next *m* arguments are all of the captures that resulted from the search. Argument *m* + 2 is the offset within the current string object where the match occurred, and argument *m* + 3 is the current string object. The result is the string value that results from replacing each matched substring with the corresponding return value of the function call.

The **replace** method updates the properties of the global **RegExp** object.

Example

The following example illustrates the use of the **replace** method to replace the first instance of the word "The" with the word "A." Note that it replaces only the first instance of "The" because the pattern is case-sensitive.

```
function ReplaceDemo(){
   var r, re;                    //Declare variables.
   var ss = "The man hit the ball with the bat.\n";
   ss += "while the fielder caught the ball with the glove.";
   re = /The/g;                 //Create regular expression pattern.
   r = ss.replace(re, "A");    //Replace "The" with "A".
   return(r);                   //Return string with replacement made.
}
```

In addition, the **replace** method can also replace subexpressions in the pattern. The following example swaps each pair of words in the string.

```
function ReplaceDemo(){
   var r, re;                      //Declare variables.
   var ss = "The rain in Spain falls mainly in the plain.";
   re = /(\S+)(\s+)(\S+)/g;        //Create regular expression pattern.
   r = ss.replace(re, "$3$2$1");   //Swap each pair of words.
   return(r);                      //Return resulting string.
}
```

The following example, which works in JScript 5.5 and later, performs a Fahrenheit to Celsius conversion, illustrates using a function as *replaceText*. To see how this function works, pass in a string containing a number followed immediately by an "F" (for example, "Water boils at 212").

```
function f2c(s) {
  var test = /(\d+(\.\d*)?)F\b/g;    //Initialize pattern.
  return(s.replace
    (test,
      function($0,$1,$2) {
        return((($1-32) * 5/9) + "C");
      }
    )
  );
}
document.write(f2c("Water freezes at 32F and boils at 212F."));
```

Requirements

Version 1

See Also

exec Method | match Method | RegExp Object | search Method | test Method

Applies To: String Object

reverse Method

Returns an **Array** object with the elements reversed.

```
function reverse() : Array
```

Remarks

The **reverse** method reverses the elements of an **Array** object in place. It does not create a new **Array** object during execution.

If the array is not contiguous, the **reverse** method creates elements in the array that fill the gaps in the array. Each of these created elements has the value undefined.

Example

The following example illustrates the use of the **reverse** method.

```
function ReverseDemo(){
   var a, l;                  //Declare variables.
   a = new Array(0,1,2,3,4);  //Create an array and populate it.
   l = a.reverse();           //Reverse the contents of the array.
   return(l);                 //Return the resulting array.
}
```

Requirements

Version 2

See Also

Methods

Applies To: Array Object

round Method

Returns a supplied numeric expression rounded to the nearest integer.

```
function round(number : Number) : Number
```

Arguments

number
Required. A numeric expression.

Remarks

If the decimal portion of *number* is 0.5 or greater, the return value is equal to the smallest integer greater than *number*. Otherwise, **round** returns the largest integer less than or equal to *number*.

Requirements

Version 1

See Also

Methods

Applies To: Math Object

Methods (S)

A method is a function that is a member of an object. Following are methods whose names begin with the letter s.

In This Section

search Method
  Returns the position of the first substring match in a regular expression search.
setDate Method
  Sets the numeric date of the **Date** object using local time.
setFullYear Method
  Sets the year value in the Date object using local time.
setHours Method
  Sets the hour value in the **Date** object using local time.
setMilliseconds Method
  Sets the milliseconds value in the **Date** object using local time.
setMinutes Method
  Sets the minutes value in the **Date** object using local time.
setMonth Method
  Sets the month value in the **Date** object using local time.
setSeconds Method
  Sets the seconds value in the **Date** object using local time.
setTime Method
  Sets the date and time value in the **Date** object.
setUTCDate Method
  Sets the numeric date in the **Date** object using Coordinated Universal Time (UTC).
setUTCFullYear Method
  Sets the year value in the **Date** object using Coordinated Universal Time (UTC).
setUTCHours Method
  Sets the hours value in the **Date** object using Coordinated Universal Time (UTC).
setUTCMilliseconds Method
  Sets the milliseconds value in the **Date** object using Coordinated Universal Time (UTC).
setUTCMinutes Method
  Sets the minutes value in the **Date** object using Coordinated Universal Time (UTC).
setUTCMonth Method
  Sets the month value in the **Date** object using Coordinated Universal Time (UTC).
setUTCSeconds Method
  Sets the seconds value in the **Date** object using Coordinated Universal Time (UTC).
setYear Method
  Sets the year value in the **Date** object.
shift Method
  Removes the first element from an array and returns it.
sin Method
  Returns the sine of a number.
slice Method (Array)
  Returns a section of an array.
slice Method (String)
  Returns a section of a string.
small Method
  Places HTML <SMALL> tags around text in a **String** object.
sort Method
  Returns an **Array** object with the elements sorted.
splice Method
  Removes elements from an array and, if necessary, inserts new elements in their place, returning the deleted elements.
split Method
  Returns the array of strings that results when a string is separated into substrings.
sqrt Method
  Returns the square root of a number.
strike Method
  Places HTML <STRIKE> tags around text in a **String** object.
sub Method

Places HTML <SUB> tags around text in a **String** object.

substr Method

Returns a substring beginning at a specified location and having a specified length.

substring Method

Returns the substring at a specified location within a **String** object.

sup Method

Places HTML <SUP> tags around text in a **String** object.

Related Sections

JScript Reference

Lists elements that comprise JScript Language Reference and links to topics that explain the details behind the proper use of language elements.

Methods

Lists the methods, classified alphabetically, available in JScript, and links to each category of methods.

Objects

Explains the concept of objects in JScript, how objects are related to properties and methods, and links to topics that provide more detail about the objects that JScript supports.

search Method

Returns the position of the first substring match in a regular expression search.

```
function search(rgExp : RegExp) : Number
```

Arguments

rgExp
Required. An instance of a **Regular Expression** object containing the regular expression pattern and applicable flags.

Remarks

The **search** method indicates whether a match is found. If a match is found, the **search** method returns an integer value that specifies the offset from the beginning of the string where the match occurred. If no match is found, it returns -1.

Example

The following example illustrates the use of the **search** method.

```
function SearchDemo(){
   var r, re;                   //Declare variables.
   var s = "The rain in Spain falls mainly in the plain.";
   re = /falls/i;               //Create regular expression pattern.
   r = s.search(re);            //Search the string.
   return(r);                   //Return the position of the string.
}
```

Requirements

Version 3

See Also

exec Method | match Method | Regular Expression Object | Regular Expression Syntax | replace Method | test Method

Applies To: String Object

setDate Method

Sets the date of a **Date** object in the local time zone.

```
function setDate(numDate : Number)
```

Arguments

numDate
Required. A numeric value equal to the numeric date.

Remarks

To set the date value using Coordinated Universal Time (UTC), use the **setUTCDate** method.

If the value of *numDate* is greater than the number of days in the month stored in the **Date** object or is a negative number, the date is set to a date equal to *numDate* minus the number of days in the stored month. For example, if the stored date is January 5, 1996, and **setDate(32)** is called, the date changes to February 1, 1996. Negative numbers behave similarly.

Example

The following example illustrates the use of the **setDate** method.

```
function SetDateDemo(newdate){
   var d, s;                  //Declare variables.
   d = new Date();            //Create date object.
   d.setDate(newdate);        //Set date to newdate.
   s = "Current setting is ";
   s += d.toLocaleString();
   return(s);                 //Return newly set date.
}
```

Requirements

Version 3

See Also

getDate Method | getUTCDate Method | setUTCDate Method

Applies To: Date Object

setFullYear Method

Sets the year value in the **Date** object using local time.

```
function setFullYear(numYear : Number [, numMonth Number [, numDate Number]])
```

Arguments

numYear
Required. A numeric value equal to the year.
numMonth
Optional. A numeric value equal to the month. Must be supplied if *numDate* is supplied.
numDate
Optional. A numeric value equal to the date.

Remarks

All **set** methods taking optional arguments use the value returned from corresponding **get** methods, if you do not specify the optional argument. For example, if the *numMonth* argument is optional, but not specified, JScript uses the value returned from the **getMonth** method.

In addition, if the value of an argument is greater than its range or is a negative number, other stored values are modified accordingly.

To set the year using Coordinated Universal Time (UTC), use the **setUTCFullYear** method.

The range of years supported in the date object is approximately 285,616 years from either side of 1970.

Example

The following example illustrates the use of the **setFullYear** method:

```
function SetFullYearDemo(newyear){
   var d, s;                  //Declare variables.
   d = new Date();            //Create Date object.
   d.setFullYear(newyear);    //Set year.
   s = "Current setting is ";
   s += d.toLocaleString();
   return(s);                 //Return new date setting.
}
```

Requirements

Version 3

See Also

getFullYear Method | getUTCFullYear Method | setUTCFullYear Method

Applies To: Date Object

setHours Method

Sets the hour value in the **Date** object using local time.

```
function setHours(numHours : Number [, numMin : Number [, numSec : Number [, numMilli : Number ]]])
```

Arguments

numHours
  Required. A numeric value equal to the hour's value.
numMin
  Optional. A numeric value equal to the minute's value.
numSec
  Optional. A numeric value equal to the second's value.
numMilli
  Optional. A numeric value equal to the milliseconds value.

Remarks

All **set** methods taking optional arguments use the value returned from corresponding **get** methods, if you do not specify an optional argument. For example, if the *numMinutes* argument is optional, but not specified, JScript uses the value returned from the **getMinutes** method.

To set the hours value using Coordinated Universal Time (UTC), use the **setUTCHours** method.

If the value of an argument is greater than its range or is a negative number, other stored values are modified accordingly. For example, if the stored date is "Jan 5, 1996 00:00:00", and **setHours(30)** is called, the date is changed to "Jan 6, 1996 06:00:00." Negative numbers have a similar behavior.

Example

The following example illustrates the use of the **setHours** method.

```
function SetHoursDemo(nhr, nmin, nsec){
   var d, s;                      //Declare variables.
   d = new Date();                //Create Date object.
   d.setHours(nhr, nmin, nsec);  //Set hours, minutes, & seconds.
   s = "Current setting is " + d.toLocaleString()
   return(s);                     //Return new date setting.
}
```

Requirements

Version 3

See Also

getHours Method | getUTCHours Method | setUTCHours Method

Applies To: Date Object

setMilliseconds Method

Sets the milliseconds value in the **Date** object using local time.

```
function setMilliseconds(numMilli : Number)
```

Arguments

numMilli
Required. A numeric value equal to the millisecond value.

Remarks

To set the milliseconds value using Coordinated Universal Time (UTC), use the **setUTCMilliseconds** method.

If the value of *numMilli* is greater than 999 or is a negative number, the stored number of seconds (and minutes, hours, and so forth if necessary) is incremented an appropriate amount.

Example

The following example illustrates the use of the **setMilliseconds** method.

```
function SetMSecDemo(nmsec){
   var d, s;                     //Declare variables.
   var sep = ":";                //Initialize separator.
   d = new Date();               //Create Date object.
   d.setMilliseconds(nmsec);     //Set milliseconds.
   s = "Current setting is ";
   s += d.toLocaleString() + sep + d.getMilliseconds();
   return(s);                    //Return new date setting.
}
```

Requirements

Version 3

See Also

getMilliseconds Method | getUTCMilliseconds Method | setUTCMilliseconds Method

Applies To: Date Object

setMinutes Method

Sets the minutes value in the **Date** object using local time.

```
function setMinutes(numMinutes : Number [, numSeconds : Number [, numMilli : Number]])
```

Arguments

numMinutes
Required. A numeric value equal to the minute's value.
numSeconds
Optional. A numeric value equal to the seconds value. Must be supplied if the numMilli argument is used.
numMilli
Optional. A numeric value equal to the milliseconds value.

Remarks

All **set** methods taking optional arguments use the value returned from corresponding **get** methods, if you do not specify an optional argument. For example, if the *numSeconds* argument is optional, but not specified, JScript uses the value returned from the **getSeconds** method.

To set the minutes value using Coordinated Universal Time (UTC), use the **setUTCMinutes** method.

If the value of an argument is greater than its range or is a negative number, other stored values are modified accordingly. For example, if the stored date is "Jan 5, 1996 00:00:00" and **setMinutes(90)** is called, the date is changed to "Jan 5, 1996 01:30:00." Negative numbers have a similar behavior.

Example

The following example illustrates the use of the **setMinutes** method.

```
function SetMinutesDemo(nmin, nsec){
   var d, s;                     //Declare variables.
   d = new Date();               //Create Date object.
   d.setMinutes(nmin, nsec);     //Set minutes.
   s = "Current setting is " + d.toLocaleString()
   return(s);                    //Return new setting.
}
```

Requirements

Version 1

See Also

getMinutes Method | getUTCMinutes Method | setUTCMinutes Method

Applies To: Date Object

setMonth Method

Sets the month value in the **Date** object using local time.

```
function setMonth(numMonth : Number [, dateVal : Number])
```

Arguments

numMonth
Required. A numeric value equal to the month.
dateVal
Optional. A numeric value representing the date. If not supplied, the value from a call to the **getDate** method is used.

Remarks

To set the month value using Coordinated Universal Time (UTC), use the **setUTCMonth** method.

If the value of *numMonth* is greater than 11 (January is month 0) or is a negative number, the stored year is modified accordingly. For example, if the stored date is "Jan 5, 1996" and **setMonth(14)** is called, the date is changed to "Mar 5, 1997."

Example

The following example illustrates the use of the **setMonth** method.

```
function SetMonthDemo(newmonth){
   var d, s;               //Declare variables.
   d = new Date();         //Create Date object.
   d.setMonth(newmonth);   //Set month.
   s = "Current setting is ";
   s += d.toLocaleString();
   return(s);              //Return new setting.
}
```

Requirements

Version 1

See Also

getMonth Method | getUTCMonth Method | setUTCMonth Method

Applies To: Date Object

setSeconds Method

Sets the seconds value in the **Date** object using local time.

```
function setSeconds(numSeconds : Number [, numMilli : Number])
```

Arguments

numSeconds
Required. A numeric value equal to the seconds value.
numMilli
Optional. A numeric value equal to the milliseconds value.

Remarks

All **set** methods taking optional arguments use the value returned from corresponding **get** methods, if you do not specify an optional argument. For example, if the *numMilli* argument is optional, but not specified, JScript uses the value returned from the **getMilliseconds** method.

To set the seconds value using Coordinated Universal Time (UTC), use the **setUTCSeconds** method.

If the value of an argument is greater than its range or is a negative number, other stored values are modified accordingly. For example, if the stored date is "Jan 5, 1996 00:00:00" and **setSeconds(150)** is called, the date is changed to "Jan 5, 1996 00:02:30."

Example

The following example illustrates the use of the **setSeconds** method.

```
function SetSecondsDemo(nsec, nmsec){
   var d, s;                    //Declare variables.
   var sep = ":";
   d = new Date();              //Create Date object.
   d.setSeconds(nsec, nmsec);   //Set seconds and milliseconds.
   s = "Current setting is ";
   s += d.toLocaleString() + sep + d.getMilliseconds();
   return(s);                   //Return new setting.
}
```

Requirements

Version 1

See Also

getSeconds Method | getUTCSeconds Method | setUTCSeconds Method

Applies To: Date Object

setTime Method

Sets the date and time value in the **Date** object.

```
function setTime(milliseconds : Number)
```

Arguments

milliseconds
Required. A numeric value representing the number of elapsed milliseconds since midnight, January 1, 1970 UTC.

Remarks

If *milliseconds* is negative, it indicates a date before 1970. The range of available dates is approximately 285,616 years from either side of 1970.

Setting the date and time with the **setTime** method is independent of the time zone.

Example

The following example illustrates the use of the **setTime** method.

```
function SetTimeTest(newtime){
   var d, s;                  //Declare variables.
   d = new Date();            //Create Date object.
   d.setTime(newtime);        //Set time.
   s = "Current setting is ";
   s += d.toUTCString();
   return(s);                 //Return new setting.
}
```

Requirements

Version 1

See Also

getTime Method

Applies To: Date Object

setUTCDate Method

Sets the numeric date in the **Date** object using Coordinated Universal Time (UTC).

```
function setUTCDate(numDate : Number)
```

Arguments

numDate
Required. A numeric value equal to the numeric date.

Remarks

To set the date using local time, use the **setDate** method.

If the value of *numDate* is greater than the number of days in the month stored in the **Date** object or is a negative number, the date is set to a date equal to *numDate* minus the number of days in the stored month. For example, if the stored date is January 5, 1996, and **setUTCDate(32)** is called, the date changes to February 1, 1996. Negative numbers have a similar behavior.

Example

The following example illustrates the use of the **setUTCDate** method.

```
function SetUTCDateDemo(newdate){
   var d, s;                   //Declare variables.
   d = new Date();             //Create Date object.
   d.setUTCDate(newdate);      //Set UTC date.
   s = "Current setting is ";
   s += d.toUTCString();
   return(s);                  //Return new setting.
}
```

Requirements

Version 3

See Also

getDate Method | getUTCDate Method | setDate Method

Applies To: Date Object

setUTCFullYear Method

Sets the year value in the **Date** object using Coordinated Universal Time (UTC).

```
function setUTCFullYear(numYear : Number [, numMonth : Number [, numDate : Number]])
```

Arguments

numYear
Required. A numeric value equal to the year.
numMonth
Optional. A numeric value equal to the month.
numDate
Optional. A numeric value equal to the date.

Remarks

All **set** methods taking optional arguments use the value returned from corresponding **get** methods, if you do not specify an optional argument. For example, if the *numMonth* argument is optional, but not specified, JScript uses the value returned from the **getUTCMonth** method.

In addition, if the value of an argument is greater that its range or is a negative number, other stored values are modified accordingly.

To set the year using local time, use the **setFullYear** method.

The range of years supported in the **Date** object is approximately 285,616 years from either side of 1970.

Example

The following example illustrates the use of the **setUTCFullYear** method.

```
function SetUTCFullYearDemo(newyear){
   var d, s;                    //Declare variables.
   d = new Date();              //Create Date object.
   d.setUTCFullYear(newyear);   //Set UTC full year.
   s = "Current setting is ";
   s += d.toUTCString();
   return(s);                   //Return new setting.
}
```

Requirements

Version 3

See Also

getFullYear Method | getUTCFullYear Method | setFullYear Method

Applies To: Date Object

setUTCHours Method

Sets the hours value in the **Date** object using Coordinated Universal Time (UTC).

```
function setUTCHours(numHours : Number [, numMin : Number [, numSec : Number [, numMilli : Nu
mber]]])
```

Arguments

numHours
  Required. A numeric value equal to the hours value.
numMin
  Optional. A numeric value equal to the minutes value.
numSec
  Optional. A numeric value equal to the seconds value.
numMilli
  Optional. A numeric value equal to the milliseconds value.

Remarks

All **set** methods taking optional arguments use the value returned from corresponding **get** methods, if you do not specify an optional argument. For example, if the *numMin* argument is optional, but not specified, JScript uses the value returned from the **getUTCMinutes** method.

To set the hours value using local time, use the **setHours** method.

If the value of an argument is greater than its range, or is a negative number, other stored values are modified accordingly. For example, if the stored date is "Jan 5, 1996 00:00:00.00", and **setUTCHours(30)** is called, the date is changed to "Jan 6, 1996 06:00:00.00."

Example

The following example illustrates the use of the **setUTCHours** method.

```
function SetUTCHoursDemo(nhr, nmin, nsec){
   var d, s;                        //Declare variables.
   d = new Date();                  //Create Date object.
   d.setUTCHours(nhr, nmin, nsec);  //Set UTC hours, minutes, seconds.
   s = "Current setting is " + d.toUTCString()
   return(s);                       //Return new setting.
}
```

Requirements

Version 3

See Also

getHours Method | getUTCHours Method | setHours Method

Applies To: Date Object

setUTCMilliseconds Method

Sets the milliseconds value in the **Date** object using Coordinated Universal Time (UTC).

```
function setUTCMilliseconds(numMilli : Number)
```

Arguments

numMilli
Required. A numeric value equal to the millisecond value.

Remarks

To set the milliseconds using local time, use the **setMilliseconds** method.

If the value of *numMilli* is greater than 999, or is a negative number, the stored number of seconds (and minutes, hours, and so forth, if necessary) is incremented an appropriate amount.

Example

The following example illustrates the use of the **setUTCMilliseconds** method.

```
function SetUTCMSecDemo(nmsec){
   var d, s;                          //Declare variables.
   var sep = ":";                     //Initialize separator.
   d = new Date();                    //Create Date object.
   d.setUTCMilliseconds(nmsec);       //Set UTC milliseconds.
   s = "Current setting is ";
   s += d.toUTCString() + sep + d.getUTCMilliseconds();
   return(s);                         //Return new setting.
}
```

Requirements

Version 3

See Also

getMilliseconds Method | getUTCMilliseconds Method | setMilliseconds Method

Applies To: Date Object

setUTCMinutes Method

Sets the minutes value in the **Date** object using Coordinated Universal Time (UTC).

```
function setUTCMinutes(numMinutes : Number [, numSeconds : Number [, numMilli : Number ]])
```

Arguments

numMinutes
Required. A numeric value equal to the minutes value.
numSeconds
Optional. A numeric value equal to the seconds value. Must be supplied if *numMilli* is used.
numMilli
Optional. A numeric value equal to the milliseconds value.

Remarks

All **set** methods taking optional arguments use the value returned from corresponding **get** methods, if you do not specify an optional argument. For example, if the *numSeconds* argument is optional, but not specified, JScript uses the value returned from the **getUTCSeconds** method.

To modify the minutes value using local time, use the **setMinutes** method.

If the value of an argument is greater than its range, or is a negative number, other stored values are modified accordingly. For example, if the stored date is "Jan 5, 1996 00:00:00.00", and **setUTCMinutes(70)** is called, the date is changed to "Jan 5, 1996 01:10:00.00."

Example

The following example illustrates the use of the **setUTCMinutes** method:

```
function SetUTCMinutesDemo(nmin, nsec){
   var d, s;                    //Declare variables.
   d = new Date();              //Create Date object.
   d.setUTCMinutes(nmin,nsec);  //Set UTC minutes.
   s = "Current setting is " + d.toUTCString()
   return(s);                   //Return new setting.
}
```

Requirements

Version 3

See Also

getMinutes Method | getUTCMinutes Method | setMinutes Method

Applies To: Date Object

setUTCMonth Method

Sets the month value in the **Date** object using Coordinated Universal Time (UTC).

```
function setUTCMonth(numMonth : Number [, dateVal : Number ])
```

Arguments

numMonth
Required. A numeric value equal to the month.

dateVal
Optional. A numeric value representing the date. If not supplied, the value from a call to the **getUTCDate** method is used.

Remarks

To set the month value using local time, use the **setMonth** method.

If the value of *numMonth* is greater than 11 (January is month 0), or is a negative number, the stored year is incremented or decremented appropriately. For example, if the stored date is "Jan 5, 1996 00:00:00.00", and **setUTCMonth(14)** is called, the date is changed to "Mar 5, 1997 00:00:00.00."

Example

The following example illustrates the use of the **setUTCMonth** method.

```
function SetUTCMonthDemo(newmonth){
   var d, s;                       //Declare variables.
   d = new Date();                 //Create Date object.
   d.setUTCMonth(newmonth);        //Set UTC month.
   s = "Current setting is ";
   s += d.toUTCString();
   return(s);                      //Return new setting.
}
```

Requirements

Version 3

See Also

getMonth Method | getUTCMonth Method | setMonth Method

Applies To: Date Object

setUTCSeconds Method

Sets the seconds value in the **Date** object using Coordinated Universal Time (UTC).

```
function setUTCSeconds(numSeconds : Number [, numMilli : Number ])
```

Arguments

numSeconds
  Required. A numeric value equal to the seconds value.
numMilli
  Optional. A numeric value equal to the milliseconds value.

Remarks

All **set** methods taking optional arguments use the value returned from corresponding **get** methods, if you do not specify an optional argument. For example, if the *numMilli* argument is optional, but not specified, JScript uses the value returned from the **getUTCMilliseconds** method.

To set the seconds value using local time, use the **setSeconds** method.

If the value of an argument is greater than its range or is a negative number, other stored values are modified accordingly. For example, if the stored date is "Jan 5, 1996 00:00:00.00" and **setSeconds(150)** is called, the date is changed to "Jan 5, 1996 00:02:30.00."

Example

The following example illustrates the use of the **setSeconds** method.

```
function SetUTCSecondsDemo(nsec, nmsec){
   var d, s;                       //Declare variables.
   d = new Date();                 //Create Date object.
   d.setUTCSeconds(nsec, nmsec);   //Set UTC seconds and milliseconds.
   s = "Current UTC milliseconds setting is ";
   s += d.getUTCMilliseconds();    //Get new setting.
   return(s);                      //Return new setting.
}
```

Requirements

Version 3

See Also

getSeconds Method | getUTCSeconds Method | setSeconds Method

Applies To: Date Object

setYear Method

Sets the year value in the **Date** object.

```
function setYear(numYear : Number)
```

Arguments

numYear
Required. A numeric value equal to the year minus 1900.

Remarks

This method is obsolete, and is maintained for backwards compatibility only. Use the **setFullYear** method instead.

To set the year of a **Date** object to 1997, call **setYear(97)**. To set the year to 2010, call **setYear(2010)**. Finally, to set the year to a year in the range 0-99, use the **setFullYear** method.

> **Note** For JScript version 1.0, **setYear** uses a value that is the result of the addition of 1900 to the year value provided by *numYear*, regardless of the value of the year. For example, to set the year to 1899 *numYear* is -1 and to set the year 2000 *numYear* is 100.

Requirements

Version 1

See Also

getFullYear Method | getUTCFullYear Method | getYear Method | setFullYear Method | setUTCFullYear Method

Applies To: Date Object

shift Method

Removes the first element from an array and returns that element.

```
function shift() : Object
```

Remarks

The **shift** method removes the first element from an array and returns it.

Requirements

Version 5.5

See Also

unshift Method

Applies To: Array Object

sin Method

Returns the sine of a number.

```
function sin(number : Number) : Number
```

Arguments

number
Required. A numeric expression for which the sine is calculated.

Remarks

The return value is the sine of the numeric argument.

Requirements

Version 1

See Also

acos Method | asin Method | atan Method | cos Method | tan Method

Applies To: Math Object

slice Method (Array)

Returns a section of an array.

```
function slice(start : Number [, end : Number]) : Array
```

Arguments

start
Required. The index to the beginning of the specified portion of the array.
end
Optional. The index to the end of the specified portion of the array.

Remarks

The **slice** method returns an **Array** object containing the specified portion of the array.

The **slice** method copies up to, but not including, the element indicated by *end*. If *start* is negative, it is treated as *length* + *start* where *length* is the length of the array. If *end* is negative, it is treated as *length* + *end* where *length* is the length of the array. If *end* is omitted, extraction continues to the end of the array. If *end* occurs before *start*, no elements are copied to the new array.

Example

In the following example, all but the last element of *myArray* is copied into *newArray*:

```
newArray = myArray.slice(0, -1)
```

Requirements

Version 3

See Also

slice Method (String) | String Object

Applies To: Array Object

slice Method (String)

Returns a section of a string.

```
function slice(start : Number [, end : Number]) : String
```

Arguments

start
Required. The index to the beginning of the specified portion of the string.
end
Optional. The index to the end of the specified portion of the string.

Remarks

The **slice** method returns a **String** object containing the specified portion of the string.

The **slice** method copies up to, but not including, the element indicated by *end*. If *start* is negative, it is treated as *length* + *start* where *length* is the length of the string. If *end* is negative, it is treated as *length* + *end* where *length* is the length of the string. If *end* is omitted, extraction continues to the end of the string. If *end* occurs before *start*, no characters are copied to the new string.

Example

In the following example, the first call to the **slice** method returns a string that contains the first five characters of `str`. The second call to the **slice** method returns a string that contains the last five characters of `str`.

```
var str = "hello world";
var firstfive = str.slice(0,5); // Contains "hello".
var lastfive = str.slice(-5);   // Contains "world".
```

Requirements

Version 3

See Also

Array Object | slice Method (Array)

Applies To: String Object

small Method

Returns a string with HTML <SMALL> tags around the text in a **String** object.

```
function small() : String
```

Remarks

No checking is done to see if the tag has already been applied to the string.

Example

The following example illustrates the use of the **small** method:

```
var strVariable = "This is a string";
strVariable = strVariable.small();
```

The value of `strVariable` after the last statement is:

```
<SMALL>This is a string</SMALL>
```

Requirements

Version 1

See Also

big Method

Applies To: String Object

sort Method

Returns an **Array** object with the elements sorted.

```
function sort(sortFunction : Function ) : Array
```

Arguments

sortFunction
Optional. Name of the function used to determine the order of the elements.

Remarks

The **sort** method sorts the **Array** object in place; no new **Array** object is created during execution.

If you supply a function in the *sortFunction* argument, it must return one of the following values:

- A negative value if the first argument passed is less than the second argument.
- Zero if the two arguments are equivalent.
- A positive value if the first argument is greater than the second argument.

If the *sortFunction* argument is omitted, the elements are sorted in ascending, ASCII character order.

Example

The following example illustrates the use of the **sort** method.

```
function SortDemo(){
   var a, l;                       //Declare variables.
   a = new Array("X" ,"y" ,"d", "Z", "v","m","r");
   l = a.sort();                   //Sort the array.
   return(l);                      //Return sorted array.
}
```

Requirements

Version 2

See Also

Objects

Applies To: Array Object

splice Method

Removes elements from an array and, if necessary, inserts new elements in their place, returning the deleted elements. Returns the elements removed from the array.

```
function splice(start : Number, deleteCount : Number [, item1 : Object [, ... [, itemN : Obje
ct]]]]) : Array
```

Arguments

start
Required. The zero-based location in the array from which to start removing elements.
deleteCount
Required. The number of elements to remove.
item1, ... , itemN
Optional. Elements to insert into the array in place of the deleted elements.

Remarks

The **splice** method modifies the array by removing the specified number of elements from position *start* and inserting new elements. The deleted elements are returned as a new **array** object.

Requirements

Version 5.5

See Also

slice Method (Array)

Applies To: Array Object

split Method

Returns the array of strings that results when a string is separated into substrings.

```
function split([ separator : { String | RegExp } [, limit : Number]]) : Array
```

Arguments

separator
Optional. A string or an instance of a **Regular Expression** object identifying one or more characters to use in separating the string. If omitted, a single-element array containing the entire string is returned.

limit
Optional. A value used to limit the number of elements returned in the array.

Remarks

The result of the **split** method is an array of strings split at each point where *separator* occurs in the string. The *separator* is not returned as part of any array element.

Example

The following example illustrates the use of the **split** method.

```
function SplitDemo(){
   var s, ss;
   var s = "The rain in Spain falls mainly in the plain.";
   // Split at each space character.
   ss = s.split(" ");
   return(ss);
}
```

Requirements

Version 3

See Also

concat Method (String) | RegExp Object | Regular Expression Object | Regular Expression Syntax

Applies To: String Object

sqrt Method

Returns the square root of a number.

```
function sqrt(number : Number) : Number
```

Arguments

number

Required. A numeric expression for which the square root is calculated.

Remarks

If *number* is negative, the return value is **NaN**.

Requirements

Version 1

See Also

SQRT1_2 Property | SQRT2 Property

Applies To: Math Object

strike Method

Returns a string with HTML <STRIKE> tags placed around text in a **String** object.

```
function strike() : String
```

Remarks

No checking is done to see if the tag has already been applied to the string.

Example

The following example demonstrates how the **strike** method works:

```
var strVariable = "This is a string object";
strVariable = strVariable.strike();
```

The value of `strVariable` after the last statement is:

```
<STRIKE>This is a string object</STRIKE>
```

Requirements

Version 1

See Also

Methods

Applies To: String Object

sub Method

Returns as string with HTML <SUB> tags placed around text in a **String** object.

```
function sub() : String
```

Remarks

No checking is done to see if the tag has already been applied to the string.

Example

The following example demonstrates how the **sub** method works:

```
var strVariable = "This is a string object";
strVariable = strVariable.sub();
```

The value of `strVariable` after the last statement is:

```
<SUB>This is a string object</SUB>
```

Requirements

Version 1

See Also

sup Method

Applies To: String Object

substr Method

Returns a substring beginning at a specified location and having a specified length.

```
function substr(start : Number [, length : Number]) : String
```

Arguments

start
Required. The starting position of the desired substring. The index of the first character in the string is zero.
length
Optional. The number of characters to include in the returned substring.

Remarks

If *length* is zero or negative, an empty string is returned. If not specified, the substring continues to the end of the string.

Example

The following example illustrates the use of the **substr** method.

```
function SubstrDemo(){
   var s, ss;                //Declare variables.
   var s = "The rain in Spain falls mainly in the plain.";
   ss = s.substr(12, 5);  //Get substring.
   return(ss);               // Returns "Spain".
}
```

Requirements

Version 3

See Also

substring Method

Applies To: String Object

substring Method

Returns the substring at the specified location within a **String** object.

```
function substring(start : Number, end : Number) : String
```

Arguments

start
  Required. The zero-based index integer indicating the beginning of the substring.
end
  Required. The zero-based index integer indicating the end of the substring.

Remarks

The **substring** method returns a string containing the substring from *start* up to, but not including, *end*.

The **substring** method uses the lower value of *start* and *end* as the beginning point of the substring. For example, *strvar*.**substring(**0, 3**)** and *strvar*.**substring(**3, 0**)** return the same substring.

If either *start* or *end* is **NaN** or negative, it is replaced with zero.

The length of the substring is equal to the absolute value of the difference between *start* and *end*. For example, the length of the substring returned in *strvar*.**substring(**0, 3**)** and *strvar*.**substring(**3, 0**)** is three.

Example

The following example illustrates the use of the **substring** method.

```
function SubstringDemo(){
   var ss;                          //Declare variables.
   var s = "The rain in Spain falls mainly in the plain..";
   ss = s.substring(12, 17);    //Get substring.
   return(ss);                      //Return substring.
}
```

Requirements

Version 1

See Also

substr Method

Applies To: String Object

sup Method

Returns as string with HTML <SUP> tags placed around text in a **String** object.

```
function sup() : String
```

Remarks

No checking is done to see if the tag has already been applied to the string.

Example

The following example demonstrates how the **sup** method works.

```
var strVariable = "This is a string object";
strVariable = strVariable.sup();
```

The value of `strVariable` after the last statement is:

```
<SUP>This is a string object</SUP>
```

Requirements

Version 1

See Also

sub Method

Applies To: String Object

Methods (T-Z)

A method is a function that is a member of an object. Following are methods whose names begin with letters t through z.

In This Section

tan Method
  Returns the tangent of a number.
test Method
  Returns a Boolean value that indicates whether or not a pattern exists in a searched string.
toArray Method
  Returns a standard JScript array converted from a VBArray.
toDateString Method
  Returns a date as a string value.
toExponential Method
  Returns a string containing a number represented in exponential notation.
toFixed Method
  Returns a string representing a number in fixed-point notation.
toGMTString Method
  Returns a date converted to a string using Greenwich Mean Time (GMT).
toLocaleDateString Method
  Returns a date as a string value appropriate to the host environment's current locale.
toLocaleLowerCase Method
  Returns a string where all alphabetic characters have been converted to lowercase, taking into account the host environment's current locale.
toLocaleString Method
  Returns a date converted to a string using the current locale.
toLocaleTimeString Method
  Returns a time as a string value appropriate to the host environment's current locale.
toLocaleUpperCase Method
  Returns a string where all alphabetic characters have been converted to uppercase, taking into account the host environment's current locale.
toLowerCase Method
  Returns a string where all alphabetic characters have been converted to lowercase.
toPrecision Method
  Returns a string containing a number represented either in exponential or fixed-point notation with a specified number of digits.
toString Method
  Returns a string representation of an object.
toTimeString Method
  Returns a time as a string value.
toUpperCase Method
  Returns a string where all alphabetic characters have been converted to uppercase.
toUTCString Method
  Returns a date converted to a string using Coordinated Universal Time (UTC).
ubound Method
  Returns the highest index value used in the specified dimension of the VBArray.
unescape Method
  Decodes **String** objects encoded with the **escape** method.
unshift Method
  Returns an array with specified elements inserted at the beginning.
UTC Method
  Returns the number of milliseconds between midnight, January 1, 1970 Coordinated Universal Time (UTC) (or GMT) and the supplied date.
valueOf Method
  Returns the primitive value of the specified object.

Related Sections

JScript Reference
  Lists elements that comprise JScript Language Reference and links to topics that explain the details behind the proper use of

language elements.

Methods

Lists the methods, classified alphabetically, available in JScript, and links to each category of methods.

Objects

Explains the concept of objects in JScript, how objects are related to properties and methods, and links to topics that provide more detail about the objects that JScript supports.

tan Method

Returns the tangent of a number.

```
function tan(number : Number) : Number
```

Arguments

number
Required. A numeric expression for which the tangent is calculated.

Remarks

The return value is the tangent of *number*.

Requirements

Version 1

See Also

acos Method | asin Method | atan Method | atan2 Method | cos Method | sin Method

Applies To: Math Object

test Method

Returns a Boolean value that indicates whether a regular expression pattern exists in a searched string.

```
function test(str : String) : Boolean
```

Arguments

str
Required. The string on which to perform the search.

Remarks

The **test** method checks to see if a pattern exists within a string and returns **true** if so, and **false** otherwise. If a match is found, the properties of the global **RegExp** object are updated to reflect the results of the match.

If the global flag is set for a regular expression, **test** searches the string beginning at the position indicated by the value of **lastIndex**. If the global flag is not set, **test** ignores the value of **lastIndex** and searches from the beginning of the string.

Example

The following example illustrates the use of the **test** method. To use this example, pass the function a regular expression pattern and a string. The function will test for the occurrence of the regular expression pattern in the string and return a string indicating the results of that search:

```
function TestDemo(re, s){
   var s1;                         //Declare variable.
   // Test string for existence of regular expression.
   if (re.test(s))                 //Test for existence.
      s1 = " contains ";           //s contains pattern.
   else
      s1 = " does not contain ";   //s does not contain pattern.
   return("'" + s + "'" + s1 + "'"+ re.source + "'"); //Return string.
}
```

Requirements

Version 3

See Also

RegExp Object | Regular Expression Syntax

Applies To: Regular Expression Object

toArray Method

Returns a standard JScript array converted from a VBArray.

```
function toArray() : Array
```

Remarks

The conversion translates the multidimensional VBArray into a single dimensional JScript array. The **toArray** method appends each successive dimension to the end of the previous one. For example, a VBArray with three dimensions and three elements in each dimension converts to a JScript array as follows:

Suppose the VBArray contains: (1, 2, 3), (4, 5, 6), (7, 8, 9). After translation, the JScript array contains: 1, 2, 3, 4, 5, 6, 7, 8, 9.

There is currently no way to convert a JScript array into a VBArray.

Example

The following example consists of three parts. The first part is VBScript code that creates a Visual Basic safe array. The second part is JScript code that converts the Visual Basic safe array to a JScript array. Both of the first and the second parts go into the <HEAD> section of an HTML page. The third part is the JScript code that goes into the <BODY> section to run the other two parts.

```
<HEAD>
<SCRIPT LANGUAGE="VBScript">
<!--
Function CreateVBArray()
   Dim i, j, k
   Dim a(2, 2)
   k = 1
   For i = 0 To 2
      For j = 0 To 2
         a(j, i) = k
         document.writeln(k)
         k = k + 1
      Next
      document.writeln("<BR>")
   Next
   CreateVBArray = a
End Function
-->
</SCRIPT>

<SCRIPT LANGUAGE="JScript">
<!--
function VBArrayTest(vbarray)
{
   var a = new VBArray(vbarray);
   var b = a.toArray();
   var i;
   for (i = 0; i < 9; i++)
   {
      document.writeln(b[i]);
   }
}
-->
</SCRIPT>
</HEAD>

<BODY>
<SCRIPT LANGUAGE="JScript">
<!--
   VBArrayTest(CreateVBArray());
-->
</SCRIPT>
```

```
</BODY>
```

Requirements

Version 3

See Also

dimensions Method | getItem Method | lbound Method | ubound Method

Applies To: VBArray Object

toDateString Method

Returns a date as a string value.

```
function toDateString() : String
```

Remarks

The **toDateString** method returns a string value containing the date, in the current time zone, in a convenient, easily read format.

Requirements

Version 5.5

See Also

toTimeString Method | toLocaleDateString Method

Applies To: Date Object

toExponential Method

Returns a string containing a number represented in exponential notation.

```
function toExponential( [fractionDigits : Number] ) : String
```

Arguments

fractionDigits
Optional. Number of digits after the decimal point. Must be in the range 0 – 20, inclusive.

Remarks

The **toExponential** method returns a string representation of a number in exponential notation. The string contains one digit before the significand's decimal point, and may contain *fractionDigits* digits after it.

If *fractionDigits* is not supplied, the **toExponential** method returns as many digits necessary to uniquely specify the number.

Requirements

Version 5.5

See Also

toFixed Method | toPrecision Method

Applies To: Number Object

toFixed Method

Returns a string representing a number in fixed-point notation.

```
function toFixed( [fractionDigits : Number] ) : String
```

Arguments

fractionDigits
Optional. Number of digits after the decimal point. Must be in the range 0 – 20, inclusive.

Remarks

The **toFixed** method returns a string representation of a number in fixed-point notation. The string contains one digit before the significand's decimal point, and must contain *fractionDigits* digits after it.

If *fractionDigits* is not supplied or **undefined**, the **toFixed** method assumes the value is zero.

Requirements

Version 5.5

See Also

toExponential Method | toPrecision Method

Applies To: Number Object

toGMTString Method

Returns a date converted to a string using Greenwich Mean Time (GMT).

```
function toGMTString() : String
```

Remarks

The **toGMTString** method is obsolete, and is provided for backwards compatibility only. It is recommended that you use the **toUTCString** method instead.

The **toGMTString** method returns a **String** object that contains the date formatted using GMT convention. The format of the return value is as follows: "05 Jan 1996 00:00:00 GMT."

Requirements

Version 1

See Also

toUTCString Method

Applies To: Date Object

toLocaleDateString Method

Returns a date as a string value appropriate to the host environment's current locale.

```
function toLocaleDateString() : String
```

Remarks

The **toLocaleDateString** method returns a string value that contains a date, in the current time zone, in an easily read format. The date is in the default format of the host environment's current locale. The return value of this method cannot be relied upon in scripting, as it will vary from computer to computer. The **toLocalDateString** method should only be used to format display – never as part of a computation.

Requirements

Version 5.5

See Also

toDateString Method | toLocaleTimeString Method

Applies To: Date Object

toLocaleLowerCase Method

Returns a string where all alphabetic characters have been converted to lowercase, taking into account the host environment's current locale.

```
function tolocaleLowerCase() : String
```

Remarks

The **toLocaleLowerCase** method converts the characters in a string, taking into account the host environment's current locale. In most cases, the results are the same as you would obtain with the **toLowerCase** method. Results differ if the rules for a language conflict with the regular Unicode case mappings.

Requirements

Version 5.5

See Also

toLocaleUpperCase Method | toLowerCase Method

Applies To: String Object

toLocaleString Method

Returns a value as a string value appropriate to the host environment's current locale.

```
function toLocaleString() : String
```

Remarks

For the **Array** object, the elements of the array are converted to strings and these strings are concatenated and returned, each separated by the list separator specified for the host environment's current locale.

For the **Date** object, the **toLocaleString** method returns a **String** object that contains the date written in the current locale's long default format.

- For dates between 1601 and 9999 A.D., the date is formatted according to the user's Control Panel Regional Settings.
- For dates outside this range, the default format of the **toString** method is used.

For the **Number** object, **toLocaleString** produces a string value that represents the value of the **Number** formatted as appropriate for the host environment's current locale.

For **Object** objects, **ToLocaleString** is provided to give all objects a generic **toLocaleString** capability, even though they may not use it.

> **Note** **toLocaleString** should only be used to display results to a user; it should never be used as the basis for computation within a script as the returned result is machine-specific.

Example

The following client-side example illustrates the use of the **toLocaleString** method using an **Array**, a **Date**, and a **Number** object.

```
function toLocaleStringArray() {
   // Declare variables.
   var myArray = new Array(6);
   var i;
   // Initialize string.
   var s = "The array contains: ";
   // Populate array with values.
   for(i = 0;i < 7; i++)
   {
      // Make value same as index.
      myArray[i] = i;
   }
   s += myArray.toLocaleString();
   return(s);
}
function toLocaleStringDate() {
   // Declare variables.
   var d = new Date();
   var s = "Current date setting is ";
   // Convert to current locale.
   s += d.toLocaleString();
   return(s);
}
function toLocaleStringNumber() {
   var n = Math.PI;
   var s = "The value of Pi is: ";
   s+= n.toLocaleString();
   return(s);
}
```

Requirements

Version 1

See Also

Methods

Applies To: Array Object | Date Object | Number Object | Object Object

toLocaleTimeString Method

Returns a time as a string value appropriate to the host environment's current locale.

```
function toLocaleTimeString() : String
```

Remarks

The **toLocaleTimeString** method returns a string value that contains a time, in the current time zone, in an easily read format. The time is in the default format of the host environment's current locale. The return value of this method cannot be relied upon in scripting, as it will vary from computer to computer. The **toLocalTimeString** method should only be used to format display – never as part of a computation.

Requirements

Version 5.5

See Also

ToTimeString Method | toLocaleDateString Method

Applies To: Date Object

toLocaleUpperCase Method

Returns a string where all alphabetic characters have been converted to uppercase, taking into account the host environment's current locale.

```
function tolocaleUpperCase() : String
```

Remarks

The **toLocaleUpperCase** method converts the characters in a string, taking into account the host environment's current locale. In most cases, the results are the same as you would obtain with the **toUpperCase** method. Results differ if the rules for a language conflict with the regular Unicode case mappings.

Requirements

Version 5.5

See Also

toLocaleLowerCase Method | toUpperCase Method

Applies To: String Object

toLowerCase Method

Returns a string where all alphabetic characters have been converted to lowercase.

```
function toLowerCase() : String
```

Remarks

The **toLowerCase** method has no effect on non-alphabetic characters.

Example

The following example demonstrates the effects of the **toLowerCase** method:

```
var strVariable = "This is a STRING object";
strVariable = strVariable.toLowerCase();
```

The value of `strVariable` after the last statement is:

```
this is a string object
```

Requirements

Version 1

See Also

toUpperCase Method

Applies To: String Object

toPrecision Method

Returns a string containing a number represented either in exponential or fixed-point notation with a specified number of digits.

```
function toPrecision ( [precision : Number] ) : String
```

Arguments

precision
Optional. Number of significant digits. Must be in the range 1 – 21, inclusive.

Remarks

For numbers in exponential notation, *precision - 1* digits are returned after the decimal point. For numbers in fixed notation, *precision* significant digits are returned.

If precision is not supplied or is **undefined**, the **toString** method is called instead.

Requirements

Version 5.5

See Also

toFixed Method | toExponential Method

Applies To: Number Object

toString Method

Returns a string representation of an object.

```
function toString( [radix : Number] ) : String
```

Arguments

radix
Optional. Specifies a radix for converting numeric values to strings. This value is only used for numbers.

Remarks

The **toString** method is a member of all built-in JScript objects. How it behaves depends on the object type:

Object	Behavior
Array	Elements of an **Array** are converted to strings. The resulting strings are concatenated, separated by commas.
Boolean	If the Boolean value is **true**, returns "true". Otherwise, returns "false".
Date	Returns the textual representation of the date.
Error	Returns a string containing the associated error message.
Function	Returns a string of the following form, where *functionname* is the name of the function whose **toString** method was called: "function functionname() { [native code] }"
Number	Returns the textual representation of the number.
String	Returns the value of the **String** object.
Default	Returns "[object objectname]", where objectname is the name of the object type.

Example

The following example illustrates the use of the **toString** method with a radix argument. The return value of function shown below is a Radix conversion table.

```
function CreateRadixTable (){
   var s, s1, s2, s3, x;                    //Declare variables.
   s = "Hex    Dec   Bin \n";               //Create table heading.
   for (x = 0; x < 16; x++)                 //Establish size of table
   {                                        // in terms of number of
      switch(x)                             // values shown.
      {                                     //Set intercolumn spacing.
         case 0 :
            s1 = "      ";
            s2 = "    ";
            s3 = "   ";
            break;
         case 1 :
            s1 = "      ";
            s2 = "    ";
            s3 = "   ";
            break;
         case 2 :
            s3 = "  ";
            break;
         case 3 :
            s3 = "  ";
            break;
         case 4 :
            s3 = " ";
            break;
         case 5 :
            s3 = " ";
            break;
         case 6 :
            s3 = " ";
            break;
         case 7 :
            s3 = " ";
            break;
         case 8 :
            s3 = "" ;
            break;
         case 9 :
            s3 = "";
            break;
         default:
            s1 = "     ";
            s2 = "";
            s3 = "    ";
      }                                     //Convert to hex, decimal & binary.
      s += " " + x.toString(16) + s1 + x.toString(10)
      s +=  s2 + s3 + x.toString(2)+ "\n";

   }
   return(s);                               //Return entire radix table.
}
```

Requirements

Version 2

See Also

function Statement

Applies To: Array Object | Boolean Object | Date Object | Error Object | Function Object | Number Object | Object Object | String Object

toTimeString Method

Returns a time as a string value.

```
function toTimeString() : String
```

Remarks

The **toTimeString** method returns a string value containing the time, in the current time zone, in a convenient, easily read format.

Requirements

Version 5.5

See Also

toDateString Method | toLocaleTimeString Method

Applies To: Date Object

toUpperCase Method

Returns a string where all alphabetic characters have been converted to uppercase.

```
function toUpperCase() : String
```

Remarks

The **toUpperCase** method has no effect on non-alphabetic characters.

Example

The following example demonstrates the effects of the **toUpperCase** method:

```
var strVariable = "This is a STRING object";
strVariable = strVariable.toUpperCase();
```

The value of `strVariable` after the last statement is:

```
THIS IS A STRING OBJECT
```

Requirements

Version 1

See Also

toLowerCase Method

Applies To: String Object

toUTCString Method

Returns a date converted to a string using Coordinated Universal Time (UTC).

```
function toUTCString() : String
```

Remarks

The **toUTCString** method returns a **String** object that contains the date formatted using UTC convention in a convenient, easily read form.

Example

The following example illustrates the use of the **toUTCString** method.

```
function toUTCStrDemo(){
   var d, s;                   //Declare variables.
   d = new Date();             //Create Date object.
   s = "Current setting is ";
   s += d.toUTCString();       //Convert to UTC string.
   return(s);                  //Return UTC string.
}
```

Requirements

Version 3

See Also

toGMTString Method

Applies To: Date Object

ubound Method

Returns the highest index value used in the specified dimension of the VBArray.

```
function ubound( [dimension : Number] ) : Number
```

Arguments

dimension
Optional. The dimension of the VBArray for which the higher bound index is wanted. If omitted, **ubound** behaves as if a 1 was passed.

Remarks

If the VBArray is empty, the **ubound** method returns undefined. If *dimension* is greater than the number of dimensions in the VBArray, or is negative, the method generates a "Subscript out of range" error.

Example

The following example consists of three parts. The first part is VBScript code to create a Visual Basic safe array. The second part is JScript code that determines the number of dimensions in the safe array and the upper bound of each dimension. Both of these parts go into the <HEAD> section of an HTML page. The third part is the JScript code that goes in the <BODY> section to run the other two parts.

```
<HEAD>
<SCRIPT LANGUAGE="VBScript">
<!--
Function CreateVBArray()
   Dim i, j, k
   Dim a(2, 2)
   k = 1
   For i = 0 To 2
      For j = 0 To 2
         a(j, i) = k
         k = k + 1
      Next
   Next
   CreateVBArray = a
End Function
-->
</SCRIPT>

<SCRIPT LANGUAGE="JScript">
<!--
function VBArrayTest(vba)
{
   var i, s;
   var a = new VBArray(vba);
   for (i = 1; i <= a.dimensions(); i++)
   {
      s = "The upper bound of dimension ";
      s += i + " is ";
      s += a.ubound(i)+ ".<BR>";
      return(s);
   }
}
-->
</SCRIPT>
</HEAD>

<BODY>
<SCRIPT language="jscript">
   document.write(VBArrayTest(CreateVBArray()));
```

```
</SCRIPT>
</BODY>
```

Requirements

Version 3

See Also

dimensions Method | getItem Method | lbound Method | toArray Method

Applies To: VBArray Object

unescape Method

Returns as decoded string from a **String** object encoded with the **escape** method.

```
function unescape(charString : String) : String
```

Arguments

charString
Required. A **String** object or literal to be decoded.

Remarks

The **unescape** method returns a string value that contains the contents of *charString*. All characters encoded with the %*xx* hexadecimal form are replaced by their ASCII character set equivalents.

Characters encoded in **%u**_xxxx_ format (Unicode characters) are replaced with the Unicode character with hexadecimal encoding *xxxx*.

> **Note** The **unescape** method should not be used to decode Uniform Resource Identifiers (URI). Use **decodeURI** and **decodeURIComponent** methods instead.

Requirements

Version 1

See Also

DecodeURI Method | decodeURIComponent Method | escape Method | String Object

Applies To: Global Object

unshift Method

Inserts specified elements into the beginning of an array.

```
function unshift([item1 : Object [, ... [, itemN : Object]]]) : Array
```

Arguments

item1, ... , itemN
Optional. Elements to insert at the start of the **Array**.

Remarks

The **unshift** method inserts elements into the start of an array, so they appear in the same order in which they appear in the argument list.

Requirements

Version 5.5

See Also

shift Method

Applies To: Array Object

UTC Method

Returns the number of milliseconds between midnight, January 1, 1970 Coordinated Universal Time (UTC) (or GMT) and the supplied date.

```
function UTC(year : Number , month : Number , day : Number [, hours : Number [, minutes : Num
ber [, seconds : Number [,ms : Number]]]]) : Number
```

Arguments

year
Required. The full year designation is required for cross-century date accuracy. If *year* is between 0 and 99 is used, then *year* is assumed to be 1900 + *year*.

month
Required. The month as an integer between 0 and 11 (January to December).

day
Required. The date as an integer between 1 and 31.

hours
Optional. Must be supplied if *minutes* is supplied. An integer from 0 to 23 (midnight to 11pm) that specifies the hour.

minutes
Optional. Must be supplied if *seconds* is supplied. An integer from 0 to 59 that specifies the minutes.

seconds
Optional. Must be supplied if *milliseconds* is supplied. An integer from 0 to 59 that specifies the seconds.

ms
Optional. An integer from 0 to 999 that specifies the milliseconds.

Remarks

The **UTC** method returns the number of milliseconds between midnight, January 1, 1970 UTC and the supplied date. This return value can be used in the **setTime** method and in the **Date** object constructor. If the value of an argument is greater than its range, or is a negative number, other stored values are modified accordingly. For example, if you specify 150 seconds, JScript redefines that number as two minutes and 30 seconds.

The difference between the **UTC** method and the **Date** object constructor that accepts a date is that the **UTC** method assumes UTC, and the **Date** object constructor assumes local time.

The **UTC** method is a static method. Therefore, a **Date** object does not have to be created before it can be used.

Note If *year* is between 0 and 99, use *1900 + year* for the year.

Example

The following example illustrates the use of the **UTC** method.

```
function DaysBetweenDateAndNow(yr, mo, dy){
   var d, r, t1, t2, t3;            //Declare variables.
   var MinMilli = 1000 * 60         //Initialize variables.
   var HrMilli = MinMilli * 60
   var DyMilli = HrMilli * 24
   t1 = Date.UTC(yr, mo - 1, dy)    //Get milliseconds since 1/1/1970.
   d = new Date();                  //Create Date object.
   t2 = d.getTime();                //Get current time.
   if (t2 >= t1)
      t3 = t2 - t1;
   else
      t3 = t1 - t2;
   r = Math.round(t3 / DyMilli);
   return(r);                       //Return difference.
}
```

Requirements

Version 1

See Also

setTime Method

Applies To: Date Object

valueOf Method

Returns the primitive value of the specified object.

```
function valueOf() : Object
```

Remarks

The **valueOf** method is defined differently for each intrinsic JScript object.

Object	**Return Value**
Array	The elements of the array are converted into strings, and the strings are concatenated together, separated by commas. This behaves the same as the **Array.toString** and **Array.join** methods.
Boolean	The Boolean value.
Date	The stored time value in milliseconds since midnight, January 1, 1970 UTC.
Function	The function itself.
Number	The numeric value.
Object	The object itself. This is the default.
String	The string value.

The **Math** and **Error** objects do not have a **valueOf** method.

Requirements

Version 2

See Also

toString Method

Applies To: Array Object | Boolean Object | Date Object | Function Object | Number Object | Object Object | String Object

Modifiers

JScript modifiers are used to affect the behavior and visibility of classes, interfaces, or members of classes or interfaces. You can use modifiers when defining classes and interfaces, but they are not required.

In This section

abstract Modifier
Inheritance modifier that allows for definition of classes and class members, but does not allow implementations to be given.

expando Modifier
Compatibility modifier that marks a class as dynamically extensible or a method as an expando object constructor.

final Modifier
Inheritance modifier that prevents a class from being extended or prevents a method or property from being overridden.

hide Modifier
Version-safe modifier that prevents a method or property from overriding a method or property in a base class.

internal Modifier
Visibility modifier that makes a class, interface, or member visible only to the current package.

override Modifier
Version-safe modifier for explicitly overriding a method in a base class.

private Modifier
Visibility modifier that makes a class member visible only to members of the same class.

protected Modifier
Visibility modifier that makes a member of a class or interface visible only to the current class or interface and derived classes of the current class.

public Modifier
Visibility modifier that makes members of a class or interface visible to any code that has access to the class or interface.

static Modifier
Modifier that marks a class member as belonging to the class itself.

Related Sections

JScript Modifiers
Conceptual overview of the purpose and uses of JScript .NET modifiers.

abstract Modifier

Declares that a class must be extended or that the implementation for a method or property must be provided by a derived class.

```
abstract statement
```

Arguments

statement
Required. A class, method, or property definition.

Remarks

The **abstract** modifier is used for a method or property in a class that has no implementation or for a class that contains such methods. A class with abstract members cannot be instantiated with the **new** operator. You can derive both abstract and non-abstract classes from an abstract base class.

Methods and properties in classes and classes can be marked with the **abstract** modifier. A class must be marked as **abstract** if it contains any **abstract** members. Interfaces and members of interfaces, which are implicitly abstract, cannot take the **abstract** modifier. Fields cannot be **abstract**.

You may not combine the **abstract** modifier with the other inheritance modifier (**final**). By default, class members are neither **abstract** nor **final**. The inheritance modifiers cannot be combined with the **static** modifier.

Example

The following example illustrates a use of the **abstract** modifier.

```
// CAnimal is an abstract base class.
abstract class CAnimal {
   abstract function printQualities();
}
// CDog and CKangaroo are derived classes of CAnimal.
class CDog extends CAnimal {
   function printQualities() {
      print("A dog has four legs.");
   }
}
class CKangaroo extends CAnimal {
   function printQualities() {
      print("A kangaroo has a pouch.");
   }
}

// Define animal of type CAnimal.
var animal : CAnimal;

animal = new CDog;
// animal uses printQualities from CDog.
animal.printQualities();

animal = new CKangaroo;
// animal uses printQualities from CKangaroo.
animal.printQualities();
```

The output of this program is:

```
A dog has four legs.
A kangaroo has a pouch.
```

Requirements

Version .NET

See Also

Modifiers | final Modifier | static Modifier | var Statement | function Statement | class Statement | Variable Scope | new Operator

expando Modifier

Declares that instances of a class support expando properties or that a method is an expando object constructor.

```
expando statement
```

Arguments

statement
Required. A class or method definition.

Remarks

The **expando** modifier is used to mark a class as dynamically extensible (one that supports expando properties). Expando properties on **expando** class instances must be accessed using the **[]** notation; they are not accessible with the dot operator. The **expando** modifier also marks a method as an **expando** object constructor.

Classes and methods in classes can be marked with the **expando** modifier. Fields, properties, interfaces, and members of interfaces cannot take the **expando** modifier.

An **expando** class has a hidden, private property named **Item** that takes one **Object** parameter and returns an **Object**. You are not allowed to define a property with this signature on an **expando** class.

Example 1

The following example illustrates a use of the **expando** modifier on a class. The expando class is like a JScript **Object**, but there are some differences that are illustrated here.

```
expando class CExpandoExample {
   var x : int = 10;
}

// New expando class-based object.
var testClass : CExpandoExample = new CExpandoExample;
// New JScript Object.
var testObject : Object = new Object;

// Add expando properties to both objects.
testClass["x"] = "ten";
testObject["x"] = "twelve";

// Access the field of the class-based object.
print(testClass.x);      // Prints 10.
// Access the expando property.
print(testClass["x"]);   // Prints ten.

// Access the property of the class-based object.
print(testObject.x);     // Prints twelve.
// Access the same property using the [] operator.
print(testObject["x"]);  // Prints twelve.
```

The output of this code is

```
10
ten
twelve
twelve
```

Example 2

The following example illustrates a use of the **expando** modifier on a method. When the expando method is called in the usual way, it accesses the field *x*. When the method is used as an explicit constructor with the **new** operator, it adds an expando property to a new object.

```
class CExpandoExample {
   var x : int;
   expando function constructor(val : int) {
      this.x = val;
      return "Method called as a function.";
   }
}

var test : CExpandoExample = new CExpandoExample;
// Call the expando method as a function.
var str = test.constructor(123);
print(str);        // The return value is a string.
print(test.x);     // The value of x has changed to 123.

// Call the expando method as a constructor.
var obj = new test.constructor(456);
// The return value is an object, not a string.
print(obj.x);      // The x property of the new object is 456.
print(test.x);     // The x property of the original object is still 123.
```

The output of this code is

```
Method called as a function.
123
456
123
```

Requirements

Version .NET

See Also

Modifiers | static Modifier | var Statement | function Statement | class Statement | Variable Scope | Type Annotation

final Modifier

Declares that a class cannot be extended or that a method or property cannot be overridden.

```
final statement
```

Arguments

statement
Required. A class, method, or property definition.

Remarks

The **final** modifier is used to specify that a class cannot be extended or that a method or property cannot be overridden. This prevents other classes from changing the behavior of the class by overriding important functions. Methods with the **final** modifier can be hidden or overloaded by methods in derived classes.

Methods and properties in classes and classes can be marked with the **final** modifier. Interfaces, fields, and members of interfaces cannot take the **final** modifier.

You may not combine the **final** modifier with the other inheritance modifier (**abstract**). By default, class members are neither **abstract** nor **final**. The inheritance modifiers cannot be combined with the **static** modifier.

Example

The following example illustrates a use of the **final** modifier. The **final** modifier prevents the base-class method from being overridden by methods from the derived class.

```
class CBase {
   final function methodA() { print("Final methodA of CBase.") };
   function methodB() { print("Non-final methodB of CBase.") };
}

class CDerived extends CBase {
   function methodA() { print("methodA of CDerived.") };
   function methodB() { print("methodB of CDerived.") };
}

var baseInstance : CBase = new CDerived;
baseInstance.methodA();
baseInstance.methodB();
```

The output of this program show that the final method is not overridden:

```
Final methodA of CBase.
methodB of CDerived.
```

Requirements

Version .NET

See Also

Modifiers | abstract Modifier | hide Modifier | override Modifier | var Statement | function Statement | class Statement | Variable Scope | Type Annotation

hide Modifier

Declares that a method or property hides a method or property in a base class.

```
hide statement
```

Arguments

statement
Required. A method or property definition.

Remarks

The **hide** modifier is used for a method that hides a method in a base class. You are not allowed to use the **hide** modifier for a method unless the base class has a member with the same signature.

Methods and properties in classes can be marked with the **hide** modifier. Classes, fields, interfaces and members of interfaces cannot take the **hide** modifier.

You may not combine the **hide** modifier with the other version-safe modifier (**override**). The version-safe modifiers cannot be combined with the **static** modifier. By default, a method will override a base-class method unless the base-class method has the **final** modifier. You cannot hide an **abstract** method unless you provide an explicit implementation for the abstract, base method. When running in version-safe mode, one of the version-safe modifiers must be used whenever a base-class method is overridden.

Example

The following example illustrates a use of the **hide** modifier. The method in the derived class marked with the **hide** modifier does not override the base-class method. The method marked with **override** does override the base-class method.

```
class CBase {
   function methodA() { print("methodA of CBase.") };
   function methodB() { print("methodB of CBase.") };
}

class CDerived extends CBase {
   hide function methodA() { print("Hiding methodA.") };
   override function methodB() { print("Overriding methodB.") };
}


var derivedInstance : CDerived = new CDerived;
derivedInstance.methodA();
derivedInstance.methodB();

var baseInstance : CBase = derivedInstance;
baseInstance.methodA();
baseInstance.methodB();
```

The output of this program shows that a hidden method does not override a base class method.

```
Hiding methodA.
Overriding methodB.
methodA of CBase.
Overriding methodB.
```

Requirements

Version .NET

See Also

Modifiers | override Modifier | static Modifier | var Statement | function Statement | class Statement | Variable Scope | Type Annotation | /versionsafe

internal Modifier

Declares that a class, class member, interface, or interface member has internal visibility.

```
internal statement
```

Arguments

statement
Required. A class, interface, or member definition.

Remarks

The **internal** modifier makes a class, interface, or member visible only within the current package. Code outside the current package cannot access **internal** members.

Classes and interfaces can be marked with the **internal** modifier. In the global scope, the **internal** modifier is the same as the **public** modifier. Any member of a class or interface can be marked with the **internal** modifier.

You may not combine the **internal** modifier with any of the other visibility modifiers (**public**, **private**, or **protected**). Visibility modifiers are relative to the scope in which they are defined. For example, a **public** method of an **internal** class is not publicly accessible, but any code that has access to the class can access the method.

Requirements

Version .NET

See Also

Modifiers | public Modifier | private Modifier | protected Modifier | var Statement | function Statement | class Statement | Variable Scope

override Modifier

Declares that a method or property overrides a method or property in a base class.

```
override statement
```

Arguments

statement
Required. A method or property definition.

Remarks

The **override** modifier is used for a method that overrides a method in a base class. You are not allowed to use the **override** modifier for a method unless the base class has a member with the same signature.

Methods and properties in classes can be marked with the **override** modifier. Classes, fields, interfaces and members of interfaces cannot take the **override** modifier.

You may not combine the **override** modifier with the other version-safe modifier (**hide**). The version-safe modifiers cannot be combined with the **static** modifier. By default, a method will override a base-class method unless the base-class method has the **final** modifier. You cannot override a **final** method. When running in version-safe mode, one of the version-safe modifiers must be used whenever a base-class method is overridden.

Example

The following example illustrates a use of the **override** modifier. The method in the derived class marked with the **override** modifier overrides the base-class method. The method marked with the **hide** modifier does not override the base class method.

```
class CBase {
   function methodA() { print("methodA of CBase.") };
   function methodB() { print("methodB of CBase.") };
}

class CDerived extends CBase {
   hide function methodA() { print("Hiding methodA.") };
   override function methodB() { print("Overriding methodB.") };
}


var derivedInstance : CDerived = new CDerived;
derivedInstance.methodA();
derivedInstance.methodB();

var baseInstance : CBase = derivedInstance;
baseInstance.methodA();
baseInstance.methodB();
```

The output of this program shows that an **override** method overrides a base-class method.

```
Hiding methodA.
Overriding methodB.
methodA of CBase.
Overriding methodB.
```

Requirements

Version .NET

See Also

Modifiers | hide Modifier | var Statement | function Statement | class Statement | Variable Scope | Type Annotation

private Modifier

Declares that a class member has private visibility.

```
private statement
```

Arguments

statement
Required. A class member definition.

Remarks

The **private** modifier makes a member of a class visible only within that class. Code outside the current class, including derived classes, cannot access **private** members.

Classes and interfaces in the global scope cannot be marked with the **private** modifier. Any member of a class or interface (including nested classes and nested interfaces) can be marked with the **private** modifier.

You may not combine the **private** modifier with any of the other visibility modifiers (**public**, **protected**, or **internal**).

Requirements

Version .NET

See Also

Modifiers | public Modifier | protected Modifier | internal Modifier | var Statement | function Statement | class Statement | Variable Scope

protected Modifier

Declares that a class member or interface member has protected visibility.

```
protected statement
```

Arguments

statement
Required. A class member or interface member definition.

Remarks

The **protected** modifier makes a member of a class or interface visible only within that class or interface and all derived classes of the current class. Code outside the current class cannot access **protected** members.

Classes and interfaces in the global scope cannot be marked with the **protected** modifier. Any member of a class or interface (including nested classes and nested interfaces) can be marked with the **protected** modifier.

You may not combine the **protected** modifier with any of the other visibility modifiers (**public**, **private**, or **internal**).

Requirements

Version .NET

See Also

Modifiers | public Modifier | private Modifier | internal Modifier | var Statement | function Statement | class Statement | Variable Scope

public Modifier

Declares that a class, interface, or member has public visibility.

```
public statement
```

Arguments

statement
Required. A class, interface, or member definition.

Remarks

The **public** modifier makes a member of a class visible to any code that has access to the class.

All classes and interfaces are **public** by default. A member of a class or interface can be marked with the **public** modifier.

You may not combine the **public** modifier with any of the other visibility modifiers (**private**, **protected**, or **internal**).

Requirements

Version .NET

See Also

Modifiers | private Modifier | protected Modifier | internal Modifier | var Statement | function Statement | class Statement | Variable Scope

static Modifier

Declares that a class member belongs to a class rather than to instances of the class.

```
static statement
```

Arguments

statement
Required. A class member definition.

Remarks

The **static** modifier signifies that a member belongs to the class itself rather than to instances of the class. Only one copy of a **static** member exists in a given application even if many instances of the class are created. You can only access **static** members with a reference to the class rather than a reference to an instance. However, within a class member declaration, **static** members can be accessed with the **this** object.

Members of classes can be marked with the **static** modifier. Classes, interfaces, and members of interfaces cannot take the **static** modifier.

You may not combine the **static** modifier with any of the inheritance modifiers (**abstract** and **final**) or version-safe modifiers (**hide** and **override**).

Do not confuse the **static** modifier with the **static** statement. The **static** modifier denotes a member that belongs to the class itself rather than any instance of the class.

Example

The following example illustrates a use of the **static** modifier.

```
class CTest {
   var nonstaticX : int;      // A non-static field belonging to a class instance.
   static var staticX : int;  // A static field belonging to the class.
}

// Initialize staticX. An instance of test is not needed.
CTest.staticX = 42;

// Create an instance of test class.
var a : CTest = new CTest;
a.nonstaticX = 5;
// The static field is not directly accessible from the class instance.

print(a.nonstaticX);
print(CTest.staticX);
```

The output of this program is:

```
5
42
```

Requirements

Version .NET

See Also

Modifiers | expando Modifier | var Statement | function Statement | class Statement | Variable Scope | Type Annotation | static Statement

Objects

JScript objects are collections of properties and methods. The following sections link to information that explains how to use JScript objects.

> **Note** The JScript runtime is not designed to be thread-safe. Consequently, JScript objects and methods may have unpredictable behavior when used in multithreaded applications.

In This Section

ActiveXObject Object
Enables and returns a reference to an Automation object.

arguments Object
Provides access to the arguments passed to the current function.

Array Object
Provides support for creation of arrays of any data type.

Boolean Object
Creates a new Boolean value.

Date Object
Enables basic storage and retrieval of dates and times.

Enumerator Object
Enables enumeration of items in a collection.

Error Object
An object that contains information about errors that occur while JScript code is running.

Function Object
Creates a new function.

Global Object
An intrinsic object whose purpose is to collect global methods into one object.

Math Object
An intrinsic object that provides basic mathematics functionality and constants.

Number Object
An object representation of the number data type and placeholder for numeric constants.

Object Object
Provides functionality common to all JScript objects.

RegExp Object
Stores information on regular expression pattern searches.

Regular Expression Object
Contains a regular expression pattern.

String Object
Allows manipulation and formatting of text strings and determination and location of substrings within strings.

VBArray Object
Provides access to Visual Basic safe arrays.

Related Sections

JScript Reference
Lists elements that comprise JScript Language Reference and links to topics that explain the details behind the proper use of language elements.

Methods
Lists the methods, classified alphabetically, available in JScript, and links to each category of methods.

Properties
Provides a list of properties available in JScript and links to topics that explain the proper syntax of use of each property.

JScript Objects
Explains the concept of objects in JScript, how objects are related to properties and methods, and links to topics that provide more detail about the objects that JScript supports.

ActiveXObject Object

An object that provides an interface to an Automation object.

```
function ActiveXObject(ProgID : String [, location : String])
```

Arguments

ProgID
Required. A string of the form "*serverName.typeName*", where *serverName* is the name of the application providing the object, and *typeName* is the name of the type or class of the object to create.
location
Optional. The name of the network server where the object is to be created.

Remarks

Typically, an automation server provides at least one type of object. For example, a word-processing application may provide an application object, a document object, and a toolbar object.

The following code starts an application (in this case, a Microsoft Excel worksheet) by calling the **ActiveXObject** object constructor. The **ActiveXObject** allows you to refer to the application in your code. Using the following example, you can access properties and methods of the new object using the object variable `ExcelSheet` and other Excel objects, including the Application object and the ActiveSheet.Cells collection.

```
// Declare the variables
var Excel, Book;

// Create the Excel application object.
Excel = new ActiveXObject("Excel.Application");

// Make Excel visible.
Excel.Visible = true;

// Create a new work book.
Book = Excel.Workbooks.Add()

// Place some text in the first cell of the sheet.
Book.ActiveSheet.Cells(1,1).Value = "This is column A, row 1";

// Save the sheet.
Book.SaveAs("C:\\TEST.XLS");

// Close Excel with the Quit method on the Application object.
Excel.Application.Quit();
```

Creating an object on a remote server can only be accomplished when Internet security is turned off. You can create an object on a remote networked computer by passing the name of the computer to the *servername* argument of **ActiveXObject**. That name is the same as the machine name portion of a sharename. For a network share named "\\MyServer\public", the *servername* is "MyServer". In addition, you can specify *servername* using DNS format or an IP address.

The following code returns the version number of an instance of Excel running on a remote network computer named "MyServer":

```
function GetAppVersion() {
   var Excel = new ActiveXObject("Excel.Application", "MyServer");
   return(Excel.Version);
}
```

An error occurs if the specified remote server does not exist or cannot be found.

Properties and Methods

An **ActiveXObject** object has no intrinsic properties or methods; it allows you to access the properties and methods of the Automation object.

Requirements

Version 1

See Also

new Operator | GetObject Function

arguments Object

An object representing the currently executing function, its arguments, and the function that called it. This object cannot be constructed explicitly.

Remarks

An **arguments** object is instantiated for each function when it begins execution. The **arguments** object is directly accessible only within the scope of its associated function.

All parameters passed to a function and the number of parameters are stored in the **arguments** object. The **arguments** object is not an array, but the individual arguments are accessed using [ ] notation, the same way array elements are accessed.

You can use the **arguments** object to create functions that can accept an arbitrary number of arguments. This functionality can also be achieved by using the parameter array construction when defining your function. For more information, see the **function** statement topic.

> **Note**   The **arguments** object is not available when running in fast mode, the default for JScript .NET. To compile a program from the command line that uses the **arguments** object, you must turn off the fast option by using **/fast-**. It is not safe to turn off the fast option in ASP.NET because of threading issues.

Example

The following example illustrates the use of the **arguments** object.

```
function argTest(a, b) : String {
   var i : int;
   var s : String = "The argTest function expected ";
   var numargs : int = arguments.length; // Get number of arguments passed.
   var expargs : int = argTest.length;   // Get number of arguments expected.
   if (expargs < 2)
      s += expargs + " argument. ";
   else
      s += expargs + " arguments. ";
   if (numargs < 2)
      s += numargs + " was passed.";
   else
      s += numargs + " were passed.";
   s += "\n"
   for (i =0 ; i < numargs; i++){        // Get argument contents.
      s += "  Arg " + i + " = " + arguments[i] + "\n";
   }
   return(s);                            // Return list of arguments.
}

print(argTest(42));
print(argTest(new Date(1999,8,7),"Sam",Math.PI));
```

The output of this program is:

```
The argTest function expected 2 arguments. 1 was passed.
  Arg 0 = 42

The argTest function expected 2 arguments. 3 were passed.
  Arg 0 = Tue Sep 7 00:00:00 PDT 1999
  Arg 1 = Sam
  Arg 2 = 3.141592653589793
```

Properties

arguments Object Properties

Methods

The **arguments** object has no methods.

Requirements

Version 1

See Also

new Operator | function Statement | /fast

arguments Object Properties

The arguments object represents the arguments to the currently executing function and to the functions that called it.

Properties

0...n Properties

arguments Property

callee Property

caller Property

length Property (arguments)

See Also

Properties | JScript Reference

Array Object

Provides support for expando arrays of any data type. There are three forms of the **Array** constructor.

```
function Array( [size : int] )
function Array( [... varargs : Object[]] )
function Array( [array : System.Array )
```

Arguments

size
Optional. The size of the array. As arrays are zero-based, created elements will have indexes from zero to *size* -1.

varargs
Optional. A typed array that contains all the parameters passed to the constructor. These parameters are used as the first elements of the array.

array
Optional. An array to be copied to the array being constructed.

Remarks

If only one argument is passed to the **Array** constructor and the argument is a number, it must be an unsigned 32-bit integer (any integer less than approximately four billion). The passed value is the size of the array. If the value is a number that is less than zero or is not an integer, a run-time error occurs.

A variable of data type **System.Array** can be passed to the **Array** constructor. This produces a JScript array that is a copy of the input array. The **System.Array** must have only one dimension.

If a single value is passed to the **Array** constructor and it is not a number or an array, the **length** property of the array is set to 1, and the value of the first element of the array (element 0) becomes the single, passed-in argument. If several arguments are passed to the constructor, the length of the array is set to the number of arguments, and those arguments will be the first elements in the new array.

Notice that JScript arrays are sparse arrays; that is, although you can allocate an array with many elements, only the elements that actually contain data exist. This reduces the amount of memory used by the array.

The **Array** object interoperates with **System.Array** data type. Consequently, an **Array** object can call the methods and properties of the **System.Array** data type, and a **System.Array** data type can call the methods and properties of the **Array** object. Furthermore, **Array** objects are accepted by functions that take **System.Array** data types, and vice versa. For more information, see Array Members.

When an **Array** object is passed to a function that takes a **System.Array** or when **System.Array** methods are called from an **Array** object, the contents of the **Array** are copied. Thus, the original **Array** object cannot be modified by the **System.Array** methods or by passing it to a function that accepts a **System.Array**. Only nondestructive **Array** methods can be called on a **System.Array**.

> **Tip** **Array** objects are convenient when you want a generic stack or a list of items and performance is not a top concern. In all other contexts, typed array data types should be used. A typed array, which has much of the same functionality as the **Array** object, also provides type safety, performance improvements, and better interaction with other languages.

> **Note** The **Array** object interoperates with the .NET Framework **System.Array** data type within JScript .NET. However, other Common Language Specification (CLS) languages cannot use the **Array** object because only JScript .NET provides the object; it is not derived from a .NET Framework type. Consequently, when type-annotating the parameters and return types of CLS-compliant methods, make sure to use the **System.Array** data type instead of the **Array** object. However, you may use the **Array** object to type annotate identifiers other than the parameters or return types. For more information, see Writing CLS-Compliant Code.

Example

The individual elements of the array can be accessed using [ ] notation. For example:

```
var my_array = new Array();
```

```
for (var i = 0; i < 10; i++) {
   my_array[i] = i;
}
var x = my_array[4];
```

Since arrays in Microsoft JScript are zero-based, the final statement in the preceding example accesses the fifth element of the array. That element contains the value 4.

Properties and Methods

Array Object Properties and Methods

Requirements

Version 2

See Also

new Operator | Typed Arrays

Array Object Properties and Methods

The Array object provides support for creation of arrays of any data type.

Properties

constructor Property

length Property (Array)

prototype Property

Methods

concat Method (Array)

join Method

pop Method

push Method

reverse Method

shift Method

slice Method (Array)

sort Method

splice Method

toLocaleString Method

toString Method

unshift Method

valueOf Method

See Also

Properties | Methods | JScript Reference

Boolean Object

The **Boolean** object references a Boolean value.

```
function Boolean( [boolValue : boolean] )
```

Arguments

boolValue
Optional. The initial Boolean value for the new object. If *boolValue* is omitted, or is **false**, 0, **null**, **NaN**, or an empty string, the initial value of the Boolean object is **false**. Otherwise, the initial value is **true**.

Remarks

The **Boolean** object is a wrapper for Boolean data. The primary purposes for the **Boolean** object are to collect its properties into one object and to allow Boolean values to be converted into strings via the **toString** method. The **Boolean** object is similar to the **boolean** data type. However, they have different properties and methods.

> **Note** You rarely need to construct a Boolean object explicitly. The boolean data type should be used in most circumstances. Since the Boolean object interoperates with the boolean data type, all Boolean object methods and properties are available to a variable of type Boolean. For more information, see boolean Data Type.

The data type of a **Boolean** object is **Object**, not **boolean**.

Properties and Methods

Boolean Object Properties and Methods

Requirements

Version 2

See Also

Object Object | | Boolean Structure | new Operator | var Statement

Boolean Object Properties and Methods

The Boolean object creates a new Boolean value.

Properties

constructor Property

prototype Property

Methods

toString Method

valueOf Method

See Also

Properties | Methods | JScript Reference

Date Object

An object that enables basic storage and retrieval of dates and times. There are two forms of the **Date** constructor.

```
function Date( [dateVal : { Number | String | System.DateTime } ] )
function Date( year : int, month : int, date : int[, hours : int [, minutes : int [, seconds
: int [, ms : int]]]] )
```

Arguments

dateVal
Optional. If a numeric value, *dateVal* represents the number of milliseconds in Coordinated Universal Time between the specified date and midnight January 1, 1970. If a string, *dateVal* is parsed according to the rules in the **parse** method. The *dateVal* can also be a .NET date value.

year
Required. The full year, for example, 1976 (not 76).

month
Required. The month as an integer between 0 and 11 (January to December).

date
Required. The date as an integer between 1 and 31.

hours
Optional. Must be supplied if *minutes* is supplied. An integer from 0 to 23 (midnight to 11pm) that specifies the hour.

minutes
Optional. Must be supplied if *seconds* is supplied. An integer from 0 to 59 that specifies the minutes.

seconds
Optional. Must be supplied if *milliseconds* is supplied. An integer from 0 to 59 that specifies the seconds.

ms
Optional. An integer from 0 to 999 that specifies the milliseconds.

Remarks

A **Date** object contains a number representing a particular instance in time to within a millisecond. If the value of an argument is greater than its range or is a negative number, other stored values are modified accordingly. For example, if you specify 150 seconds, JScript redefines that number as two minutes and 30 seconds.

If the number is **NaN**, the object does not represent a specific instance in time. If you pass no parameters to the **Date** constructor, it is initialized to the current time (UTC). A variable of type **Date** must be initialized before you can use it.

The range of dates that can be represented in a **Date** object is approximately 285,616 years on either side of January 1, 1970.

The **Date** object has two static methods, **parse** and **UTC**, that are called without creating a **Date** object.

If the **Date** constructor is called without the **new** operator, the **Date** object that is returned contains the current date regardless of the arguments passed to the constructor.

> **Note** The **Date** object interoperates with the .NET Framework **System.DateTime** data type within JScript .NET. However, other Common Language Specification (CLS) languages cannot use the **Date** object because only JScript .NET provides the object; it is not derived from a .NET Framework type. Consequently, when type-annotating the parameters and return types of CLS-compliant methods, make sure to use the **System.DateTime** data type instead of the **Date** object. However, you may use the **Date** object to type annotate identifiers other than the parameters or return types. For more information, see Writing CLS-Compliant Code.

Example

The following example uses the **Date** object.

```
var s : String = "Today's date is: ";   // Declare variables.
var d : Date = new Date();              // Create Date object with today's date.
s += (d.getMonth() + 1) + "/";          // Get month
s += d.getDate() + "/";                 // Get day
s += d.getYear();                       // Get year.
print(s);                               // Print date.
```

If this program were run on January 26, 1992, the output would have been:

```
Today's date is: 1/26/1992
```

Properties and Methods

Date Object Properties and Methods

Requirements

Version 1

See Also

new Operator | var Statement

Date Object Properties and Methods

The Date object enables basic storage and retrieval of dates and times.

Properties

constructor Property

prototype Property

Methods

getDate Method

getDay Method

getFullYear Method

getHours Method

getMilliseconds Method

getMinutes Method

getMonth Method

getSeconds Method

getTime Method

getTimezoneOffset Method

getUTCDate Method

getUTCDay Method

getUTCFullYear Method

getUTCHours Method

getUTCMilliseconds Method

getUTCMinutes Method

getUTCMonth Method

getUTCSeconds Method

getVarDate Method

getYear Method

parse Method

setDate Method

setFullYear Method

setHours Method

setMilliseconds Method

setMinutes Method

setMonth Method

setSeconds Method

setTime Method

setUTCDate Method

setUTCFullYear Method

setUTCHours Method

setUTCMilliseconds Method

setUTCMinutes Method

setUTCMonth Method

setUTCSeconds Method

setYear Method

toDateString Method

toGMTString Method

toLocaleDateString Method

toLocaleString Method

toLocaleTimeString Method

toString Method

toTimeString Method

toUTCString Method

UTC Method

valueOf Method

See Also

Properties | Methods | JScript Reference

Enumerator Object

Enables enumeration of items in a collection.

```
varName = new Enumerator([collection])
```

Arguments

varName
Required. The variable name to which the enumerator is assigned.
collection
Optional. Any object that implements the **IEnumerable** interface, such as an array or collection.

Remarks

Every collection is automatically enumerable in JScript .NET. Consequently, you do not need to use the **Enumerator** object to access members of a collection. You can access any member directly using the **for...in** statement. The **Enumerator** object is provided for backwards compatibility.

Collections differ from arrays in that the members of a collection are not directly accessible. Instead of using indexes, as you would with arrays, you can only move the current item pointer to the first or next element of a collection.

The **Enumerator** object, which provides a way to access any member of a collection, behaves in a manner similar to the **For...Each** statement in VBScript.

You can create a collection in JScript by defining a class that implements **IEnumerable**. Collections can also be created using another language (such as Visual Basic) or by an **ActiveXObject** object.

Example 1

The following code uses the **Enumerator** object to print the letters of the available drives and their names (if available):

```
// Declare variables.
var n, x;
var fso : ActiveXObject = new ActiveXObject("Scripting.FileSystemObject");
// Create Enumerator on Drives.
var e : Enumerator = new Enumerator(fso.Drives);
for (;!e.atEnd();e.moveNext()) {      // Loop over the drives collection.
   x = e.item();
   if (x.DriveType == 3)              // See if network drive.
      n = x.ShareName;                // Get share name
   else if (x.IsReady)                // See if drive is ready.
      n = x.VolumeName;               // Get volume name.
   else
      n = "[Drive not ready]";
   print(x.DriveLetter + " - " + n);
}
```

Depending on the system, the output will look like this:

```
A - [Drive not ready]
C - DRV1
D - BACKUP
E - [Drive not ready]
```

Example 2

The code in Example 1 can be rewritten to work without using the Enumerator object. Here, members of an enumeration are accessed directly.

```
// Declare variables.
```

```
var n, x;
var fso : ActiveXObject = new ActiveXObject("Scripting.FileSystemObject");
// The following three lines are not needed.
//    var e : Enumerator = new Enumerator(fso.Drives);
//    for (;!e.atEnd();e.moveNext()) {
//       x = e.item();
// Access the members of the enumeration directly.
for (x in fso.Drives) {               // Loop over the drives collection.
   if (x.DriveType == 3)              // See if network drive.
      n = x.ShareName;                // Get share name
   else if (x.IsReady)                // See if drive is ready.
      n = x.VolumeName;               // Get volume name.
   else
      n = "[Drive not ready]";
   print(x.DriveLetter + " - " + n);
}
```

Properties

The **Enumerator** object has no properties.

Methods

Enumerator Object Methods

Requirements

Version 3

See Also

new Operator | for...in Statement

Enumerator Object Methods

The Enumerator object enables enumeration of items in a collection.

Methods

atEnd Method

item Method

moveFirst Method

moveNext Method

See Also

Methods | JScript Reference

Error Object

Contains information about errors. There are two forms of the **Error** constructor.

```
function Error([description : String ])
function Error([number : Number [, description : String ]])
```

Arguments

number
Optional. Numeric value assigned to the error, specifying the value of the **number** property. Zero if omitted.

description
Optional. Brief string that describes the error, specifying the initial value of the **description** and **message** properties. Empty string if omitted.

Remarks

Error objects can be explicitly created using the constructor shown above. You can add properties to the **Error** object to expand its capabilities. An **Error** object is also created whenever a run-time error occurs to describe the error.

Typically, an Error object is thrown with the **throw** statement and the expectation that it will be caught by a **try...catch** statement. You can use a **throw** statement to pass any type of data as an error; the **throw** statement will not implicitly create an **Error** object. However, by throwing an **Error** object, a **catch** block can treat JScript run-time errors and user-defined errors similarly.

The **Error** object has four intrinsic properties: the description of the error (**description** and **message** properties), the error number (**number** property), and the name of the error (**name** property). The **description** and **message** properties refer to the same message; the **description** property provides backwards compatibility, while the **message** property complies with the ECMA standard.

An error number is a 32-bit value. The upper 16-bit word is the facility code, while the lower word is the actual error code. To read off the actual error code, use the **&** (bitwise And) operator to combine the number property with the hexadecimal number `0xFFFF`.

> **Caution** Attempting to use the JScript **Error** object in an ASP.NET page may product unintended results. This results from the potential ambiguity between the JScript **Error** object and the **Error** event of the ASP.NET page. Use the **System.Exception** class instead of the **Error** object for error handling in ASP.NET pages.

> **Note** Only JScript .NET provides the **Error** object. Since it is not derived from a .NET Framework type, other Common Language Specification (CLS) languages cannot use it. Consequently, when type-annotating the parameters and return types of CLS-compliant methods, make sure to use the **System.Exception** data type instead of the **Error** object. However, you may use the **Error** object to type annotate identifiers other than parameters or return types. For more information, see Writing CLS-Compliant Code.

Example

The following example illustrates a use of the **Error** object.

```
try {
   // Throw an error.
   throw new Error(42,"No question");
} catch(e) {
   print(e)
// Extract the error code from the error number.
   print(e.number & 0xFFFF)
   print(e.description)
}
```

The output of this code is:

```
Error: No question
42
No question
```

Properties and Methods

Error Object Properties and Methods

Requirements

Version 5

See Also

new Operator | throw Statement | try...catch...finally Statement | var Statement | System.Web.UI.Page Members

Error Object Properties and Methods

The Error object contains information about errors.

Properties

description Property

message Property

name Property

number Property

Methods

toString Method

See Also

Properties | Methods | JScript Reference

Function Object

Creates a new function.

```
function Function( [[param1 : String, [..., paramN : String,]] body : String ])
```

Arguments

param1, ..., paramN
Optional. The parameters of the function. Each parameter may have type annotation. The last parameter may be a *parameter array*, which is denoted by three periods (**...**) followed by a parameter array name and a typed array type annotation.

body
Optional. A string that contains the block of JScript code to be executed when the function is called.

Remarks

The **Function** constructor allows a script to create functions at run time. The parameters passed to the **Function** constructor (all but the last parameter) are used as the parameters of the new function. The last parameter passed to the constructor is interpreted as the code for the body of the function.

JScript compiles the object created by the **Function** constructor at the time the constructor is called. Although this allows your script to have great flexibility in redefining functions at run time, it is also makes the code much slower. Use the **Function** constructor as little as possible to avoid slow scripts.

When calling a function to evaluate, always include the parentheses and required arguments. Calling a function without parentheses returns the **Function** object for that function. The text of a function can be obtained by using the **toString** method of the **Function** object.

> **Note** Only JScript .NET provides the **Function** object. Since it is not derived from a .NET Framework type, other Common Language Specification (CLS) languages cannot use it. Consequently, when type-annotating the parameters and return types of CLS-compliant methods, make sure to use the **System.EventHandler** data type instead of the **Function** object. However, you may use the **Function** object to type annotate identifiers other than parameters or return types. For more information, see Writing CLS-Compliant Code.

Example

The following example illustrates a use of the **Function** object.

```
var add : Function = new Function("x", "y", "return(x+y)");
print(add(2, 3));
```

This code outputs:

```
5
```

Properties and Methods

Function Object Properties and Methods

Requirements

Version 2

See Also

function Statement | new Operator | var Statement

Function Object Properties and Methods

The Function object creates a new function.

Properties

0...n Properties

arguments Property

callee Property

caller Property

constructor Property

length Property (Function)

prototype Property

Methods

apply Method

call Method

toString Method

valueOf Method

See Also

Properties | Methods | JScript Reference

Global Object

An intrinsic object whose purpose is to collect global methods into one object.

The **Global** object has no syntax. You call its methods directly.

Remarks

The **Global** object is never used directly and cannot be created using the **new** operator. It is created when the scripting engine is initialized, thus making its methods and properties available immediately.

Properties and Methods

Global Object Properties and Methods

Requirements

Version 5

See Also

Object Object

Global Object Properties and Methods

The Global object is an intrinsic object whose purpose is to collect global methods into one object.

Properties

Infinity Property

NaN Property (Global)

undefined Property

Methods

decodeURI Method

decodeURIComponent Method

encodeURI Method

encodeURIComponent Method

escape Method

eval Method

isFinite Method

isNaN Method

parseFloat Method

parseInt Method

unescape Method

See Also

Properties | Methods | JScript Reference

Math Object

An intrinsic object that provides basic mathematics functionality and constants. This object cannot be constructed explicitly.

Remarks

The **new** operator cannot create the **Math** object and returns an error if you attempt to do so. The scripting engine creates the **Math** object when the engine is loaded. All of its methods and properties are available to a script at all times.

Example

The following example illustrates a use of the **Math** object. Note that since floating-point numbers have limited precision, calculations involving them can accumulate small rounding errors. You can use the **toFixed** method of the **Number** object to display numbers without small rounding errors.

```
var pi : double = Math.PI;         // Should be about 3.14.
print(pi);
var cosPi : double = Math.cos(pi); // Should be minus one.
print(cosPi);
var sinPi : double = Math.sin(pi); // Should be zero.
print(sinPi.toFixed(10));
```

The output of this code is:

```
3.141592653589793
-1
0.0000000000
```

Properties and Methods

Math Object Properties and Methods

Requirements

Version 1

See Also

Number Object

Math Object Properties and Methods

The Math object is an intrinsic object that provides basic mathematics functionality and constants.

Properties

E Property

LN10 Property

LN2 Property

LOG10E Property

LOG2E Property

PI Property

SQRT1_2 Property

SQRT2 Property

Methods

abs Method

acos Method

asin Method

atan Method

atan2 Method

ceil Method

cos Method

exp Method

floor Method

log Method

max Method

min Method

pow Method

random Method

round Method

sin Method

sqrt Method

tan Method

See Also

Properties | Methods | JScript Reference

Number Object

An object representation of numeric data and placeholder for numeric constants.

```
function Number( [value : Number] )
```

Arguments

value

Required. The numeric value of the **Number** object being created.

Remarks

The **Number** object is a wrapper for numeric data. The primary purposes for the **Number** object are to collect its properties into one object and to allow numbers to be converted into strings via the **toString** method. The **Number** object is similar to the **Number** data type. However, they have different properties and methods.

> **Note** You rarely need to construct a **Number** object explicitly. The **Number** data type should be used in most circumstances. Since the **Number** object interoperates with the **Number** data type, all **Number** object methods and properties are available to a variable of type **Number**. For more information, see Number Data Type.

The **Number** object is stores numeric data as an eight-byte, double-precision, floating-point number. It represents a double-precision 64-bit IEEE 754 value. The **Number** object can represent numbers in the range negative 1.79769313486231570E+308 through positive 1.79769313486231570E+308, inclusive. The smallest number that can be represented is 4.94065645841247E-324. The **Number** object can also represent **NaN** (Not a Number), positive and negative infinity, and positive and negative zero.

The data type of a **Number** object is **Object**, not **Number**.

Properties and Methods

Number Object Properties and Methods

Requirements

Version 1

See Also

Object Object | Number Data Type | Math Object | new Operator

Number Object Properties and Methods

The Number object is an object representation of the number data type and placeholder for numeric constants.

Properties

constructor Property

MAX_VALUE Property

MIN_VALUE Property

NaN Property

NEGATIVE_INFINITY Property

POSITIVE_INFINITY Property

prototype Property

Methods

toExponential Method

toFixed Method

toLocaleString Method

toPrecision Method

toString Method

valueOf Method

See Also

Properties | Methods | JScript Reference

Object Object

Provides functionality common to all JScript objects.

```
function Object([value : { ActiveXObject | Array | Boolean | Date | Enumerator | Error | Func
tion | Number | Object | RegExp | String | VBArray ])
```

Arguments

value

Optional. Any one of the JScript primitive data types. If value is an object, the object is returned unmodified. If *value* is **null**, **undefined**, or not supplied, an object with no content is created.

Remarks

The **Object** object forms the basis of all other JScript objects; all of its methods and properties are available in all other objects. The methods, which can be redefined in user-defined objects, are called by JScript at appropriate times. The **toString** method is an example of a frequently redefined **Object** method.

Variables defined without a type annotation are implicitly of type **Object**. Each JScript object has, in addition to its own properties and methods, all the properties and methods of the **Object** object.

Properties and Methods

Object object Properties and Methods

Requirements

Version 3

See Also

new Operator | Function Object | Global Object

Object Object Properties and Methods

The Object object provides functionality common to all JScript objects.

Properties

constructor Property

prototype Property

propertyIsEnumerable Property

Methods

isPrototypeOf Method

hasOwnProperty Method

toLocaleString Method

toString Method

valueOf Method

See Also

Properties | Methods | JScript Reference

RegExp Object

An intrinsic global object that stores information about the results of regular expression pattern matches. This object cannot be constructed explicitly.

Remarks

The **RegExp** object cannot be created directly, but it is always available. Until a successful regular expression search has been completed, the initial values of the various properties of the RegExp object are as follows:

Property	Shorthand	Initial Value
index		-1
input	$_	Empty string
lastIndex		-1
lastMatch	$&	Empty string.
lastParen	$+	Empty string.
leftContext	$`	Empty string.
rightContext	$'	Empty string.
$1 - $9		Empty string.

The global **RegExp** object should not be confused with the **Regular Expression** object. Although they sound similar, they are separate and distinct. The properties of the global **RegExp** object contain continually updated information about each match as it occurs, while the properties of the **Regular Expression** object contain only information about the matches that occur with a single instance of the **Regular Expression**.

> **Note** The properties of **RegExp** are not available when running in fast mode, the default for JScript .NET. To compile a program from the command line that uses these properties, you must turn off the fast option by using **/fast-**. It is not safe to turn off the fast option in ASP.NET because of threading issues.

Example

The following example illustrates the use of the global **RegExp** object. This example must be compiled with the **/fast-** option.

```
var re : RegExp = new RegExp("d(b+)(d)","ig");
var arr : Array = re.exec("cdbBdbsbdbdz");
print("$1 contains: " + RegExp.$1);
print("$2 contains: " + RegExp.$2);
print("$3 contains: " + RegExp.$3);
```

The output from this code is:

```
$1 contains: bB
$2 contains: d
$3 contains:
```

Properties

RegExp Object Properties

Methods

The **RegExp** object has no methods.

Requirements

Version 3

See Also

Regular Expression Object | Regular Expression Syntax | String Object | /fast

RegExp Object Properties

The RegExp Object is an intrinsic global object that stores information about the results of regular expression pattern matches.

Properties

$1...$9 Properties

index Property

input Property ($_)

lastIndex Property

lastMatch Property ($&)

lastParen Property ($+)

leftContext Property ($`)

rightContext Property ($')

See Also

Properties | JScript Reference

Regular Expression Object

An object that contains a regular expression pattern along with flags that identify how to apply the pattern.

Syntax 1

The explicit constructor for a Regular Expression object.

```
function RegExp(pattern : String [,flags : String])
function RegExp(regexObj : System.Text.RegularExpressions.Regex)
```

Syntax 2

The implicit constructor for a Regular Expression object.

```
/pattern/[flags]
```

Arguments

pattern
Required. The regular expression pattern to use. If you use Syntax 1, the pattern must be a string. If you use Syntax 2, the pattern is delimited by the "/" characters.

flags
Optional. If you use Syntax 1, the flags must be in a string. If you use Syntax 2, the flags characters immediately follow the last "/" character. Available flags, which may be combined, are:

- g (global search for all occurrences of *pattern*)
- i (ignore case)
- m (multiline search)

regexObj
Required. A **Regex** object that contains the regular expression pattern to use.

Remarks

The Regular Expression object should not be confused with the global **RegExp** object. Although they sound similar, they are easily distinguishable. The properties of the Regular Expression object contain information about only one particular Regular Expression instance, while the properties of the global **RegExp** object contain continually updated information about each match as it occurs.

Regular Expression objects store patterns used to search strings for character combinations. After the Regular Expression object is created, it is either passed to a string method, or a string is passed a method of the Regular Expression object. Information about the most recent search performed is stored in the global **RegExp** object.

Use Syntax 1 when the search string changes frequently or is unknown, such as strings derived from user input. Use Syntax 2 when you know the search string ahead of time.

In JScript the *pattern* argument is compiled into an internal format before use. For Syntax 1, *pattern* is compiled just before use or when the **compile** method is called. For Syntax 2, *pattern* is compiled as the script loads.

> **Note** The Regular Expression object interoperates with the .NET Framework **System.Text.RegularExpressions.Regex** data type within JScript .NET. However, other Common Language Specification (CLS) languages cannot use the Regular Expression object because only JScript .NET provides the object; it is not derived from a .NET Framework type. Consequently, when type-annotating the parameters and return types of CLS-compliant methods, make sure to use the **System.Text.RegularExpressions.Regex** data type instead of the Regular Expression object. However, you may use the Regular Expression object to type annotate identifiers other than the parameters or return types. For more information, see Writing CLS-Compliant Code.

Example

The following example illustrates the use of the Regular Expression object. Objects `re1` and `re2` are created and contain regular expression patterns with the associated flags. In this case, the resulting Regular Expression objects are then used by the **match** method:

```
var s : String = "The rain in Spain falls mainly in the plain";
// Create regular expression object using Syntax 1.
var re1 : RegExp = new RegExp("Spain","i");
// Create regular expression object using Syntax 2.
var re2 : RegExp = /IN/i;

// Find a match within string s.
print(s.match(re1));
print(s.match(re2));
```

The output from this script is

```
Spain
in
```

Properties and Methods

Regular Expression Object Properties and Methods

Requirements

Version 3

See Also

new Operator | RegExp Object | Regular Expression Syntax | String Object | Regex Class

Regular Expression Object Properties and Methods

The Regular Expression object is an object that contains a regular expression pattern along with flags that identify how to apply the pattern.

Properties

global Property

ignoreCase Property

multiline Property

source Property

Methods

compile Method

exec Method

test Method

See Also

Properties | Methods | JScript Reference

String Object

Allows manipulation and formatting of text strings and determines and locates substrings within strings.

```
function String([stringLiteral : String])
```

Arguments

stringLiteral
Optional. Any group of Unicode characters.

Remarks

String objects can be created implicitly using string literals. **String** objects created in this fashion (referred to as "primitive" strings) are treated differently from **String** objects created using the **new** operator. Although you can read properties and call methods on primitive strings, you cannot create new properties or add methods to them.

Escape sequences can be used in string literals to represent special characters that cannot be used directly in a string, such as the newline character or Unicode characters. At the time a script is compiled, each escape sequence in a string literal is converted into the characters it represents. For additional information, see String Data.

JScript also defines a **String** data type, which provides different properties and methods from the **String** object. You cannot create properties or add methods to variables of the **String** data type, while you can for instances of the **String** object.

The **String** object interoperates with **String** data type (which is the same as the **System.String** data type). This means that a **String** object can call the methods and properties of the **String** data type, and a **String** data type can call the methods and properties of the **String** object. For more information, see String Members. Furthermore, **String** objects are accepted by functions that take **String** data types, and vice versa.

The data type of a **String** object is **Object**, not **String**.

Example 1

This script demonstrates that although the length property can be read and the **toUpperCase** method can be called, the custom property `myProperty` cannot be set on the primitive string:

```
var primStr : Object = "This is a string";
print(primStr.length);            // Read the length property.
print(primStr.toUpperCase());     // Call a method.
primStr.myProperty = 42;          // Set a new property.
print(primStr.myProperty);        // Try to read it back.
```

The output of this script is:

```
16
THIS IS A STRING
undefined
```

Example 2

For **String** objects created with the **new** statement, custom properties can be set:

```
var newStr : Object = new String("This is also a string");
print(newStr.length);            // Read the length property.
print(newStr.toUpperCase());     // Call a method.
newStr.myProperty = 42;          // Set a new property.
print(newStr.myProperty);        // Try to read it back.
```

The output of this script is:

```
21
THIS IS ALSO A STRING
42
```

Properties and Methods

String Object Properties and Methods

Requirements

Version 1

See Also

Object Object | String Data Type | new Operator | String Data

String Object Properties and Methods

The String object allows manipulation and formatting of text strings and determination and location of substrings within strings.

Properties

constructor Property

length Property (String)

prototype Property

Methods

anchor Method

big Method

blink Method

bold Method

charAt Method

charCodeAt Method

concat Method (String)

fixed Method

fontcolor Method

fontsize Method

fromCharCode Method

indexOf Method

italics Method

lastIndexOf Method

link Method

localeCompare Method

match Method

replace Method

search Method

slice Method (String)

small Method

split Method

strike Method

sub Method

substr Method

substring Method

sup Method

toLocaleLowerCase Method

toLocaleUpperCase Method

toLowerCase Method

toString Method

toUpperCase Method

valueOf Method

See Also

Properties | Methods | JScript Reference

VBArray Object

Provides access to Visual Basic safe arrays.

```
varName = new VBArray(safeArray)
```

Arguments

varName
Required. The variable name to which the VBArray is assigned.
safeArray
Required. A VBArray value.

Remarks

The *safeArray* argument must have a VBArray value before being passed to the VBArray constructor. This can be acquired by retrieving the value from an existing ActiveX or other object.

> **Note** Arrays created in JScript .NET and arrays created in Visual Basic .NET both interoperate with .NET Framework arrays. Consequently, the elements of an array created in Visual Basic are directly accessible in JScript .NET. The **VBArray** object is only provided for backwards compatibility. For more information on arrays, see Array Object, Dim Statement, and Array Members.

A VBArray can have multiple dimensions. The indices of each dimension can be different. The **dimensions** method retrieves the number of dimensions in the array; the **lbound** and **ubound** methods retrieve the range of indices used by each dimension.

Properties

The VBArray object has no properties.

Methods

VBArray Object Methods

Requirements

Version 3

See Also

new Operator | Array Object | Array Class

VBArray Object Methods

The VBArray object provides access to Visual Basic safe arrays.

Methods

dimensions Method

getItem Method

lbound Method

toArray Method

ubound Method

See Also

Methods | JScript Reference

Operators

JScript includes a number of operators that fall into the arithmetic, logical, bitwise, assignment, and miscellaneous categories. The following sections link to information that explains how to use the operators.

In This Section

Addition Assignment Operator (+=)
Adds two numbers or concatenates two strings, and assigns the result to the first argument.

Addition Operator (+)
Adds two numbers or concatenates two strings.

Assignment Operator (=)
Assigns a value to a variable.

Bitwise AND Assignment Operator (&=)
Performs a bitwise AND on two expressions, and assigns the result to the first argument.

Bitwise AND Operator (&)
Performs a bitwise AND on two expressions.

Bitwise Left Shift Operator (<<)
Shifts the bits of an expression to the left.

Bitwise NOT Operator (~)
Performs a bitwise NOT (negation) on an expression.

Bitwise OR Assignment Operator (|=)
Performs a bitwise OR on two expressions, and assigns the result to the first argument.

Bitwise OR Operator (|)
Performs a bitwise OR on two expressions.

Bitwise Right Shift Operator (>>)
Shifts the bits of an expression to the right, maintaining sign.

Bitwise XOR Assignment Operator (^=)
Performs a bitwise exclusive OR on two expressions, and assigns the result to the first argument.

Bitwise XOR Operator (^)
Performs a bitwise exclusive OR on two expressions.

Comma Operator (,)
Causes two expressions to be executed sequentially.

Comparison Operators
An assortment of operators (==, >, >=, ===, !=, <, <=, !==) that return a Boolean value indicating the result of a comparison

Conditional (Ternary) Operator (?:)
Chooses one of two statements to run depending on a condition.

delete Operator
Deletes a property from an object, or removes an element from an array.

Division Assignment Operator (/=)
Divides two numbers, returns a numeric result, and assigns the result to the first argument.

Division Operator (/)
Divides two numbers and returns a numeric result.

in Operator
Tests for the existence of a property in an object.

Increment (++) and Decrement (--) Operators
Increment operator (++) increments a variable by one; decrement operator (--) decrements a variable by one.

instanceof Operator
Returns a Boolean value that indicates whether or not an object is an instance of a particular class.

Left Shift Assignment Operator (<<=)
Shifts the bits of an expression to the left, and assigns the result to the first argument.

Logical AND Operator (&&)
Performs a logical conjunction on two expressions.

Logical NOT Operator (!)
Performs logical negation on an expression.

Logical OR Operator (||)
Performs a logical disjunction on two expressions.

Modulus Assignment Operator (%=)
Divides two numbers and assigns the remainder to the first argument.

Modulus Operator (%)
Divides two numbers and returns the remainder.

Multiplication Assignment Operator (*=)
Multiplies two numbers and assigns the result to the first argument.

Multiplication Operator (*)
Multiplies two numbers.

new Operator
Creates a new object.

Right Shift Assignment Operator (>>=)
Shifts the bits of an expression to the right, maintaining sign, and assigns the result to the first argument.

Subtraction Assignment Operator (-=)
Subtracts one number from another and assigns the result to the first argument.

Subtraction Operator (-)
Indicates the negative value of a numeric expression or subtracts one number from another.

typeof Operator
Returns a string that identifies the data type of an expression.

Unsigned Right Shift Assignment Operator (>>>=)
Performs an unsigned right shift of the bits in an expression and assigns the result to the first argument.

Unsigned Right Shift Operator (>>>)
Performs an unsigned right shift of the bits in an expression.

void Operator
Prevents an expression from returning a value.

Related Sections

JScript Reference
Lists elements that comprise JScript Language Reference and links to topics that explain the details behind the proper use of language elements.

JScript Operators
Provides a conceptual overview of operators used in JScript and links to topics that explain the proper syntax for each operator and the significance of operator precedence.

Operator Precedence
Provides a list containing information about the execution precedence of JScript operators.

Operator Summary
Lists JScript operators and links to topics that explain their proper use.

Addition Assignment Operator (+=)

Adds the value of an expression to the value of a variable and assigns the result to the variable.

```
result += expression
```

Arguments

result
Any variable.
expression
Any expression.

Remarks

Using this operator is almost the same as specifying `result = result + expression`, except that *result* is only evaluated once.

The type of the expressions determines the behavior of the **+=** operator.

Result	***expression***	**Then**
char	**char**	Error
char	Numeric	Add
char	**String**	Error
Numeric	**char**	Add
Numeric	Numeric	Add
Numeric	**String**	Concatenate
String	**char**	Concatenate
String	Numeric	Concatenate
String	**String**	Concatenate

For concatenation, numbers are coerced into a string representation of the numeric value, and chars are considered to be strings of length 1. For addition of a char and a number, the char is coerced into a numeric value, and the two numbers are added. Some combinations of types give errors because the resulting type of the addition cannot be coerced into the required output type.

Example

The following example illustrates how the addition assignment operator processes expressions of different types.

```
var str : String = "42";
var n : int = 20;
var c : char = "A";  // The numeric value of "A" is 65.
var result;
c += n;         // The result is the char "U".
n += c;         // The result is the number 105.
n += n;         // The result is the number 210.
n += str;       // The result is the number 21042.
str += c;       // The result is the string "42U".
str += n;       // The result is the string "42U21042".
str += str;     // The result is the string "42U2104242U21042".
c += c;         // This returns a runtime error.
c += str;       // This returns a runtime error.
n += "string";  // This returns a runtime error.
```

Requirements

Version 1

See Also

Addition Operator (+) | Assignment Operator (=) | Operator Precedence | Operator Summary

Addition Operator (+)

Adds the value of one numeric expression to another or concatenates two strings.

```
expression1 + expression2
```

Arguments

expression1
Any expression.
expression2
Any expression.

Remarks

The type of the expressions determines the behavior of the **+** operator.

If	Then	Result Type
Both expressions are char	Concatenate	**String**
Both expressions are numeric	Add	numeric
Both expressions are strings	Concatenate	**String**
One expression is char and the other is numeric	Add	**char**
One expression is char and the other is a string	Concatenate	**String**
One expression is numeric and the other is a string	Concatenate	**String**

For concatenation, numbers are coerced into a string representation of the numeric value, and chars are considered to be strings of length 1. For addition of a char and a number, the char is coerced into a numeric value, and the two numbers are added.

> **Note** In scenarios where type annotation is not used, numeric data may be stored as a strings. Use explicit type conversion or type annotate variables to ensure that the addition operator does not treat numbers as strings, or vice versa.

Example

The following example illustrates how the addition operator processes expressions of different types.

```
var str : String = "42";
var n : double = 20;
var c : char = "A";  // the numeric value of "A" is 65
var result;
result = str + str;  // result is the string "4242"
result = n + n;      // result is the number 40
result = c + c;      // result is the string "AA"
result = c + n;      // result is the char "U"
result = c + str;    // result is the string "A42"
result = n + str;    // result is the string "2042"
// Use explicit type coversion to use numbers as strings, or vice versa.
result = int(str) + int(str);    // result is the number 84
result = String(n) + String(n);  // result is the string "2020"
result = c + int(str);           // result is the char "k"
```

Requirements

Version 1

See Also

Addition Assignment Operator (+=) | Operator Precedence | Operator Summary | Type Conversion

Assignment Operator (=)

Assigns a value to a variable.

```
result = expression
```

Arguments

result
Any variable.

expression
Any expression.

Remarks

The **=** operator returns the value of *expression* and assigns that value into *variable*. This means that you can chain assignment operators as follows:

```
j = k = l = 0;
```

`j`, `k`, and `l` equal zero after the example statement is executed.

The data type of *expression* must be coercible to the data type of *result*.

Requirements

Version 1

See Also

Operator Precedence | Operator Summary

Bitwise Left Shift Operator (<<)

Left shifts the bits of an expression.

```
expression1 << expression2
```

Arguments

expression1
  Any numeric expression.
expression2
  Any numeric expression.

Remarks

The **<<** operator shifts the bits of *expression1* left by the number of bits specified in *expression2*. The data type of *expression1* determines the data type returned by this operator.

The **<<** operator masks *expression2* to avoid shifting *expression1* by too much. Otherwise, if the shift amount exceeded the number of bits in the data type of *expression1*, all the original bits would be shifted away to give a trivial result. To ensure that each shift leaves at least one of the original bits, the shift operators use the following formula to calculate the actual shift amount: mask *expression2* (using the bitwise AND operator) with one less than the number of bits in *expression1*.

Example

For example:

```
var temp
temp = 14 << 2
```

The variable *temp* has a value of 56 because 14 (00001110 in binary) shifted left two bits equals 56 (00111000 in binary).

To illustrate how the masking works, consider the following example.

```
var x : byte = 15;
// A byte stores 8 bits.
// The bits stored in x are 00001111
var y : byte = x << 10;
// Actual shift is 10 & (8-1) = 2
// The bits stored in y are 00111100
// The value of y is 60
print(y); // Prints 60
```

Requirements

Version 1

See Also

Left Shift Assignment Operator (<<=)| Bitwise Right Shift Operator (>>) | Unsigned Right Shift Operator (>>>) | Operator Precedence | Operator Summary | Coercion By Bitwise Operators

Bitwise NOT Operator (~)

Performs a bitwise NOT (negation) on an expression.

```
~ expression
```

Arguments

expression
Any numeric expression.

Remarks

The ~ operator looks at the binary representation of the values of the expression and does a bitwise negation operation on it. The result of this operation behaves as follows:

```
0101   (expression)
----
1010   (result)
```

Any digit that is a 1 in the expression becomes a 0 in the result. Any digit that is a 0 in the expression becomes a 1 in the result.

When the ~ operator acts on an operand of an integral data type, it performs no coercion and returns a value of the same data type as the operand. When the operand is of a non-integral data type, the value is coerced to type **int** before the operation is performed, and the return value of the operator is of type **int**.

Requirements

Version 1

See Also

Logical NOT Operator (!) | Operator Precedence | Operator Summary

Bitwise OR Assignment Operator (|=)

Performs a bitwise OR on the value of a variable and the value of an expression and assigns the result to the variable.

```
result |= expression
```

Arguments

result
Any numeric variable.
expression
Any numeric expression.

Remarks

Using this operator is almost the same as specifying `result = result | expression`, except that *result* is only evaluated once.

The **|=** operator coerces the arguments to matching data types. Then the **|=** operator looks at the binary representation of the values of *result* and *expression* and does a bitwise OR operation on them. The result of this operation behaves like this:

```
0101    (result)
1100    (expression)
----
1101    (output)
```

Any time either of the expressions has a 1 in a digit, the result has a 1 in that digit. Otherwise, the result has a 0 in that digit.

Requirements

Version 1

See Also

Bitwise OR Operator (|) | Assignment Operator (=) | Operator Precedence | Operator Summary | Coercion By Bitwise Operators

Bitwise OR Operator (|)

Performs a bitwise OR on two expressions.

```
expression1 | expression2
```

Arguments

expression1
  Any numeric expression.
expression2
  Any numeric expression.

Remarks

The **|** operator coerces the arguments to matching data types. Then the **|** operator looks at the binary representation of the values of two expressions and does a bitwise OR operation on them. The data types of the arguments determine the data type returned by this operator.

The result of this operation behaves as follows:

```
0101   (expression1)
1100   (expression2)
----
1101   (result)
```

Any time either of the expressions has a 1 in a digit, the result will have a 1 in that digit. Otherwise, the result will have a 0 in that digit.

Requirements

Version 1

See Also

Bitwise OR Assignment Operator (|=) | Operator Precedence | Operator Summary | Coercion By Bitwise Operators

Bitwise Right Shift Operator (>>)

Right shifts the bits of an expression, maintaining sign.

```
expression1 >> expression2
```

Arguments

expression1
  Any numeric expression.
expression2
  Any numeric expression.

Remarks

The **>>** operator shifts the bits of *expression1* right by the number of bits specified in *expression2*. The sign bit of *expression1* is used to fill the digits from the left. Digits shifted off to the right are discarded. The data type of *expression1* determines the data type returned by this operator.

The **>>** operator masks *expression2* to avoid shifting *expression1* by too much. Otherwise, if the shift amount exceeded the number of bits in the data type of *expression1*, all the original bits would be shifted away to give a trivial result. To ensure that each shift leaves at least one of the original bits, the shift operators use the following formula to calculate the actual shift amount: mask *expression2* (using the bitwise AND operator) with one less than the number of bits in *expression1*.

Example

For example, after the following code is evaluated, *temp* has a value of -4: -14 (11110010 in binary) shifted right two bits equals -4 (11111100 in binary).

```
var temp
temp = -14 >> 2
```

To illustrate how the masking works, consider the following example.

```
var x : byte = 15;
// A byte stores 8 bits.
// The bits stored in x are 00001111
var y : byte = x >> 10;
// Actual shift is 10 & (8-1) = 2
// The bits stored in y are 00000011
// The value of y is 3
print(y); // Prints 3
```

Requirements

Version 1

See Also

Bitwise Left Shift Operator (<<) | Right Shift Assignment Operator (>>=) | Unsigned Right Shift Operator (>>>) | Operator Precedence | Operator Summary | Coercion By Bitwise Operators

Bitwise XOR Assignment Operator (^=)

Performs a bitwise exclusive OR on a variable and an expression and assigns the result to the variable.

```
result ^= expression
```

Arguments

result
Any numeric variable.
expression
Any numeric expression.

Remarks

Using this operator is almost the same as specifying `result = result ^ expression`, except that *result* is only evaluated once.

The **^=** operator coerces the arguments to matching data types. Then the **^=** operator looks at the binary representation of the values of two expressions and does a bitwise exclusive OR operation on them. The result of this operation behaves as follows:

```
0101    (result)
1100    (expression)
----
1001    (result)
```

When one, and only one, of the expressions has a 1 in a digit, the result has a 1 in that digit. Otherwise, the result has a 0 in that digit.

Requirements

Version 1

See Also

Bitwise XOR Operator (^) | Assignment Operator (=) | Operator Precedence | Operator Summary | Coercion By Bitwise Operators

Bitwise XOR Operator (^)

Performs a bitwise exclusive OR on two expressions.

```
expression1 ^ expression2
```

Arguments

expression1
  Any numeric expression.
expression2
  Any numeric expression.

Remarks

The **^** operator coerces the arguments to matching data types. Then the **^** operator looks at the binary representation of the values of two expressions and does a bitwise exclusive OR operation on them. The data types of the arguments determine the data type returned by this operator.

The result of this operation behaves as follows:

```
0101   (expression1)
1100   (expression2)
----
1001   (result)
```

When one, and only one, of the expressions has a 1 in a digit, the result has a 1 in that digit. Otherwise, the result has a 0 in that digit.

Requirements

Version 1

See Also

Bitwise XOR Assignment Operator (^=) | Operator Precedence | Operator Summary | Coercion By Bitwise Operators

Comma Operator (,)

Causes two expressions to be executed sequentially.

```
expression1, expression2
```

Arguments

expression1
  Any expression.
expression2
  Any expression.

Remarks

The **,** operator causes the expressions on either side of it to be executed in left-to-right order, and obtains the value of the expression on the right. The most common use for the **,** operator is in the increment expression of a **for** loop. For example:

```
var i, j, k;
for (i = 0; i < 10; i++, j++) {
   k = i + j;
}
```

The **for** statement only allows a single expression to be executed at the end of every pass through a loop. The **,** operator is used to allow multiple expressions to be treated as a single expression, thereby getting around the restriction.

Requirements

Version 1

See Also

for Statement | Operator Precedence | Operator Summary

Comparison Operators

Returns a Boolean value indicating the result of the comparison.

```
expression1 comparisonoperator expression2
```

Arguments

expression1
Any expression.
comparisonoperator
Any comparison operator (<, >, <=, >=, ==, !=, ===, !==)
expression2
Any expression.

Remarks

When comparing strings, JScript uses the Unicode character value of the string expression.

The following describes how the different groups of operators behave depending on the types and values of *expression1* and *expression2*:

Relational (<, >, <=, >=)

- Attempt to convert both *expression1* and *expression2* into numbers.
- If both expressions are strings, do a lexicographical string comparison.
- If either expression is **NaN**, return **false**.
- Negative zero equals Positive zero.
- Negative Infinity is less than everything including itself.
- Positive Infinity is greater than everything including itself.

Equality (==, !=)

- If the types of the two expressions are different, attempt to convert them to string, number, or Boolean.
- **NaN** is not equal to anything including itself.
- Negative zero equals positive zero.
- **null** equals both **null** and **undefined**.
- Values are considered equal if they are identical strings, numerically equivalent numbers, the same object, identical Boolean values, or (if different types) they can be coerced into one of these situations.
- Every other comparison is considered unequal.

Identity (===, !==)

These operators behave identically to the equality operators except no type conversion is done, and the types must be the same to be considered equal.

Requirements

Version 1

See Also

Operator Precedence | Operator Summary

Conditional (Ternary) Operator (?:)

Returns one of two expressions depending on a condition.

```
test ? expression1 : expression2
```

Arguments

test
Any Boolean expression.

expression1
An expression returned if *test* is **true**. May be a comma expression.

expression2
An expression returned if *test* is **false**. May be a comma expression.

Remarks

The **?:** operator can be used as a shortcut for an **if...else** statement. It is typically used as part of a larger expression where an **if...else** statement would be awkward. For example:

```
var now = new Date();
var greeting = "Good" + ((now.getHours() > 17) ? " evening." : " day.");
```

The example creates a string containing "Good evening." if it is after 6pm. The equivalent code using an **if...else** statement would look as follows:

```
var now = new Date();
var greeting = "Good";
if (now.getHours() > 17)
   greeting += " evening.";
else
   greeting += " day.";
```

Requirements

Version 1

See Also

if...else Statement | Operator Precedence | Operator Summary

delete Operator

Deletes a property from an object, removes an element from an array, or removes an entry from an IDictionary object.

```
delete expression
```

Arguments

expression
Required. Any expression that results in a property reference, array element, or IDictionary object.

Remarks

If the result of *expression* is an object, the property specified in *expression* exists, and the object will not allow it to be deleted, **false** is returned.

In all other cases, **true** is returned.

Example

The following example illustrates a use of the **delete** operator.

```
// Make an object with city names and an index letter.
var cities : Object = {"a" : "Athens" , "b" : "Belgrade", "c" : "Cairo"}

// List the elements in the object.
var key : String;
for (key in cities) {
   print(key + " is in cities, with value " + cities[key]);
}

print("Deleting property b");
delete cities.b;

// List the remaining elements in the object.
for (key in cities) {
   print(key + " is in cities, with value " + cities[key]);
}
```

The output of this code is:

```
a is in cities, with value Athens
b is in cities, with value Belgrade
c is in cities, with value Cairo
Deleting property b
a is in cities, with value Athens
c is in cities, with value Cairo
```

Requirements

Version 3

See Also

IDictionary Interface | Operator Precedence | Operator Summary

Division Assignment Operator (/=)

Divides the value of a variable by the value of an expression and assigns the result to the variable.

```
result /= expression
```

Arguments

result
Any numeric variable.
expression
Any numeric expression.

Remarks

Using this operator is almost the same as specifying `result = result / expression`, except that *result* is only evaluated once.

Requirements

Version 1

See Also

Division Operator (/) | Assignment Operator (=) | Operator Precedence | Operator Summary

Division Operator (/)

Divides the value of two expressions.

```
number1 / number2
```

Arguments

number1
  Any numeric expression.
number2
  Any numeric expression.

Remarks

If *number1* is a finite, non-zero number, and *number2* is zero, the result of the division is **Infinity** if the *number1* is positive, and **-Infinity** if negative. If both *number1* and *number2* are zero, the result is **NaN**.

Requirements

Version 1

See Also

Division Assignment Operator (/=) | Operator Precedence | Operator Summary

in Operator

Tests for the existence of a property in an object.

```
property in object
```

Arguments

property
Required. An expression that evaluates to a string.
object
Required. Any object.

Remarks

The **in** operator checks if an object has a property named *property*. It also checks the object's prototype to see if *property* is part of the prototype chain. If *property* is in the object or prototype chain, the **in** operator returns **true**, otherwise it returns **false**.

The **in** operator should not be confused with the **for...in** statement.

> **Note** To test if the object itself has a property, and does not inherit the property from the prototype chain, use the object's **hasOwnProperty** method.

Example

The following example illustrates a use of the **in** operator.

```
function cityName(key : String, cities : Object) : String {
   // Returns a city name associated with an index letter.
   var ret : String = "Key '" + key + "'";
   if( key in cities )
      return ret + " represents " + cities[key] + ".";
   else  // no city indexed by the key
      return ret + " does not represent a city."
}

// Make an object with city names and an index letter.
var cities : Object = {"a" : "Athens" , "b" : "Belgrade", "c" : "Cairo"}

// Look up cities with an index letter.
print(cityName("a",cities));
print(cityName("z",cities));
```

The output of this code is:

```
Key 'a' represents Athens.
Key 'z' does not represent a city.
```

Requirements

Version 1

See Also

Operator Precedence | Operator Summary | for...in Statement | hasOwnProperty Method

Increment (++) and Decrement (--) Operators

Increments or decrements the value of a variable by one.

Prefix Syntax

```
++variable
--variable
```

Postfix Syntax

```
variable++
variable--
```

Arguments

variable
Any numeric variable.

Remarks

The increment and decrement operators are used as a shortcut to modify the value stored in a variable and access that value. Either operator may be used in a prefix or postfix syntax.

If	Equivalent Action	Return value
++*variable*	*variable* += 1	value of *variable* after incrementing
*variable***++**	*variable* += 1	value of *variable* before incrementing
--*variable*	*variable* -= 1	value of *variable* after decrementing
*variable***--**	*variable* -= 1	value of *variable* before decrementing

Example

The following example illustrates the difference between prefix and postfix syntax for the **++** operator.

```
// Example of prefix increment operator
var j1 : int = 2;
var k1 : int;
k1 = ++j1;          // k1 is 3, the value of j1 after incrementing

// Example of postfix increment operator
var j2 : int = 2;
var k2 : int;
k2 = j2++;          // k2 is 2, the value of j2 before incrementing
```

Requirements

Version 1

See Also

Operator Precedence | Operator Summary

instanceof Operator

Returns a Boolean value that indicates whether or not an object is an instance of a particular class or constructed function.

```
object instanceof class
```

Arguments

object
Required. Any object expression.
class
Required. Any object class or constructed function.

Remarks

The **instanceof** operator returns **true** if *object* is an instance of *class* or constructed function. It returns **false** if *object* is not an instance of the specified class or function, or if *object* is **null**.

The JScript **Object** is special. An object is only considered an instance of **Object** if and only if the object was constructed with the **Object** constructor.

Example 1

The following example illustrates a use of the **instanceof** operator to check the type of a variable.

```
// This program uses System.DateTime, which must be imported.
import System

function isDate(ob) : String {
   if (ob instanceof Date)
      return "It's a JScript Date"
   if (ob instanceof DateTime)
      return "It's a .NET Framework Date"
   return "It's not a date"
}

var d1 : DateTime = DateTime.Now
var d2 : Date = new Date
print(isDate(d1))
print(isDate(d2))
```

The output of this code is:

```
It's a .NET Date
It's a JScript Date
```

Example 2

The following example illustrates a use of the **instanceof** operator to check instances of a constructed function.

```
function square(x : int) : int {
   return x*x
}

function bracket(s : String) : String{
   return("[" + s + "]");
}

var f = new square
print(f instanceof square)
print(f instanceof bracket)
```

The output of this code is:

```
true
false
```

Example 3

The following example illustrates how the **instanceof** operator checks if objects are instances of **Object**.

```
class CDerived extends Object {
   var x : double;
}

var f : CDerived = new CDerived;
var ob : Object = f;
print(ob instanceof Object);

ob = new Object;
print(ob instanceof Object);
```

The output of this code is:

```
false
true
```

Requirements

Version 5

See Also

Operator Precedence | Operator Summary

Left Shift Assignment Operator (<<=)

Left shifts the value of a variable by the number of bits specified in the value of an expression and assigns the result to the variable.

```
result <<= expression
```

Arguments

result
Any numeric variable.
expression
Any numeric expression.

Remarks

Using this operator is almost the same as specifying `result = result << expression`, except that *result* is only evaluated once.

The **<<=** operator shifts the bits of *result* left by the number of bits specified in *expression*. The operator masks *expression* to avoid shifting *result* by too much. Otherwise, if the shift amount exceeded the number of bits in the data type of *result*, all the original bits would be shifted away to give a trivial result. To ensure that each shift leaves at least one of the original bits, the shift operators use the following formula to calculate the actual shift amount: mask *expression* (using the bitwise AND operator) with one less than the number of bits in *result*.

Example

For example:

```
var temp
temp = 14
temp <<= 2
```

The variable *temp* has a value of 56 because 14 (00001110 in binary) shifted left two bits equals 56 (00111000 in binary). Bits are filled in with zeroes when shifting.

To illustrate how the masking works, consider the following example.

```
var x : byte = 15;
// A byte stores 8 bits.
// The bits stored in x are 00001111
x <<= 10;
// Actual shift is 10 & (8-1) = 2
// The bits stored in x are 00111100
// The value of x is 60
print(x); // Prints 60
```

Requirements

Version 1

See Also

Bitwise Left Shift Operator (<<) | Bitwise Right Shift Operator (>>) | Unsigned Right Shift Operator (>>>) | Assignment Operator (=) | Operator Precedence | Operator Summary

Logical NOT Operator (!)

Performs logical negation on an expression.

```
!expression
```

Arguments

expression
Any expression.

Remarks

The following table illustrates how the result is determined.

If `expression` coerces to	Then `result` is
true	**false**
false	**true**

All unary operators, such as the **!** operator, evaluate expressions as follows:

- If applied to undefined or **null** expressions, a run-time error is raised.
- Objects are converted to strings.
- Strings are converted to numbers if possible. If not, a run-time error is raised.
- Boolean values are treated as numbers (0 if **false**, 1 if **true**).

The operator is applied to the resulting number.

For the **!** operator, if *expression* is nonzero, *result* is zero. If *expression* is zero, *result* is 1.

Requirements

Version 1

See Also

Bitwise NOT Operator (~) | Operator Precedence | Operator Summary

Logical OR Operator (||)

Performs a logical disjunction on two expressions.

```
expression1 || expression2
```

Arguments

expression1
Any expression.
expression2
Any expression.

Remarks

If either or both expressions evaluate to **true**, the result is **true**. The following table illustrates how the result is determined:

If `expression1` coerces to	And `expression2` coerces to	The result is	The result coerces to
true	**true**	`expression1`	**true**
true	**false**	`expression1`	**true**
false	**true**	`expression2`	**true**
false	**false**	`expression2`	**false**

JScript uses the following rules for converting non-Boolean values to Boolean values:

- All objects are considered **true**.
- Strings are considered **false** if and only if they are empty.
- **null** and undefined are considered false.
- Numbers are **false** if, and only if, they are 0.

Requirements

Version 1

See Also

Operator Precedence | Operator Summary

Modulus Assignment Operator (%=)

Divides the value of a variable by the value of an expression, and assigns the remainder to the variable.

```
result %= expression
```

Arguments

result
Any numeric variable.
expression
Any numeric expression.

Remarks

Using this operator is almost the same as specifying `result = result % expression`, except that *result* is only evaluated once.

Requirements

Version 1

See Also

Modulus Operator (%) | Assignment Operator (=) | Operator Precedence | Operator Summary

Modulus Operator (%)

Divides the value of one expression by the value of another, and returns the remainder.

```
number1 % number2
```

Arguments

number1
Any numeric expression.
number2
Any numeric expression.

Remarks

The modulus, or remainder, operator divides *number1* by *number2* and returns only the remainder. The sign of the result is the same as the sign of *number1*. The value of the result is between 0 and the absolute value of *number2*.

The arguments to the modulus operator may be floating-point numbers, so that `5.6 % 0.5` returns `0.1`.

Example

The following example illustrates a use of the modulus operator.

```
var myMoney : int = 128;
var cookiePrice : int = 33;
// Calculate the change if the maximum number of cookies are bought.
var change : int = myMoney % cookiePrice;
// Calculate number of cookies bought.
var numCookies : int = Math.round((myMoney-change)/cookiePrice);
```

Requirements

Version 1

See Also

Modulus Assignment Operator (%=) | Operator Precedence | Operator Summary

Multiplication Assignment Operator (*=)

Multiplies the value of a variable by the value of an expression and assigns the result to the variable.

```
result *= expression
```

Arguments

result
Any numeric variable.
expression
Any numeric expression.

Remarks

Using this operator is almost the same as specifying `result = result * expression`, except that *result* is only evaluated once.

Requirements

Version 1

See Also

Multiplication Operator (*) | Assignment Operator (=) | Operator Precedence | Operator Summary

Multiplication Operator (*)

Multiplies the value of two expressions.

```
number1 * number2
```

Arguments

number1
Any numeric expression.

number2
Any numeric expression.

Remarks

The multiplication operator multiplies *number1* by *number2* and returns the result. If either *number* is **NaN**, the result is **NaN**. Multiplication of **Infinity** by zero gives a result of **NaN**, while multiplying **Infinity** by any non-zero number (including **Infinity**) gives a result of **Infinity**.

Requirements

Version 1

See Also

Multiplication Assignment Operator (*=) | Operator Precedence | Operator Summary

new Operator

Creates a new object.

```
new constructor[( [arguments] )]
```

Arguments

constructor
Required. Object's construction. The parentheses can be omitted if the constructor takes no arguments.

arguments
Optional. Any arguments to be passed to the new object's constructor.

Remarks

The **new** operator performs the following tasks:

- It creates an object with no members.
- It calls the constructor for that object, passing a reference to the newly created object as the **this** pointer.
- The constructor then initializes the object according to the arguments passed to the constructor.

Example

These examples demonstrate some uses of the **new** operator.

```
var myObject : Object = new Object;
var myArray : Array = new Array();
var myDate : Date = new Date("Jan 5 1996");
```

Requirements

Version 1

See Also

function Statement

Right Shift Assignment Operator (>>=)

Right shifts the value of a variable by the number of bits specified in the value of an expression, maintains the sign, and assigns the result to the variable.

```
result >>= expression
```

Arguments

result
Any numeric variable.
expression
Any numeric expression.

Remarks

Using this operator is almost the same as specifying `result = result >> expression`, except that *result* is only evaluated once.

The **>>=** operator shifts the bits of *result* right by the number of bits specified in *expression*. The sign bit of *result* is used to fill the digits from the left. Digits shifted off to the right are discarded. The operator masks *expression* to avoid shifting *result* by too much. Otherwise, if the shift amount exceeded the number of bits in the data type of *result*, all the original bits would be shifted away to give a trivial result. To ensure that each shift leaves at least one of the original bits, the shift operators use the following formula to calculate the actual shift amount: mask *expression* (using the bitwise AND operator) with one less than the number of bits in *result*.

Example

For example, after the following code is evaluated, *temp* has a value of -4: 14 (11110010 in binary) shifted right two bits equals -4 (11111100 in binary).

```
var temp
temp = -14
temp >>= 2
```

To illustrate how the masking works, consider the following example.

```
var x : byte = 15;
// A byte stores 8 bits.
// The bits stored in x are 00001111
x >>= 10;
// Actual shift is 10 & (8-1) = 2
// The bits stored in x are 00000011
// The value of x is 3
print(x); // Prints 3
```

Requirements

Version 1

See Also

Bitwise Left Shift Operator (<<) | Bitwise Right Shift Operator (>>) | Unsigned Right Shift Operator (>>>) | Assignment Operator (=) | Operator Precedence | Operator Summary | Coercion By Bitwise Operators

Subtraction Assignment Operator (-=)

Subtracts the value of an expression from the value of a variable and assigns the result to the variable.

```
result -= expression
```

Arguments

result
Any numeric variable.
expression
Any numeric expression.

Remarks

Using this operator is almost the same as specifying `result = result - expression`, except that *result* is only evaluated once.

Requirements

Version 1

See Also

Subtraction Operator (-) | Assignment Operator (=) | Operator Precedence | Operator Summary

Subtraction Operator (-)

Subtracts the value of one expression from another or provides unary negation of a single expression.

Syntax 1

```
number1 - number2
```

Syntax 2

```
-number
```

Arguments

number1
Any numeric expression.
number2
Any numeric expression.
number
Any numeric expression.

Remarks

In Syntax 1, the **-** operator is the arithmetic subtraction operator used to find the difference between two numbers. In Syntax 2, the **-** operator is used as the unary negation operator to indicate the negative value of an expression.

For Syntax 2, as for all unary operators, expressions are evaluated as follows:

- If applied to undefined or **null** expressions, a run-time error is raised.
- Objects are converted to strings.
- Strings are converted to numbers if possible. If not, a run-time error is raised.
- Boolean values are treated as numbers (0 if false, 1 if true).

The operator is applied to the resulting number. In Syntax 2, if the resulting number is nonzero, *result* is equal to the resulting number with its sign reversed. If the resulting number is zero, *result* is zero.

Requirements

Version 1

See Also

Subtraction Assignment Operator (-=) | Operator Precedence | Operator Summary

typeof Operator

Returns a string that identifies the data type of an expression.

```
typeof[(]expression[)] ;
```

Arguments

expression
Required. Any expression.

Remarks

The **typeof** operator returns type information as a string. There are six possible values that **typeof** returns: "number", "string", "boolean", "object", "function", and "undefined".

The parentheses are optional in the **typeof** syntax.

> **Note** : All expressions in JScript .NET have a **GetType** method. This method returns the data type (not a string representing the data type) of the expression. The **GetType** method provides more information than the **typeof** operator.

Example

The following example illustrates the use of the **typeof** operator.

```
var x : double = Math.PI;
var y : String = "Hello";
var z : int[] = new int[10];

print("The type of x (a double) is " + typeof x  );
print("The type of y (a String) is " + typeof(y) );
print("The type of z (an int[]) is " + typeof(z) );
```

The output of this code is:

```
The type of x (a double) is number

The type of y (a String) is string

The type of z (an int[]) is object
```

Requirements

Version 1

See Also

Operator Precedence | Operator Summary | GetType Method

Unsigned Right Shift Assignment Operator (>>>=)

Right shifts the value of a variable by the number of bits specified in the value of an expression, without maintaining sign, and assigns the result to the variable.

```
result >>>= expression
```

Arguments

result
Any numeric variable.
expression
Any numeric expression.

Remarks

Using this operator is almost the same as specifying `result = result >>> expression`, except that *result* is only evaluated once.

The **>>>=** operator shifts the bits of *result* right by the number of bits specified in *expression*. Zeroes are filled in from the left. Digits shifted off to the right are discarded. The operator masks *expression* to avoid shifting *result* by too much. Otherwise, if the shift amount exceeded the number of bits in the data type of *result*, all the original bits would be shifted away to give a trivial result. To ensure that each shift leaves at least one of the original bits, the shift operators use the following formula to calculate the actual shift amount: mask *expression* (using the bitwise AND operator) with one less than the number of bits in *result*.

Example

For example:

```
var temp
temp = -14
temp >>>= 2
```

The variable *temp* has a value of 1073741820 as -14 (11111111 11111111 11111111 11110010 in binary) shifted right two bits equals 1073741820 (00111111 11111111 11111111 11111100 in binary).

To illustrate how the masking works, consider the following example.

```
var x : byte = 15;
// A byte stores 8 bits.
// The bits stored in x are 00001111
x >>>= 10;
// Actual shift is 10 & (8-1) = 2
// The bits stored in x are 00000011
// The value of x is 3
print(x); // Prints 3
```

Requirements

Version 1

See Also

Unsigned Right Shift Operator (>>>) | Bitwise Left Shift Operator (<<) | Bitwise Right Shift Operator (>>) | Assignment Operator (=) | Operator Precedence | Operator Summary | Coercion By Bitwise Operators

Unsigned Right Shift Operator (>>>)

Right shifts the bits of an expression, without maintaining sign.

```
expression1 >>> expression2
```

Arguments

expression1
  Any numeric expression.
expression2
  Any numeric expression.

Remarks

The **>>>** operator shifts the bits of *expression1* right by the number of bits specified in *expression2*. Zeroes are filled in from the left. Digits shifted off to the right are discarded. The data type of *expression1* determines the data type returned by this operator.

The **>>>** operator masks *expression2* to avoid shifting *expression1* by too much. Otherwise, if the shift amount exceeded the number of bits in the data type of *expression1*, all the original bits would be shifted away to give a trivial result. To ensure that each shift leaves at least one of the original bits, the shift operators use the following formula to calculate the actual shift amount: mask *expression2* (using the bitwise AND operator) with one less than the number of bits in *expression1*.

Example

For example:

```
var temp
temp = -14 >>> 2
```

The variable *temp* has a value of 1073741820 as -14 (11111111 11111111 11111111 11110010 in binary) shifted right two bits equals 1073741820 (00111111 11111111 11111111 11111100 in binary).

To illustrate how the masking works, consider the following example.

```
var x : byte = 15;
// A byte stores 8 bits.
// The bits stored in x are 00001111
var y : byte = x >>> 10;
// Actual shift is 10 & (8-1) = 2
// The bits stored in y are 00000011
// The value of y is 3
print(y); // Prints 3
```

Requirements

Version 1

See Also

Unsigned Right Shift Assignment Operator (>>>=) | Bitwise Left Shift Operator (<<) | Bitwise Right Shift Operator (>>) | Operator Precedence | Operator Summary | Coercion By Bitwise Operators

void Operator

Prevents an expression from returning a value.

```
void expression
```

Arguments

expression
Required. Any expression.

Remarks

The **void** operator evaluates its expression, and returns undefined. It is most useful in situations where you want an expression evaluated but do not want the results visible to the remainder of the script.

Requirements

Version 2

See Also

Operator Precedence | Operator Summary

Properties

A property is a value or set of values (in the form of an array or object) that is a member of an object. The following sections link to information that explains how to use the properties in JScript.

In This Section

0...n Properties
Returns the actual value of individual arguments from an **arguments** object returned by the **arguments** property of an executing function.

$1...$9 Properties
Returns the nine most-recently memorized portions found during pattern matching.

arguments Property
Returns the arguments object for the currently executing Function object.caller Property

callee Property
Returns the **Function** object being executed that is the body text of the specified **Function** object.

caller Property
Returns a reference to the function that invoked the current function.

constructor Property
Specifies the function that creates an object.

description Property
Returns or sets the descriptive string associated with a specific error.

E Property
Returns the mathematical constant *e*, the base of natural logarithms.

global Property
Returns a Boolean value indicating the state of the global flag (**g**) used with a regular expression.

ignoreCase Property
Returns a Boolean value indicating the state of the ignoreCase flag (**i**) used with a regular expression.

index Property
Returns the character position where the first successful match begins in a searched string.

Infinity Property
Returns an initial value of **Number.POSITIVE_INFINITY**.

input Property ($_)
Returns the string against which a regular expression search was performed.

lastIndex Property
Returns the character position where the next match begins in a searched string.

lastMatch Property ($&)
Returns the last matched characters from any regular expression search.

lastParen Property ($+)
Returns the last parenthesized submatch, if any, from any regular expression search.

leftContext Property ($`)
Returns the characters from the beginning of a searched string up to the position before the beginning of the last match.

length Property (arguments)
Returns the actual number of arguments passed to a function by the caller.

length Property (Array)
Returns an integer value one higher than the highest element defined in an array.

length Property (Function)
Returns the number of arguments defined for a function.

length Property (String)
Returns the length of a **String** object.

LN10 Property
Returns the natural logarithm of 10.

LN2 Property
Returns the natural logarithm of 2.

LOG10E Property
Returns the base-10 logarithm of *e*, the base of natural logarithms.

LOG2E Property
Returns the base-2 logarithm of *e*, the base of natural logarithms.

MAX_VALUE Property
Returns the largest number representable in JScript. Equal to approximately 1.79E+308.

message Property

Returns an error message string.

MIN_VALUE Property

Returns the number closest to zero that JScript can represent. Equal to approximately 5.00E-324.

multiline Property

Returns a Boolean value indicating the state of the multiline flag (**m**) used with a regular expression.

name Property

Returns the name of an error.

NaN Property

A special value that indicates that an arithmetic expression returned a value that was not a number.

NaN Property (Global)

Returns the special value **NaN** indicating that an expression is not a number.

NEGATIVE_INFINITY Property

Returns a value more negative than the largest negative number (**-Number.MAX_VALUE**) that JScript can represent.

number Property

Returns a value more negative than the largest negative number (**-Number.MAX_VALUE**) that JScript can represent.

PI Property

Returns the ratio of the circumference of a circle to its diameter, approximately 3.141592653589793.

POSITIVE_INFINITY Property

Returns a value larger than the largest number (**Number.MAX_VALUE**) that JScript can represent.

propertyIsEnumerable Property

Returns a Boolean value indicating whether a specified property is part of an object and if it is enumerable.

prototype Property

Returns a reference to the prototype for a class of objects.

rightContext Property ($')

Returns the characters from the position following the last match to the end of the searched string.

source Property

Returns a copy of the text of the regular expression pattern.

SQRT1_2 Property

Returns he square root of 0.5, or one divided by the square root of 2.

SQRT2 Property

Returns the square root of 2

undefined Property

Returns an initial value of **undefined**.

Related Sections

JScript Reference

Lists elements that comprise JScript Language Reference and links to topics that explain the details behind the proper use of language elements.

.NET Framework Reference

Lists links to topics that explain the syntax and structure of the .NET Framework class library and other essential elements.

0...n Properties

Returns the value of individual arguments from an **arguments** object returned by the **arguments** property of an executing function.

```
[function.]arguments[[n]]
```

Arguments

function
Optional. The name of the currently executing **Function** object.

n
Required. Non-negative integer in the range of 0 to **arguments.length-1** where 0 represents the first argument and **arguments.length-1** represents the final argument.

Remarks

The values returned by the 0...*n* properties are the values passed to the executing function. While the **arguments** object is not an array, the individual arguments that comprise the **arguments** object are accessed the same way that array elements are accessed.

> **Note** The **arguments** object is not available when running in fast mode, the default for JScript .NET. To compile a program from the command line that uses the **arguments** object, you must turn off the fast option by using **/fast-**. It is not safe to turn off the fast option in ASP.NET because of threading issues. For more information, see arguments Object.

Example

The following example illustrates the use of the 0...n properties of the **arguments** object.

```
function argTest(){
   var s = "";
   s += "The individual arguments are:\n"
   for (var n=0; n< arguments.length; n++){
      s += "argument " + n;
      s += " is " + argTest.arguments[n] + "\n";
   }
   return(s);
}
print(argTest(1, 2, "hello", new Date()));
```

After compiling this program with the **/fast-** option, the output of this program is:

```
The individual arguments are:
argument 0 is 1
argument 1 is 2
argument 2 is hello
argument 3 is Sat Jan 1 00:00:00 PST 2000
```

Requirements

Version 5.5

See Also

Properties

Applies To: arguments Object | Function object

$1...$9 Properties

Returns the nine most-recently memorized portions found during pattern matching. Read-only.

```
RegExp.$n
```

Arguments

RegExp
Required. The global **RegExp** object.

n

Required. An integer from 1 through 9.

Remarks

The value of the **$1...$9** properties is modified whenever a successful parenthesized match is made. Any number of parenthesized substrings may be specified in a regular expression pattern, but only the nine most recent can be stored.

> **Note** The properties of the **RegExp** object are not available when running in fast mode, the default for JScript .NET. To compile a program from the command line that uses these properties, you must turn off the fast option by using **/fast-**. It is not safe to turn off the fast option in ASP.NET because of threading issues.

Example

The following example illustrates the use of the **$1...$9** properties:

```
var s : String;
var re : RegExp = new RegExp("d(b+)(d)","ig");
var str : String = "cdbBdbsbdbdz";
var arr : Array = re.exec(str);
s = "$1 contains: " + RegExp.$1 + "\n";
s += "$2 contains: " + RegExp.$2 + "\n";
s += "$3 contains: " + RegExp.$3;
print(s);
```

After compiling with the **/fast-** option, the output of this program is:

```
$1 contains: bB
$2 contains: d
$3 contains:
```

Requirements

Version 1

See Also

Regular Expression Syntax

Applies To: RegExp Object

arguments Property

Returns the **arguments** object for the currently executing **Function** object.

```
[function.]arguments
```

Arguments

function
Optional. The name of the currently executing **Function** object.

Remarks

The **arguments** property allows a function to handle a variable number of arguments. The **length** property of the **arguments** object contains the number of arguments passed to the function. The individual arguments contained in the **arguments** object can be accessed in the same way array elements are accessed.

> **Note** The **arguments** object is not available when running in fast mode, the default for JScript .NET. To compile a program from the command line that uses the **arguments** object, you must turn off the fast option by using **/fast-**. It is not safe to turn off the fast option in ASP.NET because of threading issues. For more information, see arguments Object.

Example

The following example illustrates the use of the **arguments** property.

```
function argTest(){
   var s = "";
   s += "The individual arguments are:\n"
   for (var n=0; n< arguments.length; n++){
      s += "argument " + n;
      s += " is " + argTest.arguments[n] + "\n";
   }
   return(s);
}
print(argTest(1, 2, "hello", new Date()));
```

After compiling this program with the **/fast-** option, the output of this program is:

```
The individual arguments are:
argument 0 is 1
argument 1 is 2
argument 2 is hello
argument 3 is Sat Jan 1 00:00:00 PST 2000
```

Requirements

Version 2

See Also

Arguments Object | function Statement

Applies To: Function Object

callee Property

Returns the **Function** object being executed, that is, the body text of the specified **Function** object.

```
[function.]arguments.callee
```

Arguments

function
Optional. The name of the currently executing **Function** object.

Remarks

The **callee** property is a member of the **arguments** object that becomes available only when the associated function is executing.

The initial value of the **callee** property is the **Function** object being executed. This allows anonymous functions to be recursive.

> **Note** The **arguments** object is not available when running in fast mode, the default for JScript .NET. To compile a program from the command line that uses the **arguments** object, you must turn off the fast option by using **/fast-**. It is not safe to turn off the fast option in ASP.NET because of threading issues. For more information, see arguments Object.

Example

The following example illustrates the use of the **callee** property.

```
function factorial(n) {
   if (n <= 0)
      return 1;
   else
      return n * arguments.callee(n - 1)
}
print(factorial(3));
```

After compiling this program with the **/fast-** option, the output of this program is:

```
6
```

Requirements

Version 5.5

See Also

function Statement

Applies To: arguments Object | Function object

caller Property

Returns a reference to the function that invoked the current function.

```
function.caller
```

Arguments

function

Required. The name of the currently executing **Function** object.

Remarks

The **caller** property is only defined for a function while that function is executing. If the function is called from the top level of a JScript program, **caller** contains **null**.

If the **caller** property is used in a string context, the result is the same as *functionName*.**toString**, that is, the decompiled text of the function is displayed.

> **Note** The **caller** property is not available when running in fast mode, the default for JScript .NET. To compile a program from the command line that uses the **caller** property, you must turn off the fast option by using **/fast-**. It is not safe to turn off the fast option in ASP.NET because of threading issues.

Example

The following example illustrates the use of the **caller** property.

```
function callLevel(){
   if (callLevel.caller == null)
      print("callLevel was called from the top level.");
   else {
      print("callLevel was called by:");
      print(callLevel.caller);
   }
}
function testCall() {
   callLevel()
}
// Call callLevel directly.
callLevel();
// Call callLevel indirectly.
testCall();
```

After compiling this program with the **/fast-** option, the output of this program is:

```
callLevel was called from the top level.
callLevel was called by:
function testCall() {
   callLevel()
}
```

Requirements

Version 2

See Also

function Statement

Applies To: arguments Object | Function object

constructor Property

Specifies the function that creates an object.

```
object.constructor
```

Arguments

object
Required. The name of an object or function.

Remarks

The **constructor** property is a member of the prototype of every object that has a prototype. This includes all intrinsic JScript objects except the **arguments**, **Enumerator**, **Error**, **Global**, **Math**, **RegExp**, **Regular Expression**, and **VBArray** objects. The **constructor** property contains a reference to the function that constructs instances of that particular object.

Class-based objects do not have a **constructor** property.

Example

The following example illustrates the use of the **constructor** property.

```
function testObject(ob) {
   if (ob.constructor == String)
      print("Object is a String.");
   else if (ob.constructor == MyFunc)
      print("Object is constructed from MyFunc.");
   else
      print("Object is neither a String or constructed from MyFunc.");
}
// A constructor function.
function MyFunc() {
   // Body of function.
}

var x = new String("Hi");
testObject(x)
var y = new MyFunc;
testObject(y);
```

The output of this program is:

```
Object is a String.
Object is constructed from MyFunc.
```

Requirements

Version 2

See Also

prototype Property

Applies To: Array Object | Boolean Object | Date Object | Function Object | Number Object | Object Object | String Object

description Property

Returns or sets the descriptive string associated with a specific error.

```
object.description
```

Arguments

object
Required. An instance of an **Error** object.

Remarks

The **description** property is a string containing an error message associated with a specific error. Use the value contained in this property to alert a user to an error that the script cannot handle.

The **description** and **message** properties refer to the same message; the **description** property provides backwards compatibility, while the **message** property complies with the ECMA standard.

Example

The following example causes an exception to be thrown, and displays the description of the error.

```
function getAge(age) {
   if(age < 0)
      throw new Error("An age cannot be negative.")
   print("Age is "+age+".");
}

// Pass the getAge an invalid argument.
try {
   getAge(-5);
} catch(e) {
   print(e.description);
}
```

The output of this code is:

```
An age cannot be negative.
```

Requirements

Version 5

See Also

number Property | message Property | name Property

Applies To: Error Object

E Property

Returns the mathematical constant *e*, the base of natural logarithms.

```
Math.E
```

Arguments

Math
Required. The global **Math** object.

Remarks

The **E** property is approximately equal to 2.718.

Requirements

Version 1

See Also

exp Method

Applies To: Math Object

global Property

Returns a Boolean value indicating the state of the global flag (**g**) used with a regular expression.

```
rgExp.global
```

Arguments

rgExp
Required. An instance of a **Regular Expression** object.

Remarks

The **global** property is read-only, and returns **true** if the global flag is set for a regular expression, and returns **false** if it is not. The default value is **false**.

The global flag, when used, indicates that a search should find all occurrences of the pattern within the searched string, not just the first one. This is also known as global matching.

Example

The following example illustrates the use of the **global** property.

```
function RegExpPropDemo(re : RegExp) {
   print("Regular expression: " + re);
   print("global:     " + re.global);
   print("ignoreCase: " + re.ignoreCase);
   print("multiline:  " + re.multiline);
   print();
};

// Some regular expression to test the function.
var re1 : RegExp = new RegExp("the","i");  // Use the constructor.
var re2 = /\w+/gm;                         // Use a literal.
RegExpPropDemo(re1);
RegExpPropDemo(re2);
RegExpPropDemo(/^\s*$/im);
```

The output of this program is:

```
Regular expression: /the/i
global:     false
ignoreCase: true
multiline:  false

Regular expression: /\w+/gm
global:     true
ignoreCase: false
multiline:  true

Regular expression: /^\s*$/im
global:     false
ignoreCase: true
multiline:  true
```

Requirements

Version 5.5

See Also

ignoreCase Property | multiline Property | Regular Expression Syntax

Applies To: Regular Expression Object

ignoreCase Property

Returns a Boolean value indicating the state of the ignoreCase flag (**i**) used with a regular expression.

```
rgExp.ignoreCase
```

Arguments

rgExp
Required. An instance of a **Regular Expression** object.

Remarks

The **ignoreCase** property is read-only, and returns **true** if the ignoreCase flag is set for a regular expression, and returns **false** if it is not. The default value is **false**.

The ignoreCase flag, when used, indicates that a search should ignore case sensitivity when matching the pattern within the searched string.

Example

The following example illustrates the use of the **ignoreCase** property.

```
function RegExpPropDemo(re : RegExp) {
   print("Regular expression: " + re);
   print("global:     " + re.global);
   print("ignoreCase: " + re.ignoreCase);
   print("multiline:  " + re.multiline);
   print();
};

// Some regular expression to test the function.
var re1 : RegExp = new RegExp("the","i");  // Use the constructor.
var re2 = /\w+/gm;                         // Use a literal.
RegExpPropDemo(re1);
RegExpPropDemo(re2);
RegExpPropDemo(/^\s*$/im);
```

The output of this program is:

```
Regular expression: /the/i
global:     false
ignoreCase: true
multiline:  false

Regular expression: /\w+/gm
global:     true
ignoreCase: false
multiline:  true

Regular expression: /^\s*$/im
global:     false
ignoreCase: true
multiline:  true
```

Requirements

Version 5.5

See Also

global property | multiline Property | Regular Expression Syntax

Applies To: Regular Expression Object

index Property

Returns the character position where the first successful match begins in a searched string.

```
{RegExp | reArray}.index
```

Arguments

RegExp
Required. The global **RegExp** object.

reArray
Required. An array returned by the **exec** method of a **Regular Expression** object.

Remarks

The **index** property is zero-based.

The initial value of the **RegExp.index** property is –1. Its value is read-only and changes whenever a successful match is made.

> **Note** The properties of the **RegExp** object are not available when running in fast mode, the default for JScript .NET. To compile a program from the command line that uses these properties, you must turn off the fast option by using **/fast-**. It is not safe to turn off the fast option in ASP.NET because of threading issues.

Example

The following example illustrates the use of the **index** property. This function iterates a search string and prints out the **index** and **lastIndex** values for each word in the string.

```
var src : String = "The rain in Spain falls mainly in the plain.";
var re : RegExp = /\w+/g;
var arr : Array;
while ((arr = re.exec(src)) != null)
   print(arr.index + "-" + arr.lastIndex + "\t" + arr);
```

The output of this program is:

```
0-3     The
4-8     rain
9-11    in
12-17   Spain
18-23   falls
24-30   mainly
31-33   in
34-37   the
38-43   plain
```

Requirements

Version 3

See Also

Regular Expression Syntax | exec Method

Applies To: RegExp Object

Infinity Property

Returns the value of **Number.POSITIVE_INFINITY**.

```
Infinity
```

Remarks

The **Infinity** property is a member of the **Global** object, and is made available when the scripting engine is initialized.

Requirements

Version 3

See Also

POSITIVE_INFINITY Property | NEGATIVE_INFINITY Property

Applies To: Global Object

input Property ($_)

Returns the string against which a regular expression search was performed.

Syntax 1

```
{RegExp | reArray}.input
```

Syntax 2

The **$_** property may be used as shorthand for the **input** property for the **RegExp** object.

```
RegExp.$_
```

Arguments

RegExp
Required. The global **RegExp** object.

reArray
Required. An array returned by the **exec** method of a **Regular Expression** object.

Remarks

The value of **input** property is the string against which a regular expression search was performed.

The initial value of the **RegExp.input** property is an empty string, "". Its value is read-only and changes whenever a successful match is made.

> **Note** The properties of the **RegExp** object are not available when running in fast mode, the default for JScript .NET. To compile a program from the command line that uses these properties, you must turn off the fast option by using **/fast-**. It is not safe to turn off the fast option in ASP.NET because of threading issues.

Example

The following example illustrates the use of the **input** property:

```
var str : String = "A test string.";
var re : RegExp = new RegExp("\\w+","ig");
var arr : Array = re.exec(str);
print("The string used for the match was: " + arr.input);
```

The output of this program is:

```
The string used for the match was: A test string.
```

Requirements

Version 3

See Also

Regular Expression Syntax | exec Method

Applies To: RegExp Object

lastIndex Property

Returns the character position where the next match begins in a searched string.

```
{RegExp | reArray}.lastIndex
```

Arguments

RegExp
Required. The global **RegExp** object.

reArray
Required. An array returned by the **exec** method of a **Regular Expression** object.

Remarks

The **lastIndex** property is zero-based, that is, the index of the first character is zero. Its initial value is –1. Its value is modified whenever a successful match is made.

The **lastIndex** property of the **RegExp** object is modified by the **exec** and **test** methods of the **RegExp** object, and the **match**, **replace**, and **split** methods of the **String** object.

The following rules apply to values of **lastIndex**:

- If there is no match, **lastIndex** is set to -1.
- If **lastIndex** is greater than the length of the string, **test** and **exec** fail and **lastIndex** is set to -1.
- If **lastIndex** is equal to the length of the string, the regular expression matches if the pattern matches the empty string. Otherwise, the match fails and **lastIndex** is reset to -1.
- Otherwise, **lastIndex** is set to the next position following the most recent match.

The initial value of the **RegExp.lastIndex** property is –1. Its value is read-only and changes whenever a successful match is made.

> **Note** The properties of the **RegExp** object are not available when running in fast mode, the default for JScript .NET. To compile a program from the command line that uses these properties, you must turn off the fast option by using **/fast-**. It is not safe to turn off the fast option in ASP.NET because of threading issues.

Example

The following example illustrates the use of the **lastIndex** property. This function iterates a search string and prints out the **index** and **lastIndex** values for each word in the string.

```
var src : String = "The rain in Spain falls mainly in the plain.";
var re : RegExp = /\w+/g;
var arr : Array;
while ((arr = re.exec(src)) != null)
   print(arr.index + "-" + arr.lastIndex + "\t" + arr);
```

The output of this program is:

```
0-3     The
4-8     rain
9-11    in
12-17   Spain
18-23   falls
24-30   mainly
31-33   in
34-37   the
38-43   plain
```

Requirements

Version 3

See Also

Regular Expression Syntax | exec Method

Applies To: RegExp Object

lastParen Property ($+)

Returns the last parenthesized submatch from any regular expression search, if any. Read-only.

```
RegExp.lastParen
```

Arguments

RegExp
Required. The global **RegExp** object.

Remarks

The initial value of the **lastParen** property is an empty string. The value of the **lastParen** property changes whenever a successful match is made.

> **Note** The properties of the **RegExp** object are not available when running in fast mode, the default for JScript .NET. To compile a program from the command line that uses these properties, you must turn off the fast option by using **/fast-**. It is not safe to turn off the fast option in ASP.NET because of threading issues.

Example

The following example illustrates the use of the **lastParen** property:

```
var s;                                //Declare variable.
var re = new RegExp("d(b+)(d)","ig"); //Regular expression pattern.
var str = "cdbBdbsbdbdz";             //String to be searched.
var arr = re.exec(str);               //Perform the search.
s = "$1 returns: " + RegExp.$1 + "\n";
s += "$2 returns: " + RegExp.$2 + "\n";
s += "$3 returns: " + RegExp.$3 + "\n";
s += "input returns : " + RegExp.input + "\n";
s += "lastMatch returns: " + RegExp.lastMatch + "\n";
s += "leftContext returns: " + RegExp.leftContext + "\n";
s += "rightContext returns: " + RegExp.rightContext + "\n";
s += "lastParen returns: " + RegExp.lastParen + "\n";
print(s);                            //Return results.
```

After compiling this program with the **/fast-** option, the output of this program is:

```
$1 returns: bB
$2 returns: d
$3 returns:
input returns : cdbBdbsbdbdz
lastMatch returns: dbBd
leftContext returns: c
rightContext returns: bsbdbdz
lastParen returns: d
```

Requirements

Version 5.5

See Also

$1...$9 Properties | index Property | input Property ($_) | lastIndex Property | lastMatch Property ($&) | leftContext Property ($`) | rightContext Property ($')

Applies To: RegExp Object

leftContext Property ($`)

Returns the characters from the beginning of a searched string up to the position before the beginning of the last match. Read-only.

```
RegExp.leftContext
```

Arguments

RegExp
Required. The global **RegExp** object.

Remarks

The initial value of the **leftContext** property is an empty string. The value of the **leftContext** property changes whenever a successful match is made.

> **Note** The properties of the **RegExp** object are not available when running in fast mode, the default for JScript .NET. To compile a program from the command line that uses these properties, you must turn off the fast option by using **/fast-**. It is not safe to turn off the fast option in ASP.NET because of threading issues.

Example

The following example illustrates the use of the **leftContext** property:

```
var s;                                //Declare variable.
var re = new RegExp("d(b+)(d)","ig"); //Regular expression pattern.
var str = "cdbBdbsbdbdz";             //String to be searched.
var arr = re.exec(str);               //Perform the search.
s = "$1 returns: " + RegExp.$1 + "\n";
s += "$2 returns: " + RegExp.$2 + "\n";
s += "$3 returns: " + RegExp.$3 + "\n";
s += "input returns : " + RegExp.input + "\n";
s += "lastMatch returns: " + RegExp.lastMatch + "\n";
s += "leftContext returns: " + RegExp.leftContext + "\n";
s += "rightContext returns: " + RegExp.rightContext + "\n";
s += "lastParen returns: " + RegExp.lastParen + "\n";
print(s);                            //Return results.
```

After compiling this program with the **/fast-** option, the output of this program is:

```
$1 returns: bB
$2 returns: d
$3 returns:
input returns : cdbBdbsbdbdz
lastMatch returns: dbBd
leftContext returns: c
rightContext returns: bsbdbdz
lastParen returns: d
```

Requirements

Version 5.5

See Also

$1...$9 Properties | index Property | input Property ($_) | lastIndex Property | lastMatch Property ($&) | lastParen Property ($+) | rightContext Property ($')

Applies To: RegExp Object

length Property (arguments)

Returns the actual number of arguments passed to a function by the caller.

```
[function.]arguments.length
```

Arguments

function

Optional. The name of the currently executing **Function** object.

Remarks

The **length** property of the **arguments** object is initialized by the scripting engine to the actual number of arguments passed to a **Function** object when execution begins in that function.

Example

The following example illustrates the use of the **length** property of the **arguments** object.

```
function argTest(a, b) : String {
   var i : int;
   var s : String = "The argTest function expected ";
   var numargs : int = arguments.length; // Get number of arguments passed.
   var expargs : int = argTest.length;   // Get number of arguments expected.
   if (expargs < 2)
      s += expargs + " argument. ";
   else
      s += expargs + " arguments. ";
   if (numargs < 2)
      s += numargs + " was passed.";
   else
      s += numargs + " were passed.";
   s += "\n"
   for (i =0 ; i < numargs; i++){        // Get argument contents.
      s += "  Arg " + i + " = " + arguments[i] + "\n";
   }
   return(s);                            // Return list of arguments.
}

print(argTest(42));
print(argTest(new Date(1999,8,7),"Sam",Math.PI));
```

After compiling with the **/fast-** option, the output of this program is:

```
The argTest function expected 2 arguments. 1 was passed.
  Arg 0 = 42

The argTest function expected 2 arguments. 3 were passed.
  Arg 0 = Tue Sep 7 00:00:00 PDT 1999
  Arg 1 = Sam
  Arg 2 = 3.141592653589793
```

Requirements

Version 5.5

See Also

arguments Property | length Property (Array) | length Property (String)

Applies To: arguments Object

length Property (Array)

Returns an integer value one higher than the highest element defined in an array.

```
arrayObj.length
```

Arguments

arrayObj
Required. Any **Array** object.

Remarks

As the elements in a JScript array do not have to be contiguous, the **length** property is not necessarily the number of elements in the array.

If a value smaller than its previous value is assigned to the **length** property, the array is truncated, and any elements with array indexes equal to or greater than the new value of the **length** property are lost.

If a value larger than its previous value is assigned to the **length** property, the array is formally expanded, but no new elements are created.

Example

The following example illustrates the use of the **length** property. An array is declared, and two elements are added to it. Since the largest index in the array is 6, the length is 7.

```
var my_array : Array = new Array();
my_array[2] = "Test";
my_array[6] = "Another Test";
print(my_array.length); // Prints 7.
```

Requirements

Version 2

See Also

length Property (Function) | length Property (String)

Applies To: Array Object

length Property (Function)

Returns the number of arguments defined for a function.

```
function.length
```

Arguments

function
Required. The name of the currently executing **Function** object.

Remarks

The **length** property of a function is initialized by the scripting engine to the number of arguments in the function's definition when an instance of the function is created.

What happens when a function is called with a number of arguments different from the value of its **length** property depends on the function.

Example

The following example illustrates the use of the **length** property:

```
function argTest(a, b) : String {
   var s : String = "The argTest function expected " ;
   var expargs : int = argTest.length;
   s += expargs;
   if (expargs < 2)
      s += " argument.";
   else
      s += " arguments.";
   return(s);
}
// Display the function output.
print(argTest(42,"Hello"));
```

The output of this program is:

```
The argTest function expected 2 arguments.
```

Requirements

Version 2

See Also

arguments Property | length Property (Array) | length Property (String)

Applies To: Function Object

length Property (String)

Returns the length of a string.

```
str.length
```

Arguments

str
Required. A string literal or the name of a **String** object.

Remarks

The **length** property contains an integer that indicates the number of characters in the **String** object. The last character in the **String** object has an index of **length** - 1.

Requirements

Version 1

See Also

length Property (Array) | length Property (Function)

Applies To: String Object

LN10 Property

Returns the natural logarithm of 10.

```
Math.LN10
```

Arguments

Math
Required. The global **Math** object.

Remarks

The **LN10** property is approximately equal to 2.302.

Requirements

Version 1

See Also

length Property (Array) | length Property (Function)

Applies To: Math Object

LN2 Property

Returns the natural logarithm of 2.

```
Math.LN2
```

Arguments

Math
Required. The global **Math** object.

Syntax

The **LN2** property is approximately equal to 0.693.

Requirements

Version 1

See Also

length Property (Array) | length Property (Function)

Applies To: Math Object

LOG10E Property

Returns the base-10 logarithm of *e*, the base of natural logarithms.

```
Math.LOG10E
```

Arguments

Math
Required. The global **Math** object.

Remarks

The **LOG10E** property, a constant, is approximately equal to 0.434.

Requirements

Version 1

See Also

length Property (Array) | length Property (Function)

Applies To: Math Object

LOG2E Property

Returns the base-2 logarithm of *e*, the base of natural logarithms.

```
Math.LOG2E
```

Arguments

Math
Required. The global **Math** object.

Remarks

The **LOG2E** property, a constant, is approximately equal to 1.442.

Requirements

Version 1

See Also

length Property (Array) | length Property (Function)

Applies To: Math Object

MAX_VALUE Property

Returns the largest number representable in JScript. Equal to approximately 1.79E+308.

```
Number.MAX_VALUE
```

Arguments

Number
Required. The global **Number** object.

Remarks

The **Number** object does not have to be created before the **MAX_VALUE** property can be accessed.

Requirements

Version 2

See Also

MIN_VALUE Property | NaN Property | NEGATIVE_INFINITY Property | POSITIVE_INFINITY Property | toString Method

Applies To: Number Object

message Property

Returns an error message string.

```
errorObj.message
```

Arguments

errorObj
Required. Instance of **Error** object.

Remarks

The **message** property is a string containing an error message associated with a specific error. Use the value contained in this property to alert a user to an error that you can't or don't want to handle.

The **description** and **message** properties refer to the same message; the **description** property provides backwards compatibility, while the **message** property complies with the ECMA standard.

Example

The following example causes an exception to be thrown, and displays the message of the error.

```
function getAge(age) {
   if(age < 0)
      throw new Error("An age cannot be negative.")
   print("Age is "+age+".");
}

// Pass the getAge an invalid argument.
try {
   getAge(-5);
} catch(e) {
   print(e.message);
}
```

The output of this code is:

```
An age cannot be negative.
```

Requirements

Version 5.5

See Also

description Property | name Property

Applies To: Error Object

MIN_VALUE Property

Returns the number closest to zero representable in JScript. Equal to approximately 5.00E-324.

```
Number.MIN_VALUE
```

Arguments

Number

Required. The global **Number** object.

Remarks

The **Number** object does not have to be created before the **MIN_VALUE** property can be accessed.

Requirements

Version 2

See Also

MAX_VALUE Property | NaN Property | NEGATIVE_INFINITY Property | POSITIVE_INFINITY Property | toString Method

Applies To: Number Object

multiline Property

Returns a Boolean value indicating the state of the multiline flag (**m**) used with a regular expression.

```
rgExp.multiline
```

Arguments

rgExp
Required. An instance of a **Regular Expression** object.

Remarks

The **multiline** property is read-only, and returns **true** if the multiline flag is set for a regular expression, and returns **false** if it is not. The **multiline** property is **true** if the regular expression object was created with the **m** flag. The default value is **false**.

If **multiline** is **false**, "^" matches the position at the beginning of a string, and "$" matches the position at the end of a string. If **multiline** is **true**, "^" matches the position at the beginning of a string as well as the position following a "\n" or "\r", and "$" matches the position at the end of a string and the position preceding "\n" or "\r".

Example

The following example illustrates the use of the **multiline** property.

```
function RegExpPropDemo(re : RegExp) {
   print("Regular expression: " + re);
   print("global:      " + re.global);
   print("ignoreCase: " + re.ignoreCase);
   print("multiline:  " + re.multiline);
   print();
};

// Some regular expression to test the function.
var re1 : RegExp = new RegExp("the","i");  // Use the constructor.
var re2 = /\w+/gm;                          // Use a literal.
RegExpPropDemo(re1);
RegExpPropDemo(re2);
RegExpPropDemo(/^\s*$/im);
```

The output of this program is:

```
Regular expression: /the/i
global:     false
ignoreCase: true
multiline:  false

Regular expression: /\w+/gm
global:     true
ignoreCase: false
multiline:  true

Regular expression: /^\s*$/im
global:     false
ignoreCase: true
multiline:  true
```

Requirements

Version 5.5

See Also

global property | ignoreCase Property | Regular Expression Syntax

Applies To: Regular Expression Object

name Property

Returns the name of an error.

```
errorObj.name
```

Arguments

errorObj
Required. Instance of **Error** object.

Remarks

The **name** property returns the name or exception type of an error. When a runtime error occurs, the name property is set to one of the following native exception types:

Exception Type	Meaning
Error	This error is a user-defined error, created using the **Error** object constructor.
ConversionError	This error occurs whenever there is an attempt to convert an object into something to which it cannot be converted.
RangeError	This error occurs when a function is supplied with an argument that has exceeded its allowable range. For example, this error occurs if you attempt to construct an **Array** object with a length that is not a valid positive integer.
ReferenceError	This error occurs when an invalid reference has been detected. This error will occur, for example, if an expected reference is **null**.
RegExpError	This error occurs when a compilation error occurs with a regular expression. Once the regular expression is compiled, however, this error cannot occur. This example will occur, for example, when a regular expression is declared with a pattern that has an invalid syntax, or flags other than **i**, **g**, or **m**, or if it contains the same flag more than once.
SyntaxError	This error occurs when source text is parsed and that source text does not follow correct syntax. This error will occur, for example, if the **eval** function is called with an argument that is not valid program text.
TypeError	This error occurs whenever the actual type of an operand does not match the expected type. An example of when this error occurs is a function call made on something that is not an object or does not support the call.
URIError	This error occurs when an illegal Uniform Resource Indicator (URI) is detected. For example, this is error occurs when an illegal character is found in a string being encoded or decoded.

Example

The following example causes an exception to be thrown, and displays the error and the description of the error.

```
function getAge(age) {
   if(age < 0)
      throw new Error("An age cannot be negative.")
   print("Age is "+age+".");
}

// Pass the getAge an invalid argument.
try {
   getAge(-5);
} catch(e) {
   print(e.name);
   print(e.description);
}
```

The output of this code is:

```
Error
```

```
An age cannot be negative.
```

Requirements

Version 5.5

See Also

description Property | message Property | number Property

Applies To: Error Object

NaN Property

A special value that indicates an arithmetic expression returned a value that was not a number.

```
Number.NaN
```

Arguments

Number
Required. The global **Number** object.

Remarks

The **Number** object does not have to be created before the **NaN** property can be accessed.

NaN does not compare equal to any value, including itself. To test if a value is equivalent to **NaN**, use the **isNaN** method of the **Global** object..

Requirements

Version 2

See Also

isNaN Method | MAX_VALUE Property | MIN_VALUE Property | NEGATIVE_INFINITY Property | POSITIVE_INFINITY Property | toString Method

Applies To: Number Object

NaN Property (Global)

Returns the special value **NaN** indicating that an expression is not a number.

```
NaN
```

Remarks

The **NaN** property (not a number) is a member of the **Global** object, and is made available when the scripting engine is initialized.

NaN does not compare equal to any value, including itself. To test if a value is equivalent to **NaN**, use the **isNaN** method of the **Global** object..

Requirements

Version 3

See Also

isNaN Method

Applies To: Global Object

NEGATIVE_INFINITY Property

Returns a value more negative than the largest negative number (**-Number.MAX_VALUE**) representable in JScript.

```
Number.NEGATIVE_INFINITY
```

Arguments

Number
Required. The global **Number** object.

Remarks

The **Number** object does not have to be created before the **NEGATIVE_INFINITY** property can be accessed.

JScript displays **NEGATIVE_INFINITY** values as -Infinity. This value behaves mathematically as infinity.

Requirements

Version 2

See Also

MAX_VALUE Property | MIN_VALUE Property | NaN Property | POSITIVE_INFINITY Property | toString Method

Applies To: Number Object

number Property

Returns or sets the numeric value associated with a specific error.

```
object.number
```

Arguments

object
Any instance of the **Error** object.

Remarks

An error number is a 32-bit value. The upper 16-bit word is the facility code, while the lower word is the actual error code. To read off the actual error code, use the **&** (bitwise And) operator to combine the number property with the hexadecimal number `0xFFFF`.

Example

The following example causes an exception to be thrown, and displays the error number.

```
function getAge(age) {
   if(age < 0)
      throw new Error(100)
   print("Age is "+age+".");
}

// Pass the getAge an invalid argument.
try {
   getAge(-5);
} catch(e) {
// Extract the error code from the error number.
   print(e.number & 0xFFFF)
}
```

The output of this code is:

```
100
```

Requirements

Version 5

See Also

description Property | message Property | name Property

Applies To: Error Object

PI Property

Returns the value of the mathematical constant pi.

```
Math.PI
```

Arguments

Math

Required. The global **Math** object.

Syntax

The **PI** property is a constant approximately equal to 3.14159. It represents the ratio of the circumference of a circle to its diameter.

Requirements

Version 1

See Also

Properties

Applies To: Math Object

POSITIVE_INFINITY Property

Returns a value larger than the largest number (**Number.MAX_VALUE**) that can be represented in JScript.

```
Number.POSITIVE_INFINITY
```

Arguments

Number
Required. The global **Number** object.

Remarks

The **Number** object does not have to be created before the **POSITIVE_INFINITY** property can be accessed.

JScript displays **POSITIVE_INFINITY** values as Infinity. This value behaves mathematically as infinity.

Requirements

Version 2

See Also

MAX_VALUE Property | MIN_VALUE Property | NaN Property | NEGATIVE_INFINITY Property | toString Method

Applies To: Number Object

propertyIsEnumerable Property

Returns a Boolean value indicating whether a specified property is part of an object and if it is enumerable.

```
object.propertyIsEnumerable(propName)
```

Arguments

object
Required. Instance of an object.
propName
Required. String value of a property name.

Remarks

The **propertyIsEnumerable** property returns **true** if *propName* exists in *object* and can be enumerated using a **For...In** loop. The **propertyIsEnumerable** property returns **false** if *object* does not have a property of the specified name or if the specified property is not enumerable. Typically, predefined properties are not enumerable while user-defined properties are always enumerable.

The **propertyIsEnumerable** property does not consider objects in the prototype chain.

Example

The following example illustrates the use of the **propertyIsEnumerable** property.

```
var a : Array = new Array("apple", "banana", "cactus");
print(a.propertyIsEnumerable(1));
```

The output of this program is:

```
true
```

Requirements

Version 5.5

See Also

Properties

Applies To: Object Object

prototype Property

Returns a reference to the prototype for a class of objects.

```
object.prototype
```

Arguments

object
Required. The name of an object.

Remarks

Use the **prototype** property to provide a base set of functionality to a class of objects. New instances of an object "inherit" the behavior of the prototype assigned to that object.

All intrinsic JScript objects have a **prototype** property that is read-only. Functionality may be added to the prototype, as in the example, but the object may not be assigned a different prototype. However, user-defined objects may be assigned a new prototype.

The method and property lists for each intrinsic object in this language reference indicate which ones are parts of the object's prototype, and which are not.

> **Note** The **prototype** property of a built-in object cannot be modified when running in fast mode, the default for JScript .NET. To compile a program from the command line that uses the **prototype** property, you must turn off the fast option by using **/fast-**. It is not safe to turn off the fast option in ASP.NET because of threading issues.

Example

Say you want to add a method to the **Array** object that returns the value of the largest element of the array. To do this, declare the function, add it to **Array.prototype**, and then use it.

```
function array_max() {
   var i, max = this[0];
   for (i = 1; i < this.length; i++) {
      if (max < this[i])
         max = this[i];
   }
   return max;
}
Array.prototype.max = array_max;
var x = new Array(1, 2, 3, 4, 5, 6);
print(x.max());
```

After compiling the with the /fast- option, the output of the programs is:

```
6
```

Requirements

Version 2

See Also

constructor Property

Applies To: Array Object | Boolean Object | Date Object | Function Object | Number Object | Object Object | String Object

rightContext Property ($')

Returns the characters from the position following the last match to the end of the searched string. Read-only.

```
RegExp.rightContext
```

Arguments

RegExp
Required. The global **RegExp** object.

Remarks

The initial value of the **rightContext** property is an empty string. The value of the **rightContext** property changes whenever a successful match is made.

> **Note** The properties of the **RegExp** object are not available when running in fast mode, the default for JScript .NET. To compile a program from the command line that uses these properties, you must turn off the fast option by using **/fast-**. It is not safe to turn off the fast option in ASP.NET because of threading issues.

Example

The following example illustrates the use of the **rightContext** property:

```
var s;                                //Declare variable.
var re = new RegExp("d(b+)(d)","ig"); //Regular expression pattern.
var str = "cdbBdbsbdbdz";             //String to be searched.
var arr = re.exec(str);               //Perform the search.
s = "$1 returns: " + RegExp.$1 + "\n";
s += "$2 returns: " + RegExp.$2 + "\n";
s += "$3 returns: " + RegExp.$3 + "\n";
s += "input returns : " + RegExp.input + "\n";
s += "lastMatch returns: " + RegExp.lastMatch + "\n";
s += "leftContext returns: " + RegExp.leftContext + "\n";
s += "rightContext returns: " + RegExp.rightContext + "\n";
s += "lastParen returns: " + RegExp.lastParen + "\n";
print(s);                            //Return results.
```

After compiling this program with the **/fast-** option, the output of this program is:

```
$1 returns: bB
$2 returns: d
$3 returns:
input returns : cdbBdbsbdbdz
lastMatch returns: dbBd
leftContext returns: c
rightContext returns: bsbdbdz
lastParen returns: d
```

Requirements

Version 5.5

See Also

$1...$9 Properties | index Property | input Property ($_) | lastIndex Property | lastMatch Property ($&) | lastParen Property ($+) | leftContext Property ($`)

Applies To: RegExp Object

source Property

Returns a copy of the text of the regular expression pattern. Read-only.

```
rgExp.source
```

Arguments

rgExp
Required. A **Regular Expression** object.

Remarks

The *rgExp* can be a variable that stores a **Regular Expression** object, or it can be a regular expression literal.

Example

The following example illustrates the use of the **source** property:

```
var src : String = "Spain";
var re : RegExp = /in/g;
var s1;
// Test string for existence of regular expression.
if (re.test(src))
   s1 = " contains ";
else
   s1 = " does not contain ";
// Get the text of the regular expression itself.
print("The string " + src + s1 + re.source + ".");
```

The output of this program is:

```
The string Spain contains in.
```

Requirements

Version 3

See Also

Regular Expression Object | Regular Expression Syntax

Applies To: Regular Expression Object

SQRT1_2 Property

Returns the square root of 0.5, or one divided by the square root of 2.

```
Math.SQRT1_2
```

Arguments

Math
Required. The global **Math** object.

Remarks

The **SQRT1_2** property, a constant, is approximately equal to 0.707.

Requirements

Version 1

See Also

sqrt Method | SQRT2 Property

Applies To: Math Object

SQRT2 Property

Returns the square root of 2.

```
Math.SQRT2
```

Arguments

Math
Required. The global **Math** object.

Syntax

The **SQRT2** property, a constant, is approximately equal to 1.414.

Requirements

Version 1

See Also

sqrt Method | SQRT1_2 Property

Applies To: Math Object

undefined Property

Returns the value of **undefined**.

```
undefined
```

Remarks

The **undefined** property is a member of the **Global** object, and becomes available when the scripting engine is initialized. When a variable has been declared but not initialized, its value is **undefined**.

If a variable has not been declared, you cannot compare it to **undefined**, but you can compare the type of the variable to the string "undefined". Undeclared variables cannot be used in fast mode.

The **undefined** property is useful when explicitly testing or setting a variable to undefined.

Example

The following example illustrates the use of the **undefined** property:

```
var declared;                         //Declare variable.
if (declared == undefined)            //Test variable.
   print("The variable declared has not been given a value.");
```

The output of this code is:

```
The variable declared has not been given a value.
```

Requirements

Version 5.5

See Also

Undefined Values

Applies To: Global Object

Statements

A statement is a piece of JScript code that performs an action. Some statements declare user-defined elements, such as variables, functions, classes, and enumerations, while other statements control program flow. The following sections link to information that explains how to use statements in JScript.

In This Section

break Statement
Terminates the current loop, or if in conjunction with a label, terminates the associated statement.

class Statement
Defines a class and the members of that class.

@cc_on Statement
Activates conditional compilation support.

Comment Statements
Causes comments, either single line (//) or multiline (/* */) to be ignored by the JScript parser.

const Statement
Defines a constant identifier and its value.

continue Statement
Stops the current iteration of a loop and starts a new iteration.

debugger Statement
Launches the installed debugger.

do...while Statement
Executes a statement block once, and then repeats execution of the loop until a condition expression evaluates to **false**.

enum Statement
Declares an enumeration and the enumeration values.

for Statement
Executes a block of statements for as long as a specified condition is **true**.

for...in Statement
Executes one or more statements for each element of an object or array.

function Statement
Declares a new function.

function get Statement
Declares the setter for a property.

function set Statement
Declares the getter for a property.

@if...@elif...@else...@end Statement
Conditionally executes a group of statements depending on the value of an expression.

if...else Statement
Conditionally executes a group of statements depending on the value of an expression.

import Statement
Enables access to external library.

interface Statement
Declares an interface and the members of the interface.

Labeled Statement
Provides an identifier for a statement.

package Statement
Provides a way to package classes and interfaces in named components.

print Statement
Provides a way to display information from a program run from the command line.

return Statement
Exits from the current function and returns a value from that function.

@set Statement
Creates variables used with conditional compilation statements.

static Statement
Declares a block of code that initializes a class.

super Statement
Refers to the base class of the current object.

switch Statement
Enables the execution of one or more statements when a specified expression's value matches a label.

this Statement

Refers to the current object.

throw Statement

Generates an error condition that can be handled by a **try...catch** statement.

try...catch...finally Statement

Implements error handling for JScript.

var Statement

Declares a variable.

while Statement

Executes a statement until a specified condition is **false**.

with Statement

Establishes the default object for a statement.

Related Sections

JScript Reference

Lists elements that comprise JScript Language Reference and links to topics that explain the details behind the proper use of language elements.

.NET Framework Reference

Lists links to topics that explain the syntax and structure of the .NET Framework class library and other essential elements.

break Statement

Terminates the current loop, or if in conjunction with a label, terminates the associated statement.

```
break [label];
```

Arguments

label
Optional. Specifies the label of the statement you are breaking from.

Remarks

You typically use the **break** statement in **switch** statements and **while**, **for**, **for...in**, or **do...while** loops. You most commonly use the *label* argument in **switch** statements, but it can be used in any statement, whether simple or compound.

Executing the **break** statement causes the program flow to exit the current loop or statement. Program flow resumes with the next statement immediately following the current loop or statement.

Example 1

The following example illustrates the use of the **break** statement.

```
function breakTest(breakpoint){
   var i = 0;
   while (i < 100) {
      if (i == breakpoint)
         break;
      i++;
   }
   return(i);
}
```

Example 2

The following example illustrates the use of the labeled **break** statement.

```
function nameInDoubleArray(name, doubleArray) {
   var i, j, inArray;
   inArray = false;
   mainloop:
   for(i=0; i<doubleArray.length; i++)
      for(j=0; j<doubleArray[i].length; j++)
         if(doubleArray[i][j] == name) {
            inArray = true;
            break mainloop;
         }
   return inArray;
}
```

Requirements

Version 1

See Also

continue Statement | do...while Statement | for Statement | for...in Statement | Labeled Statement | while Statement

class Statement

Declares the name of a class as well as a definition of the variables, properties, and methods that comprise the class.

```
[modifiers] class classname [extends baseclass] [implements interfaces]{
   [classmembers]
}
```

Arguments

modifiers
Optional. Modifiers that control the visibility and behavior of the class.

classname
Required. Name of the **class**; follows standard variable naming conventions.

extends
Optional. Keyword indicating that the class *classname* extends *baseclass*. If this keyword is not used, a standard JScript base class is created that extends **System.Object**.

baseclass
Optional. The name of the class being extended.

implements
Optional. Keyword indicating that the class *classname* implements one or more interfaces.

interfaces
Optional. A comma-delimited list of interface names.

classmembers
Optional. *classmembers* can be method or constructor declarations (defined with the **function** statement), property declarations (defined with the **function get** and **function set** statements), field declarations (defined with the **var** or **const** statements), initializer declarations (defined with the **static** statement), enumeration declarations (defined with the **enum** statement), or nested class declarations.

Remarks

Classes can be used to create instances or serve as the base for other classes, depending on the modifiers of the class. If a class is marked with the **abstract** modifier, the class can serve as a base class for other classes to extend, but instances of an **abstract** class cannot be created. If a class is marked with the **final** modifier, instances of the class can be created with the **new** operator, but the class cannot serve as a base.

Methods and constructors may be overloaded in a class. Consequently, multiple methods (or constructors) may have the same names. Overloaded class members are distinguished by their unique signatures, which are comprised of the name of the member and the data type of each of its formal parameters. Overloads allow a class to group methods with similar functionality.

A class can inherit the functionality of an existing base class by using the **extends** keyword. Methods of the base class can be overridden by declaring a new method with the same signature as the new class method. Methods in the new class can access overridden members of the base class using the **super** statement.

A class can be patterned on one or more interfaces using the **implements** keyword. A class cannot inherit any behavior from an interface because an interface does not provide an implementation for any member. An interface does provide the class with a 'signature' that can be used when interacting with other classes. Unless the class that implements an interface is **abstract**, an implementation must be provided for every method defined in the interface.

Modifiers can be used to make a class instance behave more like a JScript object. To allow a class instance to handle dynamically added properties, use the **expando** modifier, which automatically creates a default indexed property for the class. Only expando properties are accessible using the square bracket notation of the JScript **Object** object.

Example 1

The following example creates a `CPerson` class with various fields and methods, the details of which have been omitted. The `CPerson` class serves as the base class for the `CCustomer` class in the second example.

```
// All members of CPerson are public by default.
class CPerson{
   var name : String;
   var address : String;
   // CPerson constuctor
   function CPerson(name : String){
      this.name = name;
   };

   // printMailingLabel is an instance method, as it uses the
   // name and address information of the instance.
   function printMailingLabel(){
      print(name);
      print(address);
   };

   // printBlankLabel is static as it does not require
   // any person-specific information.
   static function printBlankLabel(){
      print("-blank-");
   };
}

// Print a blank mailing label.
// Note that no CPerson object exists at this time.
CPerson.printBlankLabel();

// Create a CPerson object and add some data.
var John : CPerson = new CPerson("John Doe");
John.address = "15 Broad Street, Atlanta, GA 30315";
// Print a mailing label with John's name and address.
John.printMailingLabel();
```

The output of this code is:

```
-blank-
John Doe
15 Broad Street, Atlanta, GA 30315
```

Example 2

A `CCustomer` class is derived from `CPerson`, having additional fields and methods not applicable to a generic member of the `CPerson` class.

```
// Create an extension to CPerson.
class CCustomer extends CPerson{
   var billingAddress : String;
   var lastOrder : String;

   // Constructor for this class.
   function CCustomer(name : String, creditLimit : double){
      super(name); // Call superclass constructor.
      this.creditLimit = creditLimit;
   };

   // Customer's credit limit. This is a private field
   // so that only member functions can change it.
   private var creditLimit : double;
   // A public property is needed to read the credit limit.
   function get CreditLimit() : double{
      return creditLimit;
   }
}

// Create a new CCustomer.
var Jane : CCustomer = new CCustomer("Jane Doe",500.);
// Do something with it.
Jane.billingAddress = Jane.address = "12 Oak Street, Buffalo, NY 14201";
Jane.lastOrder = "Windows 2000 Server";
// Print the credit limit.
print(Jane.name + "'s credit limit is " + Jane.CreditLimit);
// Call a method defined in the base class.
Jane.printMailingLabel();
```

The output of this part of the code is:

```
Jane Doe's credit limit is 500
Jane Doe
12 Oak Street, Buffalo, NY 14201
```

Requirements

Version .NET

See Also

Modifiers | interface Statement | function Statement | function get Statement | function set Statement | var Statement | const Statement | static Statement | new Operator | this Statement | super Statement

@cc_on Statement

Activates conditional compilation support.

```
@cc_on
```

Remarks

The **@cc_on** statement activates conditional compilation in the scripting engine.

It is strongly recommended that you use the **@cc_on** statement in a comment, so that browsers that do not support conditional compilation will accept your script as valid syntax:

```
/*@cc_on*/
// The remainder of the script.
```

Alternatively, an **@if** or **@set** statement outside of a comment also activates conditional compilation.

Requirements

Version 3

See Also

Conditional Compilation | Conditional Compilation Variables | @if Statement | @set Statement

Comment Statements

Causes comments to be ignored by the JScript parser.

Syntax 1

Single-line comment:

```
// comment
```

Syntax 2

Multiline comment:

```
/*
comment
*/
```

The *comment* is the text of any comment you want to include in your script.

Syntax 3

Single-line conditional comment:

```
//@CondStatement
```

Syntax 4

Multiline conditional comment:

```
/*@
condStatement
@*/
```

The c*ondStatement* argument is conditional compilation code to be used if conditional compilation is activated. If Syntax 3 is used, there can be no space between the "//" and "@" characters.

Remarks

Use comments to keep parts of a script from being read by the JScript parser. You can use comments to include explanatory remarks in a program.

If Syntax 1 is used, the parser ignores any text between the comment marker and the end of the line. If Syntax 2 is used, it ignores any text between the beginning and end markers.

Syntaxes 3 and 4 are used to support conditional compilation while retaining compatibility with browsers that do not support that feature. These browsers treat those forms of comments as syntaxes 1 and 2 respectively.

Example

The following example illustrates the most common uses of the **comment** statement.

```
function myfunction(arg1, arg2){
   /* This is a multiline comment that
      can span as many lines as necessary. */
   var r = 0;
   // This is a single line comment.
   r = arg1 + arg2; // Sum the two arguments.
   return(r);
   }
```

Requirements

Version 1

See Also

Conditional Compilation | Conditional Compilation Variables | @cc_on Statement | @set Statement

const Statement

Declares a constant.

Syntax for declaring a constant of global scope or function scope.

```
const name1 [: type1] = value1 [, ... [, nameN [: typeN] = valueN]]
```

Syntax for declaring a constant field in a class.

```
[modifiers] const name1 [: type1] = value1 [, ... [, nameN [: typeN] = valueN]]
```

Arguments

modifiers
Optional. Modifiers that control the visibility and behavior of the field.
name1, ..., nameN
Required. The names of the constants being declared.
type1, ..., typeN
Optional. The types of the constants being declared.
value1, ..., valueN
The values assigned to the constants.

Remarks

Use the **const** statement to declare constants. A constant may be bound to a specific data type to ensure type safety. These constants must be assigned values when they are declared, and these values cannot be changed later in the script.

A constant field in a class is similar to a global or function constant, except that it is scoped to the class and it can have various modifiers governing its visibility and usage.

> **Note** When a constant is bound to a reference data type (such as an **Object**, **Array**, class instance, or typed array), the data referenced by the constant may be changed. This is allowed because the **const** statement makes only the reference type constant; the data to which it refers is not constant.

Example

The following examples illustrate the use of the **const** statement.

```
class CSimple {
   // A static public constant field. It will always be 42.
   static public const constantValue : int = 42;
}
const index = 5;
const name : String = "Thomas Jefferson";
const answer : int = 42, oneThird : float = 1./3.;
const things : Object[] = new Object[50];
things[1] = "thing1";
// Changing data referenced by the constant is allowed.
```

Requirements

Version .NET

See Also

Modifiers | var Statement | function Statement | class Statement | Scope of Variables and Constants | Type Annotation

continue Statement

Stops the current iteration of a loop, and starts a new iteration.

```
continue [label];
```

Arguments

label
Optional. Specifies the statement to which **continue** applies.

Remarks

You can use the continue statement inside while, **do...while**, **for**, or **for...in** loops only. Executing the continue statement stops the current iteration of the loop and continues program flow with the beginning of the loop. This has the following effects on the different types of loops:

- **while** and **do...while** loops test their condition, and if true, execute the loop again.
- **for** loops execute their increment expression, and if the test expression is true, execute the loop again.
- **for...in** loops proceed to the next field of the specified variable and execute the loop again.

Example

The following example illustrates the use of the **continue** statement.

```
function skip5(){
   var s = "", i=0;
   while (i < 10) {
      i++;
      // Skip 5
      if (i==5) {
         continue;
      }
      s += i;
   }
   return(s);
}
```

Requirements

Version 1

See Also

break Statement | do...while Statement | for Statement | for...in Statement | Labeled Statement | while Statement

debugger Statement

Launches the debugger.

```
debugger
```

Remarks

The **debugger** statement launches the installed debugger. The effect is similar to setting a breakpoint in the program where the debugger statement is used.

If no debugger is installed, the **debugger** statement has no effect.

Requirements

Version 3

See Also

Statements | Writing, Compiling, and Debugging JScript Code

do...while Statement

Executes a statement block once, and then repeats execution of the loop until a condition expression evaluates to **false**.

```
do
   statement
while (expression)
```

Arguments

statement
Required. Statement to be executed if *expression* is **true**. Can be a compound statement.
expression
Required. An expression that can be coerced to Boolean **true** or **false**. If *expression* is **true**, the loop is executed again. If *expression* is **false**, the loop is terminated.

Remarks

The value of *expression* is not checked until after the first iteration of the loop, guaranteeing that the loop is executed at least once. Thereafter, it is checked after each succeeding iteration of the loop.

Example

The following example illustrates the use of the **do...while** statement to iterate the **Drives** collection.

```
function GetDriveList(){
   var fso, s, n, e, x;
   fso = new ActiveXObject("Scripting.FileSystemObject");
   e = new Enumerator(fso.Drives);
   s = "";
   do {
      x = e.item();
      s = s + x.DriveLetter;
      s += " - ";
      if (x.DriveType == 3)
         n = x.ShareName;
      else if (x.IsReady)
         n = x.VolumeName;
      else
         n = "[Drive not ready]";
         s +=  n + "\n";
      e.moveNext();
   }
   while (!e.atEnd());
   return(s);
}
```

Requirements

Version 3

See Also

break Statement | continue Statement | for Statement | for...in Statement | while Statement | Labeled Statement

enum Statement

Declares the name of an enumerated data type and the names of the members of the enumeration.

```
[modifiers] enum enumName [ : typeAnnotation]{
   enumValue1 [ = initializer1]
   [,enumValue2 [ = initializer2]
   [, ... [,enumValueN [ = initializerN ] ]]]
}
```

Arguments

modifiers
Optional. Modifiers that control the visibility and behavior of the enumeration.

enumName
Required. Name of the enumerated type.

typeAnnotation
Optional. The underlying data type of the enumeration. Must be an integral data type. The default is **int**.

enumValue1, enumValue2, ..., enumValueN
Optional. An enumerated type member.

initializer1, initializer2, ..., initializerN
Optional. A constant expression that overrides the default numerical value of an enumeration member.

Remarks

An **enum** declaration introduces a new enumerated data type into the program. An **enum** declaration can appear only in contexts where a class declaration can appear, that is, at global scope, at package scope, or at class scope, but not inside a function or method.

You can declare the underlying type of an enumeration to be any integral data type (**int**, **short**, **long**, **byte**, **uint**, **ushort**, **ulong**, or **sbyte**). Enumeration members implicitly coerce to and from the underlying data type, allowing for direct assignments of numeric data to variables typed as **enum**. By default, the underlying data type of an enumeration is **int**.

Each enumerated type member has a name and an optional initializer. An initializer must be a compile-time, constant expression that is of the same type as the enumeration specified, or convertible to that type. The value of the first enumerated type member is zero or the value of the initializer, if provided. The value of each subsequent enumerated type member is one more then the previous member or the value of the initializer, if provided.

An **enum** value is accessed in a manner that's similar to accessing a static class member. The name of the member must be qualified with the name of the enumeration, for example `Color.Red`. When assigning a value to a variable of an **enum** type, one of following may be used: a fully qualified name (such as `Color.Red`), a string representation of the name (such as `"Red"`), or a numeric value.

If an **enum** is assigned a string that is known at compile time, the compiler will perform the necessary conversion. For example, `"Red"` would be replaced with `Color.Red`. If the string is not known at compile time, a conversion will be made at run time. That conversion may fail if the string is not a valid member of the enumerated type. Because the conversion takes time and run-time errors may be generated, avoid assigning an **enum** to a variable string.

A variable of an enumerated type can hold values outside the range of declared values. One use of this feature is to allow combinations of members used as bit flags, as in done in the example below. Converting an **enum** variable to a string results in the string representation of the member name.

Example 1

The following example shows the behavior of enumerations. It declares a simple enumeration named `CarType` that has members `Honda`, `Toyota`, and `Nissan`.

```
enum CarType {
   Honda,    // Value of zero, since it is first.
   Toyota,   // Value of 1, the successor of zero.
   Nissan    // Value of 2.
}
```

```
// Declare a variable of type CarType, and give it the value Honda.
var myCar : CarType = CarType.Honda;
print(int(myCar) + ": " + myCar);

myCar = "Nissan"; // Change the value to "Nissan".
print(int(myCar) + ": " + myCar);

myCar = 1; // 1 is the value of the Toyota member.
print(int(myCar) + ": " + myCar);
```

The output of this code is:

```
0: Honda
2: Nissan
1: Toyota
```

Example 2

The following example shows how to use an enumeration to hold bit flags and also that an **enum** variable must be able to hold values not explicitly in the member list. It defines an enumeration `FormatFlags` that is used to modify the behavior of a `Format` function.

```
// Explicitly set the type to byte, as there are only a few flags.
enum FormatFlags : byte {
   // Can't use the default values, since we need explicit bits
   ToUpperCase = 1,   // Should not combine ToUpper and ToLower.
   ToLowerCase = 2,
   TrimLeft    = 4,   // Trim leading spaces.
   TrimRight   = 8,   // Trim trailing spaces.
   UriEncode   = 16   // Encode string as a URI.
}

function Format(s : String, flags : FormatFlags) : String {
   var ret : String = s;
   if(flags & FormatFlags.ToUpperCase) ret = ret.toUpperCase();
   if(flags & FormatFlags.ToLowerCase) ret = ret.toLowerCase();
   if(flags & FormatFlags.TrimLeft)    ret = ret.replace(/^\s+/g, "");
   if(flags & FormatFlags.TrimRight)   ret = ret.replace(/\s+$/g, "");
   if(flags & FormatFlags.UriEncode)   ret = encodeURI(ret);
   return ret;
}

// Combine two enumeration values and store in a FormatFlags variable.
var trim : FormatFlags = FormatFlags.TrimLeft | FormatFlags.TrimRight;
// Combine two enumeration values and store in a byte variable.
var lowerURI : byte = FormatFlags.UriEncode | FormatFlags.ToLowerCase;

var str : String = "  hello, WORLD  ";

print(trim + ": " + Format(str, trim));
print(FormatFlags.ToUpperCase + ": " + Format(str, FormatFlags.ToUpperCase));
print(lowerURI + ": " + Format(str, lowerURI));
```

The output of this code is:

```
12: hello, WORLD
ToUpperCase:   HELLO, WORLD
18: %20%20hello,%20world%20%20
```

Requirements

Version .NET

See Also

Modifiers | Type Conversion | Type Annotation

for Statement

Executes a block of statements for as long as a specified condition is true.

```
for (initialization; test; increment)
...statement
```

Arguments

initialization
Required. An expression. This expression is executed only once, before the loop is executed.

test
Required. A Boolean expression. If *test* is **true**, *statement* is executed. If *test* if **false**, the loop is terminated.

increment
Required. An expression. The increment expression is executed at the end of every pass through the loop.

statement
Optional. Statement to be executed if *test* is **true**. Can be a compound statement.

Remarks

You usually use a **for** loop when the loop is to be executed a known number of times.

Example

The following example demonstrates a **for** loop.

```
/* i is set to 0 at start, and is incremented by 1 at the end
   of each iteration. Loop terminates when i is not less
   than 10 before a loop iteration. */
var myarray = new Array();
for (var i = 0; i < 10; i++) {
   myarray[i] = i;
}
```

Requirements

Version 1

See Also

for...in Statement | while Statement

for...in Statement

Executes one or more statements for each property of an object, or each element of an array or collection.

```
for ( [var] variable in {object | array | collection})
   statement
```

Arguments

variable
Required. A variable that can be any property name of *object*, any index of *array*, or any element of *collection*.

object
A JScript object over which to iterate.

array
An array over which to iterate. Can be a JScript **Array** object or a .NET Framework array.

collection
A collection over which to iterate. Can be any class that implements the **IEnumerable** or **IEnumerator** interfaces from the .NET Framework.

statement
Optional. Statements to be executed for each property of *object* or each element of *array* or *collection*. Can be a compound statement.

Remarks

Before each iteration of a loop, *variable* is assigned the next property name of *object*, the next index of *array*, or the next element of *collection*. You can use *variable* in any of the statements inside the loop to reference the property of *object* or the element of *array*.

When iterating over an object, there is no way to determine or control the order in which the member names of the object are assigned to *variable*. The **for...in** statement cannot loop over the members of non-JScript objects, such as .NET Framework objects.

Arrays are iterated in element order, starting with the smallest index and ending with the largest index. Because JScript **Array** objects can be sparse, the **for...in** statement accesses only the defined elements of the array. JScript **Array** objects may also have expando properties, in which case *variable* is assigned array indexes as property names. If the array is a multidimensional .NET Framework array, only the first dimension is enumerated.

For iteration over a collection, the elements are assigned to *variable* in the order in which they appear in the collection.

Example 1

The following example illustrates the use of the **for ... in** statement with an object used as an associative array.

```
function ForInDemo1() {
   var ret = "";

   // Initialize the object with properties and values.
   var obj : Object = {"a" : "Athens" ,
                       "b" : "Belgrade",
                       "c" : "Cairo"};

   // Iterate over the properties.
   for (var key in obj)
      // Loop and assign 'a', 'b', and 'c' to key.
      ret += key + ":\t" + obj[key] + "\n";

   return(ret);
} // ForInDemo1
```

This function returns the string that contains:

```
a:      Athens
```

```
b:      Belgrade
c:      Cairo
```

Example 2

This example illustrates the use of the **for ... in** statement with a JScript **Array** object which has expando properties.

```
function ForInDemo2() {
   var ret = "";

   // Initialize the array.
   var arr : Array = new Array("zero","one","two");
   // Add a few expando properties to the array.
   arr["orange"] = "fruit";
   arr["carrot"] = "vegetable";

   // Iterate over the properties and elements.
   for (var key in arr)
      // Loop and assign 0, 1, 2, 'orange', and 'carrot' to key.
      ret += key + ":\t" + arr[key] + "\n";

   return(ret);
} // ForInDemo2
```

This function returns the string that contains:

```
0:      zero
1:      one
2:      two
orange: fruit
carrot: vegetable
```

Example 3

The following example illustrates the use of the **for ... in** statement with a collection. Here, the **GetEnumerator** method of the **System.String** object provides a collection of the characters in the string.

```
function ForInDemo3() {
   var ret = "";

   // Initialize collection.
   var str : System.String = "Test.";
   var chars : System.CharEnumerator = str.GetEnumerator();

   // Iterate over the collection elements.
   var i : int = 0;
   for (var elem in chars) {
      // Loop and assign 'T', 'e', 's', 't', and '.' to elem.
      ret += i + ":\t" + elem + "\n";
      i++;
   }

   return(ret);
} // ForInDemo3
```

This function returns the string that contains:

```
0:      T
1:      e
2:      s
3:      t
4:      .
```

Requirements

Version 5

Note Looping over collections requires Version .NET.

See Also

for Statement | while Statement | JScript Arrays | String.GetEnumerator Method

function Statement

Declares a new function. This can be used in several contexts:

Syntax 1: In the global scope

```
function functionname([parmlist]) [: type] {
   [body]
}
```

Syntax 2: Declares a method in a class.

```
[attributes] [modifiers] function functionname([parmlist]) [: type] {
   [body]
}
```

Syntax 3: Declares a method in an interface.

```
[attributes] [modifiers] function functionname([parmlist]) [: type]
```

Arguments

attributes
Optional. Attributes that control the visibility and behavior of the method.

modifiers
Optional. Modifiers that control the visibility and behavior of the method.

functionname
Required. The name of the function or method.

paramlist
Optional. A comma delimited parameter list for the function or method. Each parameter may include a type specification. The last parameter may be a *parameter array*, which is denoted by three periods (**...**) followed by a parameter array name followed by a type annotation of a typed array.

type
Optional. Return type of the method.

body
Optional. One or more statements that define how the function or method operates.

Remarks

Use the **function** statement to declare a function for later use. The code contained in the *body* is not executed until the function is called from elsewhere in the script. The **return** statement is used to return a value from the function. You do not have to use a **return** statement, the program will return when it gets to the end of the function.

Methods are similar to global functions, except that they are scoped to the **class** or **interface** where they are defined and may have various modifiers governing their visibility and behavior. A method in an **interface** cannot have a body, while a method in a **class** must have a body. There is an exception to this rule; if a method in a **class** is **abstract** or the **class** is **abstract**, the method cannot have a body.

You may use type annotation to declare what data type the function or method returns. If **void** is specified as the return type, no value may be returned by any of the **return** statements inside the function. If any return type other than **void** is specified, all **return** statements in the function must return a value that is coercible to the specified return type. The value **undefined** is returned if a return type is specified, but a **return** statement appears with no value or if the end of the function is reached without a **return** statement. Constructor functions cannot specify a return type, since the **new** operator automatically returns the object being created.

If no explicit return type is specified for the function, the return type is set to either **Object** or **void**. The **void** return type is selected only when there are no **return** statements or the **return** statements appear with no value in the function body.

A parameter array can be used as the last the parameter of a function. Any additional arguments passed to the function (if any) after the required parameters will be entered into the parameter array. The type annotation of the parameter is not optional; it must a typed array. To accept parameters of arbitrary types, use **Object[]** as the typed array. When calling a function that can accept a variable number of arguments, an explicit array of the expected type may be used in place of supplying a list of parameters.

When calling a function, make sure that you always include the parentheses and any required arguments. Calling a function without parentheses causes the text of the function to be returned instead of the results of the function.

Example 1

The following example illustrates the use of the **function** statement in the first syntax:

```
interface IForm {
   // This is using function in Syntax 3.
   function blank() : String;
}

class CForm implements IForm {
   // This is using function in Syntax 2.
   function blank() : String {
      return("This is blank.");
   }
}

// This is using function in Syntax 1.
function addSquares(x : double, y : double) : double {
   return(x*x + y*y);
}

// Now call the function.
var z : double = addSquares(3.,4.);
print(z);

// Call the method.
var derivedForm : CForm = new CForm;
print(derivedForm.blank());

// Call the inherited method.
var baseForm : IForm = derivedForm;
print(baseForm.blank());
```

The output from this program is:

```
25
This is blank.
This is blank.
```

Example 2

In this example, a function `printFacts` takes as input a **String** and a used a parameter array to accept a variable number of **Objects**.

```
function printFacts(name : String, ... info : Object[]) {
   print("Name: " + name);
   print("Number of extra information: " + info.length);
   for (var factNum in info) {
      print(factNum + ": " + info[factNum]);
   }
}

// Pass several arguments to the function.
printFacts("HAL 9000", "Urbana, Illinois", new Date(1997,0,12));
// Here the array is intrepeted as containing arguments for the function.
printFacts("monolith", [1, 4, 9]);
// Here the array is just one of the arguments.
printFacts("monolith", [1, 4, 9], "dimensions");
printFacts("monolith", "dimensions are", [1, 4, 9]);
```

This program displays the following output when run:

```
Name: HAL 9000
Number of extra information: 2
0: Urbana, Illinois
1: Sun Jan 12 00:00:00 PST 1997
Name: monolith
Number of extra information: 3
0: 1
1: 4
2: 9
Name: monolith
Number of extra information: 2
0: 1,4,9
1: dimentions
Name: monolith
Number of extra information: 2
0: dimentions are
1: 1,4,9
```

Requirements

Version 1 (for syntax 1)
Version .NET (for syntaxes 2 and 3)

See Also

Modifiers | new Operator | class Statement | interface Statement | return Statement | Scope of Variables and Constants | Type Annotation | Typed Arrays

function get Statement

Declares the accessors for a new property in a class or an interface. Often **function get** will appear in conjunction with a **function set** to allow read/write access to a property.

Syntax for the **get** accessor for a property in a class.

```
[modifiers] function get propertyname() [: type] {
   [body]
}
```

Syntax for the **get** accessor for a property in an interface.

```
[modifiers] function get propertyname() [: type]
```

Arguments

modifiers
Optional. Modifiers that control the visibility and behavior of the property.

propertyname
Required. Name of property being created. Must be unique within the class except the same *propertyname* can be used with both **get** and **set** accessors to identify a property than can be read from and written to.

type
Optional. Return type of the **get** accessor. This must match the parameter type of the **set** accessor, if defined.

body
Optional. One or more statements that define how a **get** accessor operates.

Remarks

The properties of an object are accessed in much the same way as a field is accessed, except that properties allow for more control over the values that are stored in and returned from the object. Properties can be read-only, write-only, or read-write depending on the combination of **get** and **set** property accessors defined within the class. Properties are often used to ensure that only appropriate values are stored in a **private** or **protected** field. You may not assign a value to a read-only property or read a value from a write-only property.

A **get** accessor, which must specify a return type, does not have any arguments. A **get** accessor may be paired with a **set** accessor, which has one argument and does not have a return type. If both accessors are used for a property, the return type of the **get** accessor must match the argument type of the **set** accessor.

A property may have either a **get** accessor or **set** accessor or both. Only the **get** accessor (or **set** accessor if there is no **get** accessor) of a property may have custom attributes that apply to the property as a whole. Both the **get** and **set** accessors can have modifiers and custom attributes that apply to the individual accessor. Property accessors cannot be overloaded, but they can be hidden or overridden.

Properties can be specified in the definition of an **interface**, but no implementation can be given in the interface.

Example

The following example shows several property declarations. An `Age` property is defined as read from and written to. A read-only `FavoriteColor` property is also defined.

```
class CPerson {
   // These variables are not accessible from outside the class.
   private var privateAge : int;
   private var privateFavoriteColor : String;

   // Set the initial favorite color with the constructor.
   function CPerson(inputFavoriteColor : String) {
      privateAge = 0;
      privateFavoriteColor = inputFavoriteColor;
   }
```

```
      // Define an accessor to get the age.
      function get Age() : int {
         return privateAge;
      }
      // Define an accessor to set the age, since ages change.
      function set Age(inputAge : int) {
         privateAge = inputAge;
      }

      // Define an accessor to get the favorite color.
      function get FavoriteColor() : String {
         return privateFavoriteColor;
      }
      // No accessor to set the favorite color, making it read only.
      // This assumes that favorite colors never change.
   }

   var chris: CPerson = new CPerson("red");

   // Set Chris age.
   chris.Age = 27;
   // Read chris age.
   print("Chris is " + chris.Age + " years old.");

   // FavoriteColor can be read from, but not written to.
   print("Favorite color is " + chris.FavoriteColor + ".");
```

When this program is run, it displays the following:

```
Chrisis 27 years old.
Favorite color is red.
```

Requirements

Version .NET

See Also

Modifiers | class Statement | interface Statement | function Statement | function set Statement | Type Annotation

function set Statement

Declares the accessors for a new property in a class or an interface. Often **function set** will appear in conjunction with a **function get** to allow read/write access to a property.

Syntax for the **set** accessor of a property in a class.

```
[modifiers] function set propertyname(parameter [: type]) {
   [body]
}
```

Syntax for the **set** accessor of a property in an interface.

```
[modifiers] function set propertyname(parameter [: type])
```

Arguments

modifiers
Optional. Modifiers that control the visibility and behavior of the property.

propertyname
Required. Name of property being created. Must be unique within the class except the same *propertyname* can be used with both **get** and **set** accessors to identify a property than can be read from and written to.

parameter
Required. Formal parameter accepted by the **set** accessor.

type
Optional. Parameter type of the **set** accessor. This must match the return type of the **get** accessor, if defined.

body
Optional. One or more statements that define how a **set** accessor operates.

Remarks

The properties of an object are accessed in much the same way as a field is accessed, except that properties allow for more control over the values that are stored in and returned from the object. Properties can be read-only, write-only, or read-write depending on the combination of **get** and **set** property accessors defined within the class. Properties are often used to ensure that only appropriate values are stored in a **private** or **protected** a field. You may not assign a value to a read-only property or read a value from a write-only property.

A **set** accessor must have exactly one argument, and it cannot specify a return type. The **set** accessor may be paired with a **get** accessor, which does not have any arguments and must specify a return type. If both accessors are used for a property, the return type of the **get** accessor must match the argument type of the **set** accessor.

A property may have either a **get** accessor or **set** accessor or both. Only the **get** accessor (or **set** accessor if there is no **get** accessor) of a property may have custom attributes that apply to the property as a whole. Both the **get** and **set** accessors can have modifiers and custom attributes that apply to the individual accessor. Property accessors cannot be overloaded, but they can be hidden or overridden.

Properties can be specified in the definition of an **interface**, but no implementation can be given in the interface.

Example

The following example shows several property declarations. An `Age` property is defined as read from and write to. A read-only `FavoriteColor` property is also defined.

```
class CPerson {
   // These variables are not accessible from outside the class.
   private var privateAge : int;
   private var privateFavoriteColor : String;

   // Set the initial favorite color with the constructor.
   function CPerson(inputFavoriteColor : String) {
      privateAge = 0;
      privateFavoriteColor = inputFavoriteColor;
   }

   // Define an accessor to get the age.
   function get Age() : int {
      return privateAge;
   }
   // Define an accessor to set the age, since ages change.
   function set Age(inputAge : int) {
      privateAge = inputAge;
   }

   // Define an accessor to get the favorite color.
   function get FavoriteColor() : String {
      return privateFavoriteColor;
   }
   // No accessor to set the favorite color, making it read only.
   // This assumes that favorite colors never change.
}

var chris : CPerson = new CPerson("red");

// Set Chris's age.
chris.Age = 27;
// Read Chris's age.
print("Chris is " + chris.Age + " years old.");

// FavoriteColor can be read from, but not written to.
print("Favorite color is " + chris.FavoriteColor + ".");
```

When this program is run, it displays the following:

```
Chris is 27 years old.
Favorite color is red.
```

Requirements

Version .NET

See Also

Modifiers | class Statement | interface Statement | function Statement | function get Statement | Type Annotation

@if...@elif...@else...@end Statement

Conditionally executes a group of statements, depending on the value of an expression.

```
@if (
   condition1
)
   text1
[@elif (
   condition2
)
   text2]
[@else
   text3]
@end
```

Arguments

condition1, condition2
  Required. An expression that can be coerced into a Boolean expression.
text1
  Optional. Text to be parsed if *condition1* is **true**.
text2
  Optional. Text to be parsed if *condition1* is **false** and *condition2* is **true**.
text3
  Optional. Text to be parsed if both *condition1* and *condition2* are **false**.

Remarks

When you write an **@if** statement, you do not have to place each clause on a separate line. You can use multiple **@elif** clauses. However, all **@elif** clauses must come before an **@else** clause.

You commonly use the **@if** statement to determine which text among several options should be used for text output.

Example

The following example illustrates the use of the **@if...@else...@end** statement.

```
@if (@_win32)
   print("Operating system is 32-bit.");
@else
   print("Operating system is not 32-bit.");
@end
```

Requirements

Version 3

See Also

Conditional Compilation | Conditional Compilation Variables | @cc_on Statement | @set Statement

if...else Statement

Conditionally executes a group of statements, depending on the value of an expression.

```
if (condition)
   statement1
[else
   statement2]
```

Arguments

condition
Required. A Boolean expression. If *condition* is null or undefined, *condition* is treated as **false**.
statement1
Required. The statement to be executed if *condition* is **true**. Can be a compound statement.
statement2
Optional. The statement to be executed if *condition* is **false**. Can be a compound statement.

Remarks

It is generally good practice to enclose *statement1* and *statement2* in braces ({}) for clarity and to avoid inadvertent errors.

Example

In the following example, you may intend that the **else** be used with the first **if** statement, but it is used with the second one.

```
if (x == 5)
   if (y == 6)
      z = 17;
else
   z = 20;
```

Changing the code in the following manner eliminates any ambiguities:

```
if (x == 5)
   {
   if (y == 6)
      z = 17;
   }
else
   z = 20;
```

Similarly, if you want to add a statement to *statement1*, and you don not use braces, you can accidentally create an error:

```
if (x == 5)
   z = 7;
   q = 42;
else
   z = 19;
```

In this case, there is a syntax error, because there is more than one statement between the **if** and **else** statements. Braces are required around the statements between **if** and **else**.

Requirements

Version 1

See Also

Conditional (Ternary) Operator (?:)

import Statement

Enables access to a namespace contained either within the current script or in an external library.

```
import namespace
```

Arguments

namespace
Required. Name of the namespace to import.

Remarks

The **import** statement creates a property on the global object with the name supplied as *namespace* and initializes it to contain the object that corresponds to the namespace being imported. Any properties created using the **import** statement cannot be assigned to, deleted, or enumerated. All **import** statements are executed when a script starts.

The **import** statement makes a namespace available to your script. The namespace may be defined in the script by using the **package** statement, or an external assembly may provide it. If the namespace is not found within the script, JScript searches for an assembly that matches the name of the namespace in the specified assembly directories, unless the program is being compiled and the /autoref option is turned off. For example, if you import the namespace `Acme.Widget.Sprocket` and the namespace is not defined within the current script, JScript will search for the namespace in the following assemblies:

- `Acme.Widget.Sprocket.dll`
- `Acme.Widget.dll`
- `Acme.dll`

You can explicitly specify the name of the assembly to include. This must be done if the /autoref option is turned off or if the name of the namespace does not match the assembly name. The command line compiler uses the /reference option to specify the assembly name, while ASP.NET uses the **@ Import** and **@ Assembly** directives to accomplish this. For example, to explicitly include the assembly mydll.dll, from the command line you would type

```
jsc /reference:mydll.dll myprogram.js
```

To include the assembly from an ASP.NET page, you would use

```
<%@ Import namespace = "mydll" %>
<%@ Assembly name = "mydll" %>
```

When a class is referenced in code, the compiler first searches for the class in the local scope. If the compiler finds no matching class, the compiler searches for the class in each namespace, in the order in which they were imported, and stops when it finds a match. You can use the fully qualified name of the class to be certain from which namespace the class derives.

JScript does not automatically import nested namespaces; each namespace must be imported using the fully qualified namespace. For example, to access classes from a namespace named `Outer` and a nested namespace named `Outer.Inner`, both namespaces must be imported.

Example

The following example defines three simple packages and imports the namespaces into the script. Typically, each package would be in a separate assembly to allow maintenance and distribution of the package content.

```
// Create a simple package containing a class with a single field (Hello).
package Deutschland {
   class Greeting {
      static var Hello : String = "Guten tag!";
   }
};
// Create another simple package containing two classes.
// The class Greeting has the field Hello.
// The class Units has the field distance.
package France {
   public class Greeting {
      static var Hello : String = "Bonjour!";
   }
   public class Units {
      static var distance : String = "meter";
   }
};
// Use another package for more specific information.
package France.Paris {
   public class Landmark {
      static var Tower : String = "Eiffel Tower";
   }
};

// Declare a local class that shadows the imported classes.
class Greeting {
   static var Hello : String = "Greetings!";
}

// Import the Deutschland, France, and France.Paris packages.
import Deutschland;
import France;
import France.Paris;

// Access the package members with fully qualified names.
print(Greeting.Hello);
print(France.Greeting.Hello);
print(Deutschland.Greeting.Hello);
print(France.Paris.Landmark.Tower);
// The Units class is not shadowed, so it can be accessed with or without a fully qualified n
ame.
print(Units.distance);
print(France.Units.distance);
```

The output of this script is:

```
Greetings!
Bonjour!
Guten tag!
Eiffel Tower
meter
meter
```

Requirements

Version .NET

See Also

package Statement | /autoref | /lib | @ Assembly | @ Import

interface Statement

Declares the name of an interface, as well as the properties and methods that comprise the interface.

```
[modifiers] interface interfacename [implements baseinterfaces] {
   [interfacemembers]
}
```

Arguments

modifiers
Optional. Modifiers that control the visibility and behavior of the property.

interfacename
Required. The name of the **interface**; follows standard variable naming conventions.

implements
Optional. Keyword indicating that the named interface implements, or adds members to, a previously defined interface. If this keyword is not used, a standard JScript base interface is created.

baseinterfaces
Optional. A comma-delimited list of interface names that are implemented by *interfacename*.

interfacemembers
Optional. *interfacemembers* can be either method declarations (defined with the **function** statement) or property declarations (defined with the **function get** and **function set** statements).

Remarks

The syntax for **interface** declarations in JScript is similar to that for **class** declarations. An interface is like a **class** in which every member is **abstract**; it can only contain property and method declarations without function bodies. An **interface** may not contain field declarations, initializer declarations, or nested class declarations. An **interface** can implement one or more **interfaces** by using the **implements** keyword.

A **class** may extend only one base **class**, but a **class** may implement many **interfaces**. Such implementation of multiple **interfaces** by a **class** allows for a form of multiple inheritance that is simpler than in other object-oriented languages, for example, in C++.

Example

The following code shows how one implementation can be inherited by multiple interfaces.

```
interface IFormA {
   function displayName();
}

// Interface IFormB shares a member name with IFormA.
interface IFormB {
   function displayName();
}

// Class CForm implements both interfaces, but only one implementation of
// the method displayName is given, so it is shared by both interfaces and
// the class itself.
class CForm implements IFormA, IFormB {
   function displayName() {
      print("This the form name.");
   }
}

// Three variables with different data types, all referencing the same class.
var c : CForm = new CForm();
var a : IFormA = c;
var b : IFormB = c;

// These do exactly the same thing.
```

```
   a.displayName();
   b.displayName();
   c.displayName();
```

The output of this program is:

```
   This the form name.
   This the form name.
   This the form name.
```

Requirements

Version .NET

See Also

Modifiers | class Statement | function Statement | function get Statement | function set Statement

Labeled Statement

Provides an identifier for a statement.

```
label :
   [statements]
```

Arguments

label
Required. A unique identifier used when referring to the labeled statement.
statements
Optional. One or more statements associated with *label*.

Remarks

Labels are used by the **break** and **continue** statements to specify the statement to which the **break** and **continue** apply.

Example

In the following statement the **continue** statement uses a **labeled** statement to create an array in which the third column of each row contains and undefined value:

```
function labelDemo() {
   var a = new Array();
   var i, j, s = "", s1 = "";
   Outer:
   for (i = 0; i < 5; i++) {
      Inner:
      for (j = 0; j < 5; j++) {
         if (j == 2)
            continue Inner;
         else
            a[i,j] = j + 1;
      }
   }
   for (i = 0;i < 5; i++) {
      s = ""
      for (j = 0; j < 5; j++) {
         s += a[i,j];
      }
      s1 += s + "\n";
   }
   return(s1)
}
```

Requirements

Version 3

See Also

break Statement | continue Statement

package Statement

Creates a JScript package that enables the convenient packaging of named components.

```
package pname {
   [[modifiers1] pmember1]
   ...
   [[modifiersN] pmemberN]
}
```

Arguments

pname
Required. The name of the package being created.
modifiers1, ..., modifiersN
Optional. Modifiers that control the visibility and behavior of the *pmember*.
pmember1, ..., pmemberN
Optional. Class, interface, or enumeration definition.

Remarks

Only classes, interfaces, and enumerations are allowed inside a package. Package members may be marked with visibility modifiers to control access to the member. In particular, the **internal** modifier marks a member as being visible only within the current package.

Once a package has been imported, package members can be accessed directly by name, except when a member has the same name as another declaration visible to the importing scope. When that happens, the member must be qualified using its package name.

JScript does not support declaring nested packages; only class, interface, and enumeration declarations may appear inside a package. A package name can include a '.' character to indicate that it should be considered as nested in another package. For example, a package named `Outer` and a package named `Outer.Inner` do not need to have a special relationship to each other; they are both packages at the global scope. However, the names imply that `Outer.Inner` should be considered as nested within `Outer`.

Example

The following example defines three simple packages and imports the namespaces into the script. Typically, each package would be in a separate assembly to allow maintenance and distribution of the package content.

```
// Create a simple package containing a class with a single field (Hello).
package Deutschland {
   class Greeting {
      static var Hello : String = "Guten tag!";
   }
};
// Create another simple package containing two classes.
// The class Greeting has the field Hello.
// The class Units has the field distance.
package France {
   public class Greeting {
      static var Hello : String = "Bonjour!";
   }
   public class Units {
      static var distance : String = "meter";
   }
};
// Use another package for more specific information.
package France.Paris {
   public class Landmark {
      static var Tower : String = "Eiffel Tower";
   }
};
```

```
// Declare a local class that shadows the imported classes.
class Greeting {
   static var Hello : String = "Greetings!";
}

// Import the Deutschland, France, and France.Paris packages.
import Deutschland;
import France;
import France.Paris;

// Access the package members with fully qualified names.
print(Greeting.Hello);
print(France.Greeting.Hello);
print(Deutschland.Greeting.Hello);
print(France.Paris.Landmark.Tower);
// The Units class is not shadowed, so it can be accessed with or without a fully qualified n
ame.
print(Units.distance);
print(France.Units.distance);
```

The output of this script is:

```
Greetings!
Bonjour!
Guten tag!
Eiffel Tower
meter
meter
```

Requirements

Version .NET

See Also

import Statement | internal Modifier | Modifiers

print Statement

Sends a string to the console followed by a newline character.

```
function print(str : String)
```

Parameters

str

Optional. String to send to the console.

Remarks

The **print** statement allows you to display data from a JScript program compiled with the JScript command-line compiler, `jsc.exe`. The **print** statement takes a single string as a parameter and displays that string followed by a newline character by sending it to the console.

You can use escape sequences in the strings you pass to the **print** statement to format the output. Escape sequences are character combinations consisting of a backslash (****) followed by a letter or by a combination of digits. Escape sequences can be used to specify actions such as carriage returns and tab movement. More information about escape characters can be found in the **String** object topic. The **System.Console.WriteLine** method can be used when fine control over the format of console output is required.

The **print** statement is enabled by default in the JScript command-line compiler, jsc.exe. The **print** statement is disabled in ASP.NET, and you can disable it for the command-line compiler by using the /print- option.

When there is no console to which to print (for example, in a Windows GUI application), the **print** statement will silently fail.

Output from the **print** statement can be redirected to a file from the command line. If you expect that the output of a program will be redirected, you should include the \r escape character at the end of each line printed. This causes output redirected to a file to be correctly formatted, and it does not affect the way lines are displayed on the console.

Example

The following example demonstrates a use of the print statement.

```
var name : String = "Fred";
var age : int = 42;
// Use the \t (tab) and \n (newline) escape sequences to format the output.
print("Name: \t" + name + "\nAge: \t" + age);
```

The output of this script is:

```
Name:   Fred
Age:    42
```

See Also

/print | Displaying from a Command Line Program | String Object | Console Class

return Statement

Exits from the current function and returns a value from that function.

```
return[(][expression][)]
```

Arguments

expression
Optional. The value to be returned from the function. If omitted, the function does not return a value.

Remarks

You use the **return** statement to stop execution of a function and return the value of *expression*. If *expression* is omitted, or no **return** statement is executed from within the function, the expression that called the current function is assigned the value **undefined**.

Execution of the function stops when the **return** statement is executed, even if there are other statements still remaining in the function body. The exception to this rule is if the **return** statement occurs within a **try** block, and there is a corresponding **finally** block, the code in the **finally** block will execute before the function returns.

If a function returns because it reaches the end of the function body without executing a **return** statement, the value returned is the **undefined** value (this means the function result cannot be used as part of a larger expression).

> **Note** The code in a **finally** block is run after a **return** statement in a **try** or **catch** block is encountered, but before that **return** statement is executed. In this situation, a **return** statement in the **finally** block is executed *before* the initial **return** statement, allowing for a different return value. To avoid this potentially confusing situation, do not use a **return** statement in a **finally** block.

Example

The following example illustrates the use of the **return** statement.

```
function myfunction(arg1, arg2){
   var r;
   r = arg1 * arg2;
   return(r);
}
```

Requirements

Version 1

See Also

function Statement | try...catch...finally Statement

@set Statement

Creates variables used with conditional compilation statements.

```
@set @varname = term
```

Arguments

varname
Required. Valid JScript variable name. Must be preceded by an "@" character at all times.

term
Required. Zero or more unary operators followed by a constant, conditional compilation variable, or parenthesized expression.

Remarks

Numeric and Boolean variables are supported for conditional compilation. Strings are not. Variables created using **@set** are generally used in conditional compilation statements, but can be used anywhere in JScript code.

Examples of variable declarations look like this:

```
@set @myvar1 = 12
@set @myvar2 = (@myvar1 * 20)
@set @myvar3 = @_jscript_version
```

The following operators are supported in parenthesized expressions:

- `! ~`
- `* / %`
- `+ -`
- `<< >> >>>`
- `< <= > >=`
- `== != === !==`
- `& ^ |`
- `&& ||`

If a variable is used before it has been defined, its value is **NaN**. **NaN** can be checked for using the **@if** statement:

```
@if (@newVar != @newVar)
   // ...
```

This works because **NaN** is the only value not equal to itself.

Requirements

Version 3

See Also

Conditional Compilation | Conditional Compilation Variables | @cc_on Statement | @if Statement

static Statement

Declares a new class initializer inside a class declaration.

```
static identifier {
   [body]
}
```

Arguments

identifier
Required. The name of the class that contains the initializer block.

body
Optional. The code that comprises the initializer block.

Remarks

A **static** initializer is used to initialize a **class** object (not object instances) before its first use. This initialization occurs only once, and it can be used to initialize fields in the class that have the **static** modifier.

A **class** may contain several **static** initializer blocks interspersed with **static** field declarations. To initialize the **class**, all the **static** blocks and **static** field initializers are executed in the order in which they appear in the **class** body. This initialization is performed before the first reference to a **static** field.

Do not confuse the **static** modifier with the **static** statement. The **static** modifier denotes a member that belongs to the class itself, not any instance of the class.

Example

The following example shows a simple **class** declaration in which the **static** initializer is used to perform a calculation that only needs to be done one time. In this example, a table of factorials is calculated once. When factorials are needed they are read from the table. This approach is faster than calculating factorials recursively if large factorials are needed many times in the program.

The **static** modifier is used for the factorial method.

```
class CMath {
   // Dimension an array to store factorial values.
   // The static modifier is used in the next two lines.
   static const maxFactorial : int = 5;
   static const factorialArray : int[] = new int[maxFactorial];

   static CMath {
      // Initialize the array of factorial values.
      // Use factorialArray[x] = (x+1)!
      factorialArray[0] = 1;
      for(var i : int = 1; i< maxFactorial; i++) {
         factorialArray[i] = factorialArray[i-1] * (i+1);
      }
      // Show when the initializer is run.
      print("Initialized factorialArray.");
   }

   static function factorial(x : int) : int {
      // Should have code to check that x is in range.
      return factorialArray[x-1];
   }
};

print("Table of factorials:");

for(var x : int = 1; x <= CMath.maxFactorial; x++) {
   print( x + "! = " + CMath.factorial(x) );
}
```

The output of this code is:

```
Table of factorials:
Initialized factorialArray.
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
```

Requirements

Version .NET

See Also

class Statement | static Modifier

super Statement

Refers to the base object of the current object. This can be used in two contexts.

Syntax 1: Calls the base-class constructor with arguments.

```
super(arguments)
```

Syntax 2: Accesses a member of the base class.

```
super.member
```

Arguments

arguments
  Optional in syntax 1. A comma delimited list of arguments for the base-class constructor.
member
  Required in syntax 2. Member of the base class to access.

Remarks

The **super** keyword is typically used in one of two situations. You can use it to explicitly call the base-class constructor with one or more arguments. You can also use it to access base-class members that have been overridden by the current class.

Example 1

In the following example, **super** refers to the constructor of the base class.

```
class baseClass {
   function baseClass() {
      print("Base class constructor with no parameters.");
   }
   function baseClass(i : int) {
      print("Base class constructor. i is "+i);
   }
}
class derivedClass extends baseClass {
   function derivedClass() {
      // The super constructor with no arguments is implicitly called here.
      print("This is the derived class constructor.");
   }
   function derivedClass(i : int) {
      super(i);
      print("This is the derived class constructor.");
   }
}

new derivedClass;
new derivedClass(42);
```

This program displays the following output when run.

```
Base class constructor with no parameters.
This is the derived class constructor.
Base class constructor. i is 42
This is the derived class constructor.
```

Example 2

In the following example, **super** allows access to an overridden member of the base class.

```
class baseClass {
   function test() {
      print("This is the base class test.");
   }
}
class derivedClass extends baseClass {
   function test() {
      print("This is the derived class test.");
      super.test(); // Call the base class test.
   }
}

var obj : derivedClass = new derivedClass;
obj.test();
```

This program displays the following output when run.

```
This is the derived class test.
This is the base class test.
```

Requirements

Version .NET

See Also

new Operator | this Statement

switch Statement

Enables the execution of one or more statements when a specified expression's value matches a label.

```
switch (expression) {
   case label1 :
      [statementlist1]
      [break;]
 [ ...
 [ case labelN :
      [statementlistN]
      [break;] ] ]
 [ default :
      [statementlistDefault]]
}
```

Arguments

expression
Required. The expression to be evaluated.

label1, ..., labelN
Required. An identifier to be matched against *expression*. If *label* === *expression*, execution starts with the statement list immediately after the colon, and continues until it encounters either a **break** statement, which is optional, or the end of the **switch** statement.

statementlist1, ..., statementlistN, statementlistDefault
Optional. One or more statements to be executed.

Remarks

Use the **default** clause to provide a statement to be executed if none of the label values matches *expression*. It can appear anywhere within the **switch** code block.

Zero or more *label* blocks may be specified. If no *label* matches the value of *expression*, and a **default** case is not supplied, no statements are executed.

Execution flows through a switch statement as follows:

- Evaluate *expression* and look at *label* in order until a match is found.
- If a *label* value equals *expression*, execute its accompanying statement list.
  Continue execution until a **break** statement is encountered, or the **switch** statement ends. This means that multiple *label* blocks are executed if a **break** statement is not used.
- If no *label* equals *expression*, go to the **default** case. If there is no **default** case, go to last step.
- Continue execution at the statement following the end of the **switch** code block.

Example

The following ASP.NET example tests an object for its type. In this case, only one type is used, but you should be able to clearly see how the function works with other object types.

```
<%@ language="jscript" %>
<%
var d = new Number();
function MyObjectType(obj : Object) : String {
   switch (obj.constructor){
      case Date:
         return "Object is a Date.";
         break;
      case Number:
         return "Object is a Number.";
         break;
      case String:
         return "Object is a String.";
```

```
            break;
         default:
            return "Object is unknown.";
      }
   }
   Response.Write(MyObjectType(d));
   %>
```

Requirements

Version 3

See Also

break Statement | if...else Statement

this Statement

Refers to the current object.

```
this.property
```

Arguments

property
Required. The identifier of a property of the current object.

Remarks

The **this** keyword is typically used in object constructors to refer to the current object.

Example

In the following example, **this** refers to the newly created Car object, and assigns values to three properties:

```
function Car(color, make, model){
   this.color = color;
   this.make = make;
   this.model = model;
}
```

For client versions of JScript, **this** refers to the **window** object if used outside of the context of any other object.

Requirements

Version 1

See Also

new Operator

throw Statement

Generates an error condition that can be handled by a **try...catch...finally** statement.

```
throw [exception]
```

Arguments

exception
Optional. Any expression.

Remarks

The **throw** statement can be used without an argument, but only if the **throw** statement is contained within a **catch** block. In that situation, the **throw** statement re-throws the error caught by the enclosing **catch** statement. When an argument is provided, the **throw** statement throws the value of *exception*.

Example

The following example throws an error based on a passed-in value, then illustrates how that error is handled in a hierarchy of **try...catch...finally** statements:

```
function TryCatchDemo(x){
   try {
      try {
      if (x == 0)                            // Evalute argument.
         throw "x equals zero";              // Throw an error.
      else
         throw "x does not equal zero";      // Throw a different error.
      }
      catch(e) {                             // Handle "x=0" errors here.
         if (e == "x equals zero")           // Check for a handled error.
            return(e + " handled locally."); // Return error message.
         else                                // Can't handle error here.
            throw e;                         // Rethrow the error for next
      }                                      // error handler.
   }
   catch(e) {                                // Handle other errors here.
      return(e + " error handled higher up."); // Return error message.
   }
}
print(TryCatchDemo(0)+ "\n");
print(TryCatchDemo(1));
```

Requirements

Version 5

See Also

try...catch...finally Statement | Error Object

try...catch...finally Statement

Implements error handling for JScript.

```
try {
   [tryStatements]
} catch(exception) {
   [catchStatements]
} finally {
   [finallyStatements]}
```

Arguments

tryStatements
Optional. Statements where an error can occur.
exception
Required. Any variable name. The initial value of *exception* is the value of the thrown error.
catchStatements
Optional. Statements to handle errors occurring in the associated *tryStatements*.
finallyStatements
Optional. Statements that are unconditionally executed after all other error processing has occurred.

Remarks

The **try...catch...finally** statement provides a way to handle some or all of the possible errors that may occur in a given block of code, while still running code. If errors occur that the programmer has not handled, JScript simply provides its normal error message to a user, as if there was no error handling.

The *tryStatements* contain code where an error can occur, while *catchStatements* contain the code to handle any error that does occur. If an error occurs in the *tryStatements*, program control is passed to *catchStatements* for processing. The initial value of *exception* is the value of the error that occurred in *tryStatements*. If no error occurs, *catchStatements* are never executed.

If the error cannot be handled in the *catchStatements* associated with the *tryStatements* where the error occurred, use the **throw** statement to propagate, or re-throw, the error to a higher-level error handler.

After all statements in *tryStatements* have been executed and any error handling has occurred in *catchStatements*, the statements in *finallyStatements* are unconditionally executed.

Notice that the code inside *finallyStatements* is executed even if a return statement occurs inside the **try** or **catch** blocks, or if an error is thrown from a **catch** block. *finallyStatments* are guaranteed to always run.

Example

The following example shows how JScript exception handling works.

```
try {
   print("Outer try running...");
   try {
      print("Nested try running...");
      throw "an error";
   } catch(e) {
      print("Nested catch caught " + e);
      throw e + " re-thrown";
   } finally {
      print("Nested finally is running...");
   }
} catch(e) {
   print("Outer catch caught " + e);
} finally {
   print("Outer finally running");
}
```

This produces the following output:

```
Outer try running..
Nested try running...
Nested catch caught an error
Nested finally is running...
Outer catch caught an error re-thrown
Outer finally running
```

Requirements

Version 5

See Also

throw Statement | Error Object

var Statement

Declares a variable.

Syntax 1

Syntax for declaring a variable of global scope or function scope:

```
var name1 [: type1] [= value1] [, ... [, nameN [: typeN] [= valueN] ]]
```

Syntax 2

Syntax for declaring a variable field within a class:

```
[attributes] [modifiers] var name1 [: type1] [= value1] [, ... [, nameN [: typeN] [= valueN].
]]
```

Arguments

attributes
Optional. Attributes that control the visibility and behavior of the field.
modifiers
Optional. Modifiers that control the visibility and behavior of the field.
name1, ..., nameN
Required. The names of the variables being declared.
type1, ..., typeN
Optional. The types of the variables being declared.
value1, ..., valueN
Optional. The initial value assigned to the variable.

Remarks

Use the **var** statement to declare variables. A variable may be bound to a specific data type to ensure type safety. These variables may be assigned values when they are declared, and these values may be changed later in the script. Variables that are not explicitly initialized are assigned the default value of **undefined** (coerced to the type of the variable if necessary).

A variable field in a class is similar to a global or function variable, except that it is scoped to the class and it can have various attributes governing its visibility and usage.

Example

The following example illustrates some uses of the **var** statement.

```
class Simple {
   // A field declaration of the private Object myField.
   private var myField : Object;
   // Define sharedField to be a static, public field.
   // Only one copy exists, and is shared by all instances of the class.
   static public var sharedField : int = 42;
}
var index;
var name : String = "Thomas Jefferson";
var answer : int = 42, counter, numpages = 10;
var simpleInst : Simple = new Simple;
```

Requirements

Version 1

See Also

Modifiers | const Statement | function Statement | new Operator | Scope of Variables and Constants | Type Annotation

while Statement

Executes a statement until a specified condition is **false**.

```
while (expression)
   statement
```

Arguments

expression
Required. A Boolean expression checked before each iteration of the loop. If *expression* is **true**, the loop is executed. If *expression* is **false**, the loop is terminated.

statement
Required. Statement to be executed if *expression* is **true**. Can be a compound statement.

Remarks

The **while** statement checks *expression* before a loop is first executed. If *expression* is **false** at this time, the loop is never executed.

Example

The following example illustrates the use of the **while** statement.

```
function BreakTest(breakpoint){
   var i = 0;
   while (i < 100) {
      if (i == breakpoint)
         break;
      i++;
   }
   return(i);
}
```

Requirements

Version 1

See Also

break Statement | continue Statement | do...while Statement | for Statement | for...in Statement

with Statement

Establishes the default object for a statement.

```
with (object)
   statement
```

Arguments

object
Required. The new default object.

statement
Required. Statements for which *object* is the default object. Can be a compound statement.

Remarks

The **with** statement is commonly used to shorten the amount of code that you have to write in certain situations.

Example

In the example that follows, notice the repeated use of **Math**.

```
var x, y;
x = Math.cos(3 * Math.PI) + Math.sin(Math.LN10);
y = Math.tan(14 * Math.E);
```

When you use the **with** statement, your code becomes shorter and easier to read:

```
var x, y;
with (Math){
   x = cos(3 * PI) + sin (LN10);
   y = tan(14 * E);
}
```

Requirements

Version 1

See Also

this Statement

JScript Compiler Options

The JScript compiler produces executable (.exe) files and dynamic-link libraries (.dll).

Every compiler option is available in two forms: **-***option* and **/***option*. The documentation only provides the **/***option* form.

In This Section

JScript Compiler Options Listed Alphabetically
Provides a list of compiler options that are listed in ascending alphabetical order.

JScript Compiler Options Listed by Category
Provides a list of compiler options that are listed in the following categories: output files, .NET Framework assemblies, debugging/error checking, preprocessor, resources, and miscellaneous.

Related Sections

Building from the Command Line
Explains details, such as syntax and results, about building a JScript application from the command line.

Writing, Compiling, and Debugging JScript Code
Explains how to use the Visual Studio .NET Integrated Development Environment (IDE) to write and edit JScript code.

JScript Compiler Options Listed Alphabetically

The following compiler options are sorted alphabetically.

Option	Purpose
@ (Specify Response File)	Specifies a response file.
/autoref	Automatically references assemblies if they have the same name as an imported namespace or as a type annotation when declaring a variable.
/codepage	Specifies the code page to use for all source code files in the compilation.
/debug	Emits debugging information.
/define	Defines preprocessor symbols.
/fast	Produces an output file optimized for speed but that does not support certain language features from previous releases.
/help, /?	Lists compiler options to stdout.
/lcid	Specifies code page for compiler messages.
/lib	Specifies the location of assemblies referenced via /reference.
/linkresource	Creates a link to a managed resource.
/nologo	Suppresses compiler banner information.
/nostdlib	Does not import standard library (mscorlib.dll).
/out	Specifies output file name.
/print	Specifies whether the print statement is available.
/reference	Imports metadata from a file that contains an assembly.
/resource	Embeds a managed resource in an assembly.
/target	Specifies the format of the output file using one of three options: /target:exe /target:library /target:winexe
/utf8output	Displays compiler output using UTF-8 encoding.
/versionsafe	Ensures that all overrides are explicit.
/warn	Sets warning level.
/warnaserror	Promotes warnings to errors.
/win32res	Inserts a Win32 resource into the output file.

See Also

JScript Compiler Options | JScript Compiler Options Listed by Category | Building from the Command Line

JScript Compiler Options Listed by Category

The following compiler options are sorted by category.

Output Files

Option	Purpose
/out	Specifies output file name.
/target	Specifies the format of the output file using one of three options : /target:exe /target:library /target:winexe.

.NET Framework Assemblies

Option	Purpose
/autoref	Automatically references assemblies if they have the same name as an imported namespace or as a type annotation when declaring a variable.
/lib	Specifies the location of assemblies referenced via /reference.
/nostdlib	Does not import standard library (mscorlib.dll).
/reference	Imports metadata from a file that contains an assembly.

Debugging/Error Checking

Option	Purpose
/debug	Emits debugging information.
/lcid	Specifies code page for compiler messages .
/versionsafe	Ensures that all overrides are explicit.
/warn	Sets warning level.
/warnaserror	Promotes warnings to errors.

Preprocessor

Option	Purpose
/define	Defines preprocessor symbols .

Resources

Option	Purpose
/linkresource	Creates a link to a managed resource.
/resource	Embeds a managed resource in an assembly .
/win32res	Inserts a Win32 resource into the output file.

Miscellaneous

Option	Purpose
@ (Specify Response File)	Specifies a response file.
/codepage	Specifies the code page to use for all source code files in the compilation.
/fast	Produces an output file optimized for speed but that does not support certain language features from previous releases.
/help, /?	Lists compiler options to stdout.
/nologo	Suppresses compiler banner information.
/print	Specifies whether the print statement shall be defined.
/utf8output	Displays compiler output using UTF-8 encoding.

See Also

JScript Compiler Options | JScript Compiler Options Listed Alphabetically | Building from the Command Line

Building from the Command Line

The compiler can be started at the command line by typing the name of its executable (jsc.exe) on the command line. For more information, see Compiling JScript Code from the Command Line.

Sample Command Lines

- Compiles File.js producing File.exe:

  ```
  jsc File.js
  ```

- Compiles File.js producing File.dll:

  ```
  jsc /target:library File.js
  ```

- Compiles File.js and creates My.exe:

  ```
  jsc /out:My.exe File.js
  ```

- Compile `test.js` and create a .dll:

  ```
  jsc /target:library test.js
  ```

See Also

JScript Compiler Options | Compiling JScript Code from the Command Line

@ (Specify Response File)

Specifies a response file.

```
@response_file
```

Arguments

response_file
A file that lists compiler options or source code files to compile.

Remarks

The @ option lets you specify a file that contains compiler options and source code files to compile. These compiler options and source code files will be processed by the compiler just as if set out on the command line.

To specify more than one response file in a compilation, specify multiple response file options. For example:

```
@file1.rsp @file2.rsp
```

In a response file, multiple compiler options and source code files can appear on one line. A single compiler option specification must appear on one line (cannot span multiple lines).

Response files can have comments that begin with the # symbol.

Specifying compiler options from within a response file is just like making those commands on the command line. For more information, see Building from the Command Line.

The compiler processes command options as they are encountered, just as if set out on the command line. Therefore, the options in one response file may be incompatible with the options in another response file or the command line options. This can generate errors.

Response files cannot be nested. You cannot place *@response_file* inside a response file. The JScript compiler reports an error for such cases.

Example

The following are a few lines from a sample response file:

```
# build the first output file
/target:exe /out:MyExe.exe source1.js source2.js
```

See Also

JScript Compiler Options

/autoref

Automatically references assemblies if they have the same name as an imported namespace or as a type annotation when declaring a variable.

```
/autoref[+ | -]
```

Arguments

+ | -
On by default, unless /nostdlib+ is specified. Specifying /autoref+, or just /autoref, causes the compiler to automatically reference assemblies based on imported namespaces and fully qualified names.

Remarks

The /autoref option instructs the compiler to reference assemblies without having to pass the assembly to /reference. When you use import to import a namespace, or you use a fully qualified type name in your code, the JScript compiler searches for an assembly that contains the type. See /lib for a discussion of how the JScript compiler searches for assemblies.

The compiler does not try to reference an assembly if it has the same name as the output file of the program you are building.

Example

The following program will compile and run when /autoref+ is in effect; the compiler will reference System.dll as a result of the type annotation when declaring a variable.

```
var s: System.Collections.Specialized.StringCollection =
                   new System.Collections.Specialized.StringCollection();
print(s);
```

The following program will compile and run when /autoref+ is in effect; the compiler will reference System.dll as a result of the **import** statement.

```
import System;
var s = new System.Collections.Specialized.StringCollection();
print(s);
```

These examples also show how the compiler looks for assembly names based on type annotation or **import** statements. When the compiler did not find an assembly called System.Collections.Specialized.dll that contained StringCollection, it looked for System.Collections.dll. Failing to find that file, it looked for System.dll, which it did find to contain StringCollection.

See Also

import Statement | /reference | JScript Compiler Options

/codepage

Specifies the code page to use for all source code files in the compilation.

```
/codepage:id
```

Arguments

id

The id of the code page for all the source code files in the compilation.

Remarks

If you compile one or more source code files that when created did not designate use of the default code page on your computer, you can use the /codepage option to specify which code page should be used. /codepage applies to all source code files in your compilation.

If the source code files were created with the same codepage that is in effect on your computer or if the source code files were created with UNICODE or UTF-8, you need not use /codepage.

See Also

JScript Compiler Options

/debug

Emits debugging information.

```
/debug[+ | -]
```

Arguments

+ | -
Specifying /debug+, or just /debug, causes the compiler to generate debugging information and place it in an output .pdb file(s). /debug- which is in effect by default if you do not specify /debug does not generate any debug information or create an output file(s) to contain debug information.

Remarks

For information on how to configure the debug performance of an application, see Making an Image Easier to Debug.

Example

Place debugging information in output file `app.exe`:

```
jsc /debug /out:app.exe test.js
```

See Also

JScript Compiler Options

/define

Defines preprocessor symbols.

```
/define:name1[=value1][,name2[=value1]]
```

Arguments

name1, *name2*
The name of one or more symbols that you want to define.
value1, *value2*
Values for the symbols to take. These can be booleans or numbers.

Remarks

The /define option defines names as symbols in your program.

You can define multiple symbols with /define by using a comma to separate symbol names. For example:

```
/define:DEBUG,trace=true,max_Num=100
```

See Conditional Compilation for more information.

/d is the short form of /define.

Example

Compile with /define:xx.

```
print("testing")
/*@cc_on @*/
/*@if (@xx)
print("xx defined")
@else @*/
print("xx not defined")
/*@end @*/
```

See Also

JScript Compiler Options

/fast

Enables faster program execution.

```
/fast[+ | -]
```

Arguments

+ | -

/fast is on by default. /fast or /fast+ causes the compiler to generate an output file that is speed-optimized, however, if this option is used certain language features from previous versions will not be supported. Specifying /fast-, on the other hand, will provide for backward language compatibility, but the compiler will produce an output file that is not optimized for speed.

Remarks

When /fast is in effect,

- All variables must be declared.
- Functions become constants and you cannot assign to them or redefine them.
- Predefined properties of built-in objects are marked DontEnum, DontDelete, ReadOnly.
- Properties on the built-in objects may not be expanded, other than the Global object (which is also the global scope).
- The **arguments** variable is not available within function calls.
- Assignments to read-only variables, fields, or methods generate errors.

> **Note** The /fast- compilation mode is provided to help developers build standalone executables from legacy JScript code. When developing new executables or libraries, use the /fast+ compilation mode. This ensures better performance and better compatibility with other assemblies.
>
> **Security Note** The /fast- compilation mode enables the use of language features from previous versions not available in /fast+ mode. Misuse of these features can result in a less secure program. For more information, see Security Considerations for JScript.

Example

Create an output file that is speed-optimized at the expense of full backward language compatibility :

```
jsc test.js
```

See Also

JScript Compiler Options | Security Considerations for JScript

/help, /?

Displays compiler command-line help.

```
/help
```

-or-

```
/?
```

Remarks

This option causes the compiler to display a list of compiler options along with a brief description of each option.

See Also

JScript Compiler Options

/lcid

Specifies code page for compiler messages.

```
/lcid:id
```

Arguments

id

The id of the code page to use for printing out messages from the compiler.

See Also

JScript Compiler Options

/lib

Specifies assembly reference locations.

```
/lib:dir1[, dir2]
```

Arguments

dir1
A directory for the compiler to look in if a referenced assembly is not found in the current working directory (the directory from which the compiler is invoked) or in the common language runtime's system directory.
dir2
One or more additional directories for searching for assembly references. Separate additional directory names with a comma or semicolon.

Remarks

The /lib option specifies the location of assemblies referenced via the /reference option.

The compiler searches for assembly references that are not fully qualified in the following order:

1. Current working directory. This is the directory from which the compiler is invoked.
2. The common language runtime system directory.
3. Directories specified by /lib.
4. Directories specified by the LIB environment variable.

Use /reference to specify an assembly reference.

/lib is additive; specifying it more than once appends to any prior values.

Example

Compile t2.js to create an .exe. The compiler will look in the working directory and in the root directory of the C drive for assembly references.

```
jsc /lib:c:\ /reference:t2.dll t2.js
```

See Also

JScript Compiler Options

/linkresource

Creates a link to a managed resource.

```
/linkresource:filename[,name[,public|private]]
```

-or-

```
/linkres:filename[,name[,public|private]]
```

Arguments

filename
The resource file to link to the assembly.
name[,public|private] (optional)
The logical name for the resource; the name used to load the resource. The default is the name of the file. Optionally, you can specify whether the file is public or private in the assembly manifest. For example, /linkres:filename.res,myname.res,public. By default, *filename* is public in the assembly.

Remarks

The /linkresource option does not embed the resource file in the output file. Use the /resource option to embed a resource file in the output file.

If *filename* is a .NET Framework resource file created, for example, by the Resource File Generator (Resgen.exe) or in the development environment, it can be accessed with members in the System.Resources namespace (see System.Resources.ResourceManager for more information). For all other resources, use the GetManifestResource* methods in System.Reflection.Assembly class to access the resource at run time.

filename can be any file format. For example, you may want to make a native DLL part of the assembly, so it can be installed into the Global Assembly Cache and accessed from managed code in the assembly.

/linkres is the short form of /linkresource.

Example

Compile `in.js` and link to resource file `rf.resource`:

```
jsc /linkresource:rf.resource in.js
```

See Also

JScript Compiler Options

/nologo

Suppresses banner information.

```
/nologo
```

Remarks

The /nologo option suppresses display of the banner when the compiler starts.

See Also

JScript Compiler Options

/nostdlib

Does not import a standard library.

```
/nostdlib[+ | -]
```

Arguments

+ | -
/nostdlib or /nostdlib+ option causes the compiler to not import mscorlib.dll. Use this option if you want to define or create your own System namespace and objects. If you do not specify /nostdlib, mscorlib.dll will be imported into your program (same as specifying /nostdlib-).

Remarks

Specifying /nostdlib+ also specifies /autoref-.

Example

If you have a component called System. String (or any other name in mscorlib) the only way you could get at your component would be to use

```
/nostdlib /r:your_library,mscorlib
```

to search your library before mscorlib. Commonly, you would not a define a namespace in your application called System.

See Also

JScript Compiler Options

/out

Sets output filename.

```
/out:filename
```

Arguments

filename
The name of the output file created by the compiler.

Remarks

The /out option specifies the name of the output file. The compiler expects to find one or more source code files following the /out option.

If you do not specify the name of the output file:

- An .exe will take its name from the first source code file used to build the output file.
- A .dll will take its name from the first source code file used to build the output file.

On the command line, it is possible to specify multiple output files for a compilation. All source code files specified after an /out option will be compiled into the output file specified by that /out option.

Specify the full name and extension of the file you want to create. The extension must be either .exe or .dll. It is possible to specify a .dll extension for /t:exe projects.

Example

Compile `t2.js` and create output file `t2.exe` and build `t3.js` and create output file `t3.exe`:

```
jsc t2.js /out:t3.exe t3.js
```

See Also

JScript Compiler Options

/print

Enables print command.

```
/print[+ | -]
```

Arguments

+ | -

By default, /print or /print+ causes the compiler to enable the use of the print statement. An example of a print statement is,

```
print("hello world");
```

Specifying /print- will disable the print command.

Remarks

You would use /print- if your .dll will be loaded into an environment that doesn't have a console.

You can set Microsoft.JScript.ScriptStream.Out to be an instance of a TextWriter object to enable print to send output elsewhere.

Example

Cause the print statement to not be defined by the compiler:

```
jsc /print- test.js
```

See Also

JScript Compiler Options

/reference

Imports metadata.

```
/reference:file[;file2]
```

Arguments

file, file2

One or more files that contains an assembly manifest. To import more than one file, separate file names with either a comma or a semicolon.

Remarks

The /reference option directs the compiler to make public type information in the specified files available to the project you are currently compiling.

The file(s) you reference must be assemblies. For example, the referenced files must have been created with the /target:library compiler option in Visual C#, JScript or Visual Basic, or the /clr /LD compiler options of Visual C++.

/reference cannot take a module as input.

At run time, you should anticipate that only one .exe assembly can be loaded per process, even though, there may be times when more than one .exe might be loaded in the same process.. Therefore, you are recommended to not pass an assembly built with /target:exe or /target:winexe to /reference if you are compiling with /target:winexe or /target:exe. This condition may be modified in future versions of the common language runtime.

If you reference an assembly (Assembly A), which itself references another assembly (Assembly B), you will need to reference assembly B if:

- A type you use from Assembly A inherits from a type or implements an interface from Assembly B.
- If you invoke a field, property, event, or method that has a return type or parameter type from Assembly B.

Use /lib to specify the directory in which one or more of your assembly references is located.

In order for the compiler to recognize a type in an assembly (not a module), it needs to be forced to resolve the type, which you can do, for example, by defining an instance of the type. There are other ways to resolve type names in an assembly for the compiler, for example, if you inherit from a type in an assembly, the type name will then become known to the compiler.

/r is the short form of /reference.

> **Note** The JScript .NET compiler, jsc.exe, can reference assemblies created using the same version or an earlier version of the compiler. However, the JScript .NET compiler might encounter compile-time errors when referencing assemblies created with later versions of the compiler. For example, the JScript .NET 2003 compiler can reference any assembly created with the JScript .NET 2002 compiler, although the JScript .NET 2002 compiler may fail when referencing an assembly created with JScript .NET 2003.

Example

Compile source file `input.js` and import metadata from `metad1.dll` and `metad2.dll` to produce `out.exe`:

```
jsc /reference:metad1.dll;metad2.dll /out:out.exe input.js
```

See Also

JScript Compiler Options

/resource

Embeds a managed resource in an assembly.

```
/resource:filename[,name[,public|private]]
```

-or-

```
/res:filename[,name[,public|private]]
```

Arguments

filename
The resource file you want to embed in the output file.

name[,public|private] (optional)
The logical name for the resource; the name used to load the resource. The default is the name of the file. Optionally, you can specify whether the file is public or private in the assembly manifest. For example, /res:filename.res,myname.res,public. By default, *filename* is public in the assembly.

Remarks

Use the /resource option to link a resource to an assembly and not place the resource file in the output file.

If *filename* is a .NET Framework resource file created, for example, by the Resource File Generator (Resgen.exe) or in the development environment, it can be accessed with members in the System.Resources namespace (see System.Resources.ResourceManager for more information). For all other resources, use the GetManifestResource* methods in System.Reflection.Assembly class to access the resource at run time.

/res is the short form of /resource.

Example

Compile `in.js` and attach resource file `rf.resource`:

```
jsc /res:rf.resource in.js
```

See Also

JScript Compiler Options

/target

Specifies output file format.

The /target compiler option can be specified in one of three forms:

/target:exe
Creates a console .exe file.
/target:library
Creates a (.dll) code library.
/target:winexe
Creates a Windows program.

Remarks

/target causes a .NET Framework assembly manifest to be placed in an output file.

If you create an assembly, you can indicate that all or part of your code is CLS compliant with the CLSCompliantAttribute Class attribute.

```
import System;
[assembly:System.CLSCompliant(true)]   // specify assembly compliance

System.CLSCompliant(true) class TestClass   // specify compliance for element
{
   var i: int;
}
```

See Also

JScript Compiler Options

/target:exe

Creates a console application.

```
/target:exe
```

Remarks

The /target:exe option causes the compiler to create an executable (EXE), console application. The /target:exe option is in effect by default. The executable file will be created with the .exe extension.

Unless otherwise specified with the /out option, the output file name takes the name of the first source code file in the compilation for the each output file.

Use /target:winexe to create a Windows program executable.

When specified at the command line, all files up to the next /out or /target:library option are used to create the .exe. The /target:exe option is in effect for all files since the previous /out or /target:library option.

/t is the short form of /target.

Example

Each of the following command lines will compile `in.js`, creating `in.exe`:

```
jsc /target:exe in.js
jsc in.js
```

See Also

/target | JScript Compiler Options

/target:library

Creates a code library.

```
/target:library
```

Remarks

The /target:library option causes the compiler to create a DLL rather than an executable file (EXE). The DLL will be created with the .dll extension.

Unless otherwise specified with the /out option, the output file name takes the name of the first input file.

When specified at the command line, all source files up to the next /out or /target:exe option are used to create the .dll.

/t is the short form of /target.

> **Note** The JScript .NET compiler, jsc.exe, can reference assemblies created using the same version or an earlier version of the compiler. However, the JScript .NET compiler might encounter compile-time errors when referencing assemblies created with later versions of the compiler. For example, the JScript .NET 2003 compiler can reference any assembly created with the JScript .NET 2002 compiler, although the JScript .NET 2002 compiler may fail when referencing an assembly created with JScript .NET 2003.

Example

Compile `in.js`, creating `in.dll`:

```
jsc /target:library in.js
```

See Also

/target | JScript Compiler Options

/target:winexe

Creates a Windows program.

```
/target:winexe
```

Remarks

The /target:winexe option causes the compiler to create an executable (EXE), Windows program. The executable file will be created with the .exe extension. A Windows program is one that provides a user interface from the .NET Framework library.

Use /target:exe to create a console application.

Unless otherwise specified with the /out option, the output file name takes the name of the first source code file in the compilation of the output file.

When specified at the command line, all files until the next /out or /target option are used to create the Windows program.

/t is the short form of /target.

Example

Compile `in.cs` into a Windows program:

```
jsc /target:winexe in.js
```

See Also

/target | JScript Compiler Options

/utf8output

Displays compiler output using UTF-8 encoding.

```
/utf8output[+ | -]
```

Arguments

+ | -

By default /utf8output- displays output directly on the console. Specifying /utf8output or /utf8output+ redirects compiler output to a file.

Remarks

In some international configurations, compiler output cannot correctly be displayed in the console. In these configurations, use **/utf8output** and redirect compiler output to a file.

The default for this option is /utf8output-.

Specifying /utf8output is the same as specifying /utf8output+.

Example

Compile `in.js` and have the compiler display output using UTF-8 encoding:

```
jsc /utf8output in.js
```

See Also

JScript Compiler Options

/versionsafe

Flags implicit overrides.

```
/versionsafe[+ | -]
```

Arguments

+ | -
By default, /versionsafe- is in effect and the compiler will not generate an error if it finds an implicit method override. /versionsafe+, which is the same as /versionsafe, causes the compiler to generate errors for implicit method overrides.

Remarks

Use the hide or override keywords to explicitly indicate the override status of a method. For example, the following code will generate an error when compiled with /versionsafe:

```
class c
{
function f()
{
}
}
class d extends c
{
function f()
{
}
}
```

Example

Compile `in.js` and have the compiler generate errors if it finds an implicit method override.:

```
jsc /versionsafe in.js
```

See Also

JScript Compiler Options

/warn

Specifies warning level.

```
/warn:option
```

Arguments

option

The minimum warning level you want displayed for the build. Valid values are 0-4:

Warning level	**Meaning**
0	Turns off emission of all warning messages; display errors only.
1	Displays errors and severe warning messages.
2	Displays all errors and level 1 warnings plus certain, less-severe warnings, such as warnings about hiding class members.
3	Displays errors, level 1 and 2 warnings, plus certain, less-severe warnings, such as warnings about expressions that always evaluate to true or false.
4	Displays all errors, level 1-3 warnings, plus informational warnings. This is the default warning level at the command line.

Remarks

The /warn option specifies the warning level for the compiler to display.

Use /warnaserror to treat all warnings as errors up to the warning level specified. Higher-level warnings are ignored.

The compiler always displays errors.

/w is the short form of /warn.

Example

Compile `in.js` and have the compiler only display level 1 warnings:

```
jsc /warn:1 in.js
```

See Also

JScript Compiler Options

/warnaserror

Treats warnings as errors.

```
/warnaserror[+ | -]
```

Arguments

+ | -

The /warnaserror+ option treats all warnings as errors.

Remarks

Any messages that would ordinarily be reported as warnings are instead reported as errors. No output files are created. The build continues in order to identify as many errors/warnings as possible.

By default, /warnaserror- is in effect, which causes warnings to not prevent the generation of an output file. /warnaserror, which is the same as /warnaserror+, causes warnings to be treated as errors.

Use /warn to specify the level of warnings that you want the compiler to display.

Example

Compile `in.js` and have the compiler display no warnings:

```
jsc /warnaserror in.js
```

See Also

JScript Compiler Options

/win32res

Inserts a Win32 resource in the output file.

```
/win32res:filename
```

Arguments

filename
The resource file that you want to add to your output file.

Remarks

A Win32 resource file can be created with the Resource Compiler.

A Win32 resource can contain version or bitmap (icon) information that would help identify your application in the Windows Explorer. If you do not specify /win32res, the compiler will generate version information based on the assembly version.

See /linkresource (to reference) or /resource (to attach) a .NET Framework resource file.

Example

Compile `in.js` and attach a Win32 resource file `rf.res` to produce `in.exe`:

```
jsc /win32res:rf.res in.js
```

See Also

JScript Compiler Options