Implicit coercions in JScript 7

In this document I'll briefly describe the scenarios in which we perform implicit type coercions and the algorithms we use to perform the coercion.

Part 1: Under what circumstances are coercions performed?

The JS7 compiler can perform some type coercions at compile time (eg, compile-time constants). Generally the types are known at compile time but the values are not, so the compiler generates code that does run-time casts. In the worst possible late-bound case the types are not known until runtime and special late-bound helper functions must be invoked.

There are many situations in JS7 where a type coercion is generated:

- When an explicit cast is specified
- When passing arguments to a function with typed arguments
- When indexing an array
- When setting a typed global value, local value or member value using an assignment operator (=, +=, etc).
- When a function with a return type returns
- Some operators cause arguments to be coerced, e.g., (foo + "") coerces foo to string. Bitshift operators coerce arguments to 32 bit integers, etc.

A few more obscure scenarios are:

- When passing arguments to a custom attribute.
- If a JScript array is coerced to a .NET array then each member of the JScript array is coerced to the element type of the .NET array.
- If a .NET array is coerced to a JScript array then we wrap the .NET array with a JScript array wrapper. Setting values on the JScript array wrapper automatically coerces the value to the element type of the underlying .NET array.
- When subtracting a number from a char the result is coerced to char

Part 2: The coercion algorithms

How do we determine at compile time when an implicit coercion is legal?

Before I dig in we could use some definitions:

By "V **is coercible to** T" I mean that the specific instance V may be coerced to type T without data loss or error. For instance, the value Number(0.5) is coercible to the type "32 bit float" but not to "32 bit integer". (Note that some lossy coercions are allowed, such as Number to Boolean. See the coercibility algorithm for details.)

By "T1 **is promotable to** T2" I mean that any given instance of type T1 is coercible to type T2. For example: the type "32 bit signed integer" is promotable to "64 bit float" but not to "32 bit float". Any derived class is promotable to one of its base classes. Value types are promotable to their boxed types.

By "T1 is **assignable** to T2" I mean that there exists some instance of type T1 may be coerced to type T2. There may be a runtime error if the actual instance of T1 is not one assignable to T2. (The JS7 compiler will often issue a warning if the user makes an assignment entailing an implicit coercion between types known to be assignable but not promotable.) For example: all numeric types are assignable to each other. A base class is assignable to a derived class and vice-versa. Array is not assignable to Number and will produce a "type mismatch" error.

By "T1 **is element type compatible** to T2" I mean that types T1 and T2 are typed arrays with elements of type E1 and E2 respectively. If either E1 or E2 is a value type then T1 is ETC with T2 iff E1 = E2. If neither E1 nor E2 are value types then T1 is ETC with T2 iff E1 is promotable to E2.

When given an expression of the form

LHS = RHS;

JS7 determines the types of the LHS and RHS expressions at compile time as best it can and then checks to see if the RHS expression is **assignable** to the type of the LHS. If is it then it generates an implicit coercion.

The algorithms for determining promotability, assignability and coercibility follow:

2.1 The Promotability Algorithm

Consider an assignment of the form "LHS = RHS". JS7 infers the types of the expressions to be RHType and LHType. JS7 determines whether RHType is promotable to LHType with the following algorithm:

1) If RHType is equal to LHType then it is promotable.
2) If RHType is "undefined" or "null" then RHType is promotable to LHType (regardless of LHType.)
3) If RHType and LHType are both primitive numeric types then we determine if there is any possible data loss when casting any RHType value to LHType. If there is not then RHType is promotable to LHType.
4) If RHType is a subclass of LHType then it is promotable.
5) If RHType implements interface type LHType then it is promotable.
6) If RHType is a primitive Boolean, String or Number and LHType is the corresponding object type then it is promotable.
7) If either RHType or LHType is any array type then we use a special algorithm. This is rather complicated by the fact that JS7 supports both "JScript Classic" arrays and .NET arrays (which are multidimensional, hard typed, nonassociative arrays.)

7.1) If RHType is not an array type but LHType is then it is not promotable.
7.2) If RHType is an array type and LHType is not an array or "object" then it is not promotable.
7.3) If RHType is an array type and LHType is "object" then it is promotable.
7.4) If RHType is "JScript array" and LHType is not "JScript array" or "object" then it is not promotable.
7.5) If RHType is any .NET array type and LHType is the .NET "array of unknown type and rank" type then it is promotable.
7.6) If RHType is the .NET "unknown" array type then it is not promotable to LHType.
7.7) If RHType is "JScript array" or a .NET array known to be of rank 1 and LHType is "JScript array" then it is promotable.
7.8) If RHType and LHType are both .NET arrays of known type and rank then RHType is promotable to LHType iff the ranks are equal and RHType is element type compatible with LHType.

8) Boolean and String objects are promotable to their primitive value counterparts.
9) If RHType is an enumerated type then it is promotable to LHType iff LHType is not an enumerated type and RHType's implementation type is promotable to LHType.
10) If LHType is an enumerated type then RHType is promotable to LHType iff RHType is not an enumerated type and RHType is promotable to LHType's implementation type.
11) Otherwise, RHType is not promotable to LHType.

2.2: The Assignment Compatibility Algorithm

Now let me describe the algorithm we use to determine if an expression RHExp of compile-time type RHType is assignable to an object of compile-time type LHType. That is, if we can generate an implicit cast that always works OR might work.

We start with three broad categories:

A) RHExp is a compile-time constant.
B) RHExp is an array literal.
C) RHExp is something else.

Case A: RHExp is a compile-time constant:

This is perhaps the simplest case – JS7 checks to see if this compile-time constant is coercible to LHType. However there are some special cases.

A.1) **Class names**: If RHExp is the name of a class then RHExp is assignable to LHType iff LHType is "type" or "object"

A.2) **Enumerated types**: If LHType is an enumerated type and RHExp is a string then RHExp is assignable to LHType iff it names a member of that enumerated type. Otherwise, (i.e., if LHType is an enumerated type and RHExp is NOT a string) we fall through to the next case and treat LHType as though it were the "implementation type" of the enumeration.

A.3) **Other types**: If the RHExp is coercible to type LHType then it is assignable. If it is not then there are a few special cases. If RHExp is a numeric literal and RHType is "double" then it is assignable to LHType "single". If RHExp is a compile-time constant double but NOT a numeric literal and we are attempting to assign to LHType "single" then it is assignable iff the string representation of the double is identical to the string representation of the double downcast to a single.

Case B: RHExp is an array literal

This is rather complicated by the fact that JS7 supports both "JScript Classic" arrays and .NET arrays (which are multidimensional, hard typed, nonassociative arrays.)

B.1) If LHType is "object" or "JScript array" then it is assignment compatible.
B.2) If LHType is NOT ".NET array" then it is not assignment compatible.
B.3) If LHType is "a .NET array of unknown element type and unknown rank" then it is assignment compatible.
B.4) If LHType is "a .NET array of known rank" and that rank is not 1 then it is not assignment compatible.
B.5) If LHType is "a .NET array of unknown element type" then it is assignment compatible.
B.6) If LHType is "a .NET array of known element type E" then the RHExp is assignable to LHType iff every element of the array literal is assignable to E.

Case C: RHExp is something else.

C.1) If LHType is "object" then RHExp is assignable to LHType.
C.2) If RHType is "double" and LHType is any primitive numeric type then RHExp is assignable to LHType.
C.3) If RHType is promotable to LHType then it is assignable.
C.4) If LHType is "a .NET delegate type" and RHType is a compatible script function then it is assignable.
C.5) If RHType is "JScript array" and LHType is not any kind of array then it is not assignable.
C.6) If RHType is "JScript array" and LHType is "a .NET array of unknown rank" then it is assignable (with a warning).
C.7) If RHType is "JScript array" and LHType is "a .NET array of known rank" then it is assignable iff the rank is one.
C.8) If LHType is "string" then it is assignable.
C.9) If RHType is "string" and LHType is "Boolean" or any primitive numeric type then it is assignable (with a warning.)
C.10) If RHType is "string" and LHType is "char" then it is assignable (with a warning).
C.11) If LHType is promotable to RHType then RHType is assignable to LHType (with a warning). (For instance, base classes are assignable to derived classes even though they are not promotable.)
C.12) If RHType and LHType are both primitive numeric types then RHType is assignable to LHType (with a warning).

2.3: The Coercibility Algorithm for object types

Suppose we have a value V which we wish to coerce to type T without data loss or error. For clarity I'll split the algorithm up into two sections, one for "object" types and one for "primitive value" types.

We assume that T is not a "primitive value" type:

1) If T is "object" then V is coercible to T.
2) If V is "null" or "undefined" then V is coercible to T.
3) If T is a class and V is an instance of T (or an instance of a subclass of T) then V is coercible to T.

4) If T is an interface and V implements T then V is coercible to T.
5) If V is a JScript array and T is a .NET array of known element type E then V is coercible to T iff every element of V is coercible to E.
6) If V is a rank-one .NET array and T is "JScript array" then V is coercible to T.
7) If T and the type of V are both enumerated types then V is coercible to T iff the type of V is T.
8) If V is a string and T is an enumerated type then V is coercible to T iff V names a member of T.
9) If V is not from an enumerated type and T is an enumerated type then V is coercible to T iff V is coercible to the implementation type of T. (See the "primitive value" algorithm.)
10) If V is a class (i.e., not an instance of a class but actually names a class) and T is the "type" type then V is coercible to T.
11) If V is a closure or function object and T is a .NET delegate type then V is coercible to T iff it has a compatible signature with T.
12) If type T defines an appropriate type coercion function then V is coercible to T iff the function call succeeds.
13) Otherwise V is not coercible to T.

2.3: The Coercibility Algorithm for value types

Suppose we have a value V which we wish to coerce to type T without data loss or error. For clarity I'll split the algorithm up into two sections, one for "object" types and one for "primitive value" types.

We assume that T is a "primitive value" type, i.e. a Boolean, numeric type, or String, and not an enumerated type, array, class, null, interface, etc. Where convenient I'll indicate briefly what the conversion semantics are for Boolean, numeric types and String parenthetically.

JS7 supports several numeric types: 8, 16, 32 and 64 bit integers in both signed and unsigned flavours, 32 and 64 bit floats and a decimal type. JS7 also supports a "char" type which is an unsigned 16 bit integer that holds a single Unicode value. JS7 also supports a scalar Date/Time type which is treated the same as a signed 64 bit integer unless otherwise noted.

Note that data loss is allowed when coercing anything to Boolean and when coercing strings to numbers.

1) If the type of V is T then V is coercible to T.
2) If V is undefined or null then V is coercible to T. (Bool: false, Numeric: 0, String: "".)
3) If V is Boolean then V is coercible to T. (Numeric: 0/1, String: "true"/"false".)
4) If V is char then V is coercible to T. (Boolean: false if char is \u0000, true otherwise. Numeric: Unicode value. String: single-character string)
5) If V is of any numeric type and T is any numeric type then V is coercible to T provided that the specific value V may be cast to T without overflow or loss of precision. For instance, if V were of type UInt64 and equal to 300 then V could be coerced to an unsigned 16 bit int (but not an 8 bit int) regardless of the fact that the 64 bit integer type is not promotable to the 16 bit integer type.

   Note that if V is say the 64 bit float value 0.1 then this is not coercible to a 32 bit float as there will be a loss of precision. (0.5 would be coercible.) The assignability algorithm has a special case to ensure that compile-time constants inferred to be 64 bit floats are assignable to 32 bit floats even if they are not coercible.

6) If V is of any numeric type and T is "string" then V is coercible to T. (We use the standard number-to-string routine. If V is a Date/Time scalar then we output the appropriate date string.)
7) If V is of any numeric type and T is "Boolean" then V is coercible to T. (If V is zero or NaN then it goes to false, otherwise it goes to true.)
8) If V is a string and T is "Boolean" then V is coercible to T. (True if the string length is nonzero, false if it is zero.)
9) If V is a string and T is a Date/Time then V is coercible to T iff V is parsable as a date.
10) If V is a string and T is "char" then V is coercible to T iff V is a single-character string.
11) If V is a string and T is a numeric type then V is coercible to T if V may be parsed as type T. If it is not parsable as type T then V is coercible to T iff V is parsable as a 64 bit float and that value is coercible to T.