//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#pragma once

typedef int BOOL;

namespace Js {

class JavascriptConversion /\* All static \*/

{

public:

static Var OrdinaryToPrimitive(Var aValue, JavascriptHint hint, ScriptContext \* scriptContext);

static Var MethodCallToPrimitive(Var aValue, JavascriptHint hint, ScriptContext \* scriptContext);

static Var ToPrimitive(Var aValue, JavascriptHint hint, ScriptContext \* scriptContext);

static BOOL CanonicalNumericIndexString(Var aValue, double \*indexValue, ScriptContext \* scriptContext);

static BOOL ToPropertyKey(Var argument, ScriptContext\* scriptContext, const PropertyRecord\*\* propertyRecord);

static JavascriptString\* ToString(Var aValue, ScriptContext\* scriptContext);

static JavascriptString\* ToLocaleString(Var aValue, ScriptContext\* scriptContext);

static BOOL ToObject(Var aValue, ScriptContext\* scriptContext, RecyclableObject\*\* object);

static BOOL ToBoolean(Var aValue, ScriptContext\* scriptContext);

static BOOL ToBoolean\_Full(Var aValue, ScriptContext\* scriptContext);

static bool ToBool(Var aValue, ScriptContext\* scriptContext);

static double ToNumber(Var aValue, ScriptContext\* scriptContext);

static void ToFloat\_Helper(Var aValue, float \*pResult, ScriptContext\* scriptContext);

static void ToNumber\_Helper(Var aValue, double \*pResult, ScriptContext\* scriptContext);

static BOOL ToNumber\_FromPrimitive(Var aValue, double \*pResult, BOOL allowUndefined, ScriptContext\* scriptContext);

static double ToNumber\_Full(Var aValue, ScriptContext\* scriptContext);

static double ToInteger(Var aValue, ScriptContext\* scriptContext);

static double ToInteger(double value);

static double ToInteger\_Full(Var aValue, ScriptContext\* scriptContext);

static int32 ToInt32(Var aValue, ScriptContext\* scriptContext);

static \_\_int64 ToInt64(Var aValue, ScriptContext\* scriptContext);

static int32 ToInt32(double value);

static int32 ToInt32\_Full(Var aValue, ScriptContext\* scriptContext);

static int8 ToInt8(Var aValue, ScriptContext\* scriptContext);

static uint8 ToUInt8(Var aValue, ScriptContext\* scriptContext);

static uint8 ToUInt8Clamped(Var aValue, ScriptContext\* scriptContext);

static int16 ToInt16(Var aValue, ScriptContext\* scriptContext);

static float ToFloat(Var aValue, ScriptContext\* scriptContext);

static uint32 ToUInt32(Var aValue, ScriptContext\* scriptContext);

static unsigned \_\_int64 ToUInt64(Var aValue, ScriptContext\* scriptContext);

static uint32 ToUInt32(double value);

static uint32 ToUInt32\_Full(Var aValue, ScriptContext\* scriptContext);

static uint16 ToUInt16(Var aValue, ScriptContext\* scriptContext);

static uint16 ToUInt16(double value);

static uint16 ToUInt16\_Full(Var aValue, ScriptContext\* scriptContext);

static JavascriptString \*JavascriptConversion::CoerseString(Var aValue, ScriptContext\* scriptContext, const wchar\_t\* apiNameForErrorMsg);

static BOOL CheckObjectCoercible(Var aValue, ScriptContext\* scriptContext);

static bool SameValue(Var aValue, Var bValue);

static bool SameValueZero(Var aValue, Var bValue);

static bool IsCallable(Var aValue);

static BOOL ToInt32Finite(Var aValue, ScriptContext\* scriptContext, int32\* result);

// ToString(ToPrimitive(aValue), for convert to string on concat

static JavascriptString \* ToPrimitiveString(Var aValue, ScriptContext \* scriptContext);

static int64 ToLength(Var aValue, ScriptContext\* scriptContext);

private:

static BOOL ToInt32Finite(double value, int32\* result);

template<bool zero>

static bool SameValueCommon(Var aValue, Var bValue);

};

}

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#include "RuntimeLanguagePch.h"

#include "Library\JavascriptNumberObject.h"

#include "Library\JavascriptStringObject.h"

#include "Library\DateImplementation.h"

#include "Library\JavascriptDate.h"

extern "C" PVOID \_ReturnAddress(VOID);

#pragma intrinsic(\_ReturnAddress)

namespace Js

{

static const double k\_2to16 = 65536.0;

static const double k\_2to31 = 2147483648.0;

static const double k\_2to32 = 4294967296.0;

// ES5 9.10 indicates that this method should throw a TypeError if the supplied value is Undefined or Null.

// Our implementation returns FALSE in this scenario, expecting the caller to throw the TypeError.

// This allows the caller to provide more context in the error message without having to unnecessarily

// construct the message string before knowing whether or not the object is coercible.

BOOL JavascriptConversion::CheckObjectCoercible(Var aValue, ScriptContext\* scriptContext)

{

TypeId typeId = JavascriptOperators::GetTypeId(aValue);

if (typeId == TypeIds\_Null || typeId == TypeIds\_Undefined)

{

return FALSE;

}

return TRUE;

}

//ES5 9.11 Undefined, Null, Boolean, Number, String - return false

//If Object has an [[Call]] internal method, then return true, otherwise return false

bool JavascriptConversion::IsCallable(Var aValue)

{

if (!RecyclableObject::Is(aValue))

{

return false;

}

JavascriptMethod entryPoint = RecyclableObject::FromVar(aValue)->GetEntryPoint();

return RecyclableObject::DefaultEntryPoint != entryPoint;

}

//----------------------------------------------------------------------------

// ES5 9.12 SameValue algorithm implementation.

// 1. If Type(x) is different from Type(y), return false.

// 2. If Type(x) is Undefined, return true.

// 3. If Type(x) is Null, return true.

// 4. If Type(x) is Number, then.

// a. If x is NaN and y is NaN, return true.

// b. If x is +0 and y is -0, return false.

// c. If x is -0 and y is +0, return false.

// d. If x is the same number value as y, return true.

// e. Return false.

// 5. If Type(x) is String, then return true if x and y are exactly the same sequence of characters (same length and same characters in corresponding positions); otherwise, return false.

// 6. If Type(x) is Boolean, return true if x and y are both true or both false; otherwise, return false.

// 7. Return true if x and y refer to the same object. Otherwise, return false.

//----------------------------------------------------------------------------

template<bool zero>

bool JavascriptConversion::SameValueCommon(Var aLeft, Var aRight)

{

TypeId leftType = JavascriptOperators::GetTypeId(aLeft);

TypeId rightType = JavascriptOperators::GetTypeId(aRight);

//Check for undefined and null type;

if (leftType == TypeIds\_Undefined )

{

return rightType == TypeIds\_Undefined;

}

if (leftType == TypeIds\_Null)

{

return rightType == TypeIds\_Null;

}

double dblLeft, dblRight;

switch (leftType)

{

case TypeIds\_Integer:

switch (rightType)

{

case TypeIds\_Integer:

return aLeft == aRight;

case TypeIds\_Number:

dblLeft = TaggedInt::ToDouble(aLeft);

dblRight = JavascriptNumber::GetValue(aRight);

goto CommonNumber;

case TypeIds\_Int64Number:

{

int leftValue = TaggedInt::ToInt32(aLeft);

\_\_int64 rightValue = JavascriptInt64Number::FromVar(aRight)->GetValue();

return leftValue == rightValue;

}

case TypeIds\_UInt64Number:

{

int leftValue = TaggedInt::ToInt32(aLeft);

unsigned \_\_int64 rightValue = JavascriptInt64Number::FromVar(aRight)->GetValue();

return leftValue == rightValue;

}

}

break;

case TypeIds\_Int64Number:

switch (rightType)

{

case TypeIds\_Integer:

{

\_\_int64 leftValue = JavascriptInt64Number::FromVar(aLeft)->GetValue();

int rightValue = TaggedInt::ToInt32(aRight);

return leftValue == rightValue;

}

case TypeIds\_Number:

dblLeft = (double)JavascriptInt64Number::FromVar(aLeft)->GetValue();

dblRight = JavascriptNumber::GetValue(aRight);

goto CommonNumber;

case TypeIds\_Int64Number:

{

\_\_int64 leftValue = JavascriptInt64Number::FromVar(aLeft)->GetValue();

\_\_int64 rightValue = JavascriptInt64Number::FromVar(aRight)->GetValue();

return leftValue == rightValue;

}

case TypeIds\_UInt64Number:

{

\_\_int64 leftValue = JavascriptInt64Number::FromVar(aLeft)->GetValue();

unsigned \_\_int64 rightValue = JavascriptInt64Number::FromVar(aRight)->GetValue();

return ((unsigned \_\_int64)leftValue == rightValue);

}

}

break;

case TypeIds\_UInt64Number:

switch (rightType)

{

case TypeIds\_Integer:

{

unsigned \_\_int64 leftValue = JavascriptUInt64Number::FromVar(aLeft)->GetValue();

\_\_int64 rightValue = TaggedInt::ToInt32(aRight);

return (leftValue == (unsigned \_\_int64)rightValue);

}

case TypeIds\_Number:

dblLeft = (double)JavascriptUInt64Number::FromVar(aLeft)->GetValue();

dblRight = JavascriptNumber::GetValue(aRight);

goto CommonNumber;

case TypeIds\_Int64Number:

{

unsigned \_\_int64 leftValue = JavascriptUInt64Number::FromVar(aLeft)->GetValue();

\_\_int64 rightValue = JavascriptInt64Number::FromVar(aRight)->GetValue();

return (leftValue == (unsigned \_\_int64)rightValue);

}

case TypeIds\_UInt64Number:

{

unsigned \_\_int64 leftValue = JavascriptUInt64Number::FromVar(aLeft)->GetValue();

unsigned \_\_int64 rightValue = JavascriptInt64Number::FromVar(aRight)->GetValue();

return leftValue == rightValue;

}

}

break;

case TypeIds\_Number:

switch (rightType)

{

case TypeIds\_Integer:

dblLeft = JavascriptNumber::GetValue(aLeft);

dblRight = TaggedInt::ToDouble(aRight);

goto CommonNumber;

case TypeIds\_Int64Number:

dblLeft = JavascriptNumber::GetValue(aLeft);

dblRight = (double)JavascriptInt64Number::FromVar(aRight)->GetValue();

goto CommonNumber;

case TypeIds\_UInt64Number:

dblLeft = JavascriptNumber::GetValue(aLeft);

dblRight = (double)JavascriptUInt64Number::FromVar(aRight)->GetValue();

goto CommonNumber;

case TypeIds\_Number:

dblLeft = JavascriptNumber::GetValue(aLeft);

dblRight = JavascriptNumber::GetValue(aRight);

CommonNumber:

if (JavascriptNumber::IsNan(dblLeft) && JavascriptNumber::IsNan(dblRight))

{

return true;

}

if (zero)

{

// SameValueZero(+0,-0) returns true;

return dblLeft == dblRight;

}

else

{

// SameValue(+0,-0) returns false;

return (NumberUtilities::LuLoDbl(dblLeft) == NumberUtilities::LuLoDbl(dblRight) &&

NumberUtilities::LuHiDbl(dblLeft) == NumberUtilities::LuHiDbl(dblRight));

}

}

break;

case TypeIds\_Boolean:

switch (rightType)

{

case TypeIds\_Boolean:

return aLeft == aRight;

}

break;

case TypeIds\_String:

switch (rightType)

{

case TypeIds\_String:

return JavascriptString::Equals(aLeft, aRight);

}

break;

case TypeIds\_Symbol:

switch (rightType)

{

case TypeIds\_Symbol:

{

JavascriptSymbol\* leftSymbol = JavascriptSymbol::FromVar(aLeft);

JavascriptSymbol\* rightSymbol = JavascriptSymbol::FromVar(aRight);

return leftSymbol->GetValue() == rightSymbol->GetValue();

}

}

return false;

case TypeIds\_Function:

switch (rightType)

{

case TypeIds\_Function:

if (JavascriptFunction::FromVar(aLeft)->IsThrowTypeErrorFunction() &&

JavascriptFunction::FromVar(aRight)->IsThrowTypeErrorFunction())

{

return true;

}

}

break;

}

return aLeft == aRight;

}

template bool JavascriptConversion::SameValueCommon<false>(Var aLeft, Var aRight);

template bool JavascriptConversion::SameValueCommon<true>(Var aLeft, Var aRight);

//----------------------------------------------------------------------------

// ToObject() takes a value and converts it to a Object type

// Implementation of ES5 9.9

// The spec indicates that this method should throw a TypeError if the supplied value is Undefined or Null.

// Our implementation returns FALSE in this scenario, expecting the caller to throw the TypeError.

// This allows the caller to provide more context in the error message without having to unnecessarily

// construct the message string before knowing whether or not the value can be converted to an object.

//

// Undefined Return FALSE.

// Null Return FALSE.

// Boolean Create a new Boolean object whose [[PrimitiveValue]]

// internal property is set to the value of the boolean.

// See 15.6 for a description of Boolean objects.

// Return TRUE.

// Number Create a new Number object whose [[PrimitiveValue]]

// internal property is set to the value of the number.

// See 15.7 for a description of Number objects.

// Return TRUE.

// String Create a new String object whose [[PrimitiveValue]]

// internal property is set to the value of the string.

// See 15.5 for a description of String objects.

// Return TRUE.

// Object The result is the input argument (no conversion).

// Return TRUE.

//----------------------------------------------------------------------------

BOOL JavascriptConversion::ToObject(Var aValue, ScriptContext\* scriptContext, RecyclableObject\*\* object)

{

Assert(object);

switch (JavascriptOperators::GetTypeId(aValue))

{

case TypeIds\_Undefined:

case TypeIds\_Null:

return FALSE;

case TypeIds\_Number:

case TypeIds\_Integer:

\*object = scriptContext->GetLibrary()->CreateNumberObject(aValue);

return TRUE;

default:

{

\*object = RecyclableObject::FromVar(aValue)->ToObject(scriptContext);

return TRUE;

}

}

}

//----------------------------------------------------------------------------

// ToPropertyKey() takes a value and converts it to a property key

// Implementation of ES6 7.1.14

//----------------------------------------------------------------------------

BOOL JavascriptConversion::ToPropertyKey(Var argument, ScriptContext\* scriptContext, const PropertyRecord\*\* propertyRecord)

{

Var key = JavascriptConversion::ToPrimitive(argument, JavascriptHint::HintString, scriptContext);

if (JavascriptSymbol::Is(key))

{

// If we are looking up a property keyed by a symbol, we already have the PropertyId in the symbol

\*propertyRecord = JavascriptSymbol::FromVar(key)->GetValue();

}

else

{

// For all other types, convert the key into a string and use that as the property name

JavascriptString \* propName = JavascriptConversion::ToString(key, scriptContext);

if (VirtualTableInfo<Js::PropertyString>::HasVirtualTable(propName))

{

PropertyString \* propertyString = (PropertyString \*)propName;

\*propertyRecord = propertyString->GetPropertyRecord();

}

else

{

scriptContext->GetOrAddPropertyRecord(propName->GetString(), propName->GetLength(), propertyRecord);

}

}

return TRUE;

}

//----------------------------------------------------------------------------

// ToPrimitive() takes a value and an optional argument and converts it to a non Object type

// Implementation of ES5 9.1

//

// Undefined:The result equals the input argument (no conversion).

// Null: The result equals the input argument (no conversion).

// Boolean: The result equals the input argument (no conversion).

// Number: The result equals the input argument (no conversion).

// String: The result equals the input argument (no conversion).

// Object: Return a default value for the Object.

// The default value of an object is retrieved by calling the [[DefaultValue]]

// internal method of the object, passing the optional hint PreferredType.

// The behavior of the [[DefaultValue]] internal method is defined by this specification

// for all native ECMAScript objects (8.12.9).

//----------------------------------------------------------------------------

Var JavascriptConversion::ToPrimitive(Var aValue, JavascriptHint hint, ScriptContext \* requestContext)

{

switch (JavascriptOperators::GetTypeId(aValue))

{

case TypeIds\_Undefined:

case TypeIds\_Null:

case TypeIds\_Integer:

case TypeIds\_Boolean:

case TypeIds\_Number:

case TypeIds\_String:

case TypeIds\_Symbol:

return aValue;

case TypeIds\_StringObject:

{

JavascriptStringObject \* stringObject = JavascriptStringObject::FromVar(aValue);

if (stringObject->GetScriptContext()->optimizationOverrides.GetSideEffects() & (hint == JavascriptHint::HintString ? SideEffects\_ToString : SideEffects\_ValueOf))

{

return MethodCallToPrimitive(aValue, hint, requestContext);

}

return CrossSite::MarshalVar(requestContext, stringObject->Unwrap());

}

case TypeIds\_NumberObject:

{

JavascriptNumberObject \* numberObject = JavascriptNumberObject::FromVar(aValue);

if (hint == JavascriptHint::HintString)

{

if (numberObject->GetScriptContext()->optimizationOverrides.GetSideEffects() & SideEffects\_ToString)

{

return MethodCallToPrimitive(aValue, hint, requestContext);

}

return JavascriptNumber::ToStringRadix10(numberObject->GetValue(), requestContext);

}

else

{

if (numberObject->GetScriptContext()->optimizationOverrides.GetSideEffects() & SideEffects\_ValueOf)

{

return MethodCallToPrimitive(aValue, hint, requestContext);

}

return CrossSite::MarshalVar(requestContext, numberObject->Unwrap());

}

}

case TypeIds\_SymbolObject:

{

JavascriptSymbolObject\* symbolObject = JavascriptSymbolObject::FromVar(aValue);

return requestContext->GetLibrary()->CreateSymbol(symbolObject->GetValue());

}

case TypeIds\_Date:

case TypeIds\_WinRTDate:

{

JavascriptDate\* dateObject = JavascriptDate::FromVar(aValue);

if(hint == JavascriptHint::HintNumber)

{

if (dateObject->GetScriptContext()->optimizationOverrides.GetSideEffects() & SideEffects\_ValueOf)

{

// if no Method exists this function falls back to OrdinaryToPrimitive

// if IsES6ToPrimitiveEnabled flag is off we also fall back to OrdinaryToPrimitive

return MethodCallToPrimitive(aValue, hint, requestContext);

}

return JavascriptNumber::ToVarNoCheck(dateObject->GetTime(), requestContext);

}

else

{

if (dateObject->GetScriptContext()->optimizationOverrides.GetSideEffects() & SideEffects\_ToString)

{

// if no Method exists this function falls back to OrdinaryToPrimitive

// if IsES6ToPrimitiveEnabled flag is off we also fall back to OrdinaryToPrimitive

return MethodCallToPrimitive(aValue, hint, requestContext);

}

//NOTE: Consider passing requestContext to JavascriptDate::ToString

return CrossSite::MarshalVar(requestContext, JavascriptDate::ToString(dateObject));

}

}

// convert to JavascriptNumber

case TypeIds\_Int64Number:

return JavascriptInt64Number::FromVar(aValue)->ToJavascriptNumber();

case TypeIds\_UInt64Number:

return JavascriptUInt64Number::FromVar(aValue)->ToJavascriptNumber();

default:

// if no Method exists this function falls back to OrdinaryToPrimitive

// if IsES6ToPrimitiveEnabled flag is off we also fall back to OrdinaryToPrimitive

return MethodCallToPrimitive(aValue, hint, requestContext);

}

}

//----------------------------------------------------------------------------

//7.1.16 CanonicalNumericIndexString(argument)

//1. Assert : Type(argument) is String.

//2. If argument is "-0", then return -0.

//3. Let n be ToNumber(argument).

//4. If SameValue(ToString(n), argument) is false, then return undefined.

//5. Return n.

//----------------------------------------------------------------------------

BOOL JavascriptConversion::CanonicalNumericIndexString(Var aValue, double \*indexValue, ScriptContext \* scriptContext)

{

AssertMsg(JavascriptString::Is(aValue), "CanonicalNumericIndexString expects only string");

if (JavascriptString::IsNegZero(JavascriptString::FromVar(aValue)))

{

\*indexValue = -0;

return TRUE;

}

Var indexNumberValue = JavascriptOperators::ToNumber(aValue, scriptContext);

if (JavascriptString::Equals(JavascriptConversion::ToString(indexNumberValue, scriptContext), aValue))

{

\*indexValue = JavascriptNumber::GetValue(indexNumberValue);

return TRUE;

}

return FALSE;

}

Var JavascriptConversion::MethodCallToPrimitive(Var aValue, JavascriptHint hint, ScriptContext \* requestContext)

{

Var result = nullptr;

RecyclableObject \*const recyclableObject = RecyclableObject::FromVar(aValue);

ScriptContext \*const scriptContext = recyclableObject->GetScriptContext();

/\*7.3.7 GetMethod (O, P)

The abstract operation GetMethod is used to get the value of a specific property of an object when the value of the property is expected to be a function.

The operation is called with arguments O and P where O is the object, P is the property key. This abstract operation performs the following steps:

Assert: Type(O) is Object.

Assert: IsPropertyKey(P) is true.

Let func be the result of calling the [[Get]] internal method of O passing P and O as the arguments.

ReturnIfAbrupt(func).

If func is undefined, then return undefined.

If IsCallable(func) is false, then throw a TypeError exception.

Return func.\*/

Var varMethod;

if (!(requestContext->GetConfig()->IsES6ToPrimitiveEnabled()

&& JavascriptOperators::GetPropertyReference(recyclableObject, PropertyIds::\_symbolToPrimitive, &varMethod, requestContext)

&& !JavascriptOperators::IsUndefinedObject(varMethod)))

{

return OrdinaryToPrimitive(aValue, hint, requestContext);

}

if (!JavascriptFunction::Is(varMethod))

{

// Don't error if we disabled implicit calls

JavascriptError::TryThrowTypeError(scriptContext, requestContext, JSERR\_NeedFunction, requestContext->GetPropertyName(PropertyIds::\_symbolToPrimitive)->GetBuffer());

return requestContext->GetLibrary()->GetNull();

}

// Let exoticToPrim be GetMethod(input, @@toPrimitive).

JavascriptFunction\* exoticToPrim = JavascriptFunction::FromVar(varMethod);

JavascriptString\* hintString = nullptr;

if (hint == JavascriptHint::HintString)

{

hintString = requestContext->GetLibrary()->CreateStringFromCppLiteral(L"string");

}

else if (hint == JavascriptHint::HintNumber)

{

hintString = requestContext->GetLibrary()->CreateStringFromCppLiteral(L"number");

}

else

{

hintString = requestContext->GetLibrary()->CreateStringFromCppLiteral(L"default");

}

// If exoticToPrim is not undefined, then

if (nullptr != exoticToPrim)

{

ThreadContext \* threadContext = requestContext->GetThreadContext();

result = threadContext->ExecuteImplicitCall(exoticToPrim, ImplicitCall\_ToPrimitive, [=]()->Js::Var

{

// Stack object should have a pre-op bail on implicit call. We shouldn't see them here.

Assert(!ThreadContext::IsOnStack(recyclableObject));

// Let result be the result of calling the[[Call]] internal method of exoticToPrim, with input as thisArgument and(hint) as argumentsList.

return exoticToPrim->GetEntryPoint()(exoticToPrim, CallInfo(CallFlags\_Value, 2), recyclableObject, hintString);

});

Assert(!CrossSite::NeedMarshalVar(result, requestContext));

if (!result)

{

// There was an implicit call and implicit calls are disabled. This would typically cause a bailout.

Assert(threadContext->IsDisableImplicitCall());

return requestContext->GetLibrary()->GetNull();

}

}

// If result is an ECMAScript language value and Type(result) is not Object, then return result.

if (TaggedInt::Is(result) || JavascriptOperators::IsExposedType(JavascriptOperators::GetTypeId(result)))

{

return result;

}

// Else, throw a TypeError exception.

else

{

// Don't error if we disabled implicit calls

JavascriptError::TryThrowTypeError(scriptContext, requestContext, JSERR\_FunctionArgument\_Invalid, L"[Symbol.toPrimitive]");

return requestContext->GetLibrary()->GetNull();

}

}

Var JavascriptConversion::OrdinaryToPrimitive(Var aValue, JavascriptHint hint, ScriptContext \* requestContext)

{

Var result;

RecyclableObject \*const recyclableObject = RecyclableObject::FromVar(aValue);

if (!recyclableObject->ToPrimitive(hint, &result, requestContext))

{

ScriptContext \*const scriptContext = recyclableObject->GetScriptContext();

long hCode;

switch (hint)

{

case JavascriptHint::HintNumber:

hCode = JSERR\_NeedNumber;

break;

case JavascriptHint::HintString:

hCode = JSERR\_NeedString;

break;

default:

hCode = VBSERR\_OLENoPropOrMethod;

break;

}

JavascriptError::TryThrowTypeError(scriptContext, scriptContext, hCode);

return requestContext->GetLibrary()->GetNull();

}

return result;

}

JavascriptString \*JavascriptConversion::CoerseString(Var aValue, ScriptContext\* scriptContext, const wchar\_t\* apiNameForErrorMsg)

{

if (!JavascriptConversion::CheckObjectCoercible(aValue, scriptContext))

{

JavascriptError::ThrowTypeError(scriptContext, JSERR\_This\_NullOrUndefined, apiNameForErrorMsg);

}

return ToString(aValue, scriptContext);

}

//----------------------------------------------------------------------------

// ToString - abstract operation

// ES5 9.8

//Input Type Result

// Undefined

// "undefined"

// Null

// "null"

// Boolean

// If the argument is true, then the result is "true". If the argument is false, then the result is "false".

// Number

// See 9.8.1 below.

// String

// Return the input argument (no conversion)

// Object

// Apply the following steps:

// 1. Let primValue be ToPrimitive(input argument, hint String).

// 2. Return ToString(primValue).

//----------------------------------------------------------------------------

JavascriptString \*JavascriptConversion::ToString(Var aValue, ScriptContext\* scriptContext)

{

Assert(scriptContext->GetThreadContext()->IsScriptActive());

BOOL fPrimitiveOnly = false;

while(true)

{

switch (JavascriptOperators::GetTypeId(aValue))

{

case TypeIds\_Undefined:

return scriptContext->GetLibrary()->GetUndefinedDisplayString();

case TypeIds\_Null:

return scriptContext->GetLibrary()->GetNullDisplayString();

case TypeIds\_Integer:

return scriptContext->GetIntegerString(aValue);

case TypeIds\_Boolean:

return JavascriptBoolean::FromVar(aValue)->GetValue() ? scriptContext->GetLibrary()->GetTrueDisplayString() : scriptContext->GetLibrary()->GetFalseDisplayString();

case TypeIds\_Number:

return JavascriptNumber::ToStringRadix10(JavascriptNumber::GetValue(aValue), scriptContext);

case TypeIds\_Int64Number:

{

\_\_int64 value = JavascriptInt64Number::FromVar(aValue)->GetValue();

if (!TaggedInt::IsOverflow(value))

{

return scriptContext->GetIntegerString((int)value);

}

else

{

return JavascriptInt64Number::ToString(aValue, scriptContext);

}

}

case TypeIds\_UInt64Number:

{

unsigned \_\_int64 value = JavascriptUInt64Number::FromVar(aValue)->GetValue();

if (!TaggedInt::IsOverflow(value))

{

return scriptContext->GetIntegerString((uint)value);

}

else

{

return JavascriptUInt64Number::ToString(aValue, scriptContext);

}

}

case TypeIds\_String:

return JavascriptString::FromVar(CrossSite::MarshalVar(scriptContext, aValue));

case TypeIds\_VariantDate:

return JavascriptVariantDate::FromVar(aValue)->GetValueString(scriptContext);

case TypeIds\_Symbol:

return JavascriptSymbol::FromVar(aValue)->ToString(scriptContext);

case TypeIds\_GlobalObject:

aValue = static\_cast<Js::GlobalObject\*>(aValue)->ToThis();

// fall through

default:

{

AssertMsg(JavascriptOperators::IsObject(aValue), "bad type object in conversion ToString");

if(fPrimitiveOnly)

{

AssertMsg(FALSE, "wrong call in ToString, no dynamic objects should get here");

JavascriptError::ThrowError(scriptContext, VBSERR\_InternalError);

}

fPrimitiveOnly = true;

aValue = ToPrimitive(aValue, JavascriptHint::HintString, scriptContext);

}

}

}

}

JavascriptString \*JavascriptConversion::ToLocaleString(Var aValue, ScriptContext\* scriptContext)

{

switch (JavascriptOperators::GetTypeId(aValue))

{

case TypeIds\_Undefined:

return scriptContext->GetLibrary()->GetUndefinedDisplayString();

case TypeIds\_Null:

return scriptContext->GetLibrary()->GetNullDisplayString();

case TypeIds\_Integer:

return JavascriptNumber::ToLocaleString(TaggedInt::ToInt32(aValue), scriptContext);

case TypeIds\_Boolean:

return JavascriptBoolean::FromVar(aValue)->GetValue() ? scriptContext->GetLibrary()->GetTrueDisplayString() : scriptContext->GetLibrary()->GetFalseDisplayString();

case TypeIds\_Int64Number:

return JavascriptNumber::ToLocaleString((double)JavascriptInt64Number::FromVar(aValue)->GetValue(), scriptContext);

case TypeIds\_UInt64Number:

return JavascriptNumber::ToLocaleString((double)JavascriptUInt64Number::FromVar(aValue)->GetValue(), scriptContext);

case TypeIds\_Number:

return JavascriptNumber::ToLocaleString(JavascriptNumber::GetValue(aValue), scriptContext);

case TypeIds\_String:

return JavascriptString::FromVar(aValue);

case TypeIds\_VariantDate:

// Legacy behavior was to create an empty object and call toLocaleString on it, which would result in this value

return scriptContext->GetLibrary()->GetObjectDisplayString();

case TypeIds\_Symbol:

return JavascriptSymbol::FromVar(aValue)->ToString(scriptContext);

default:

{

RecyclableObject\* object = RecyclableObject::FromVar(aValue);

Var value = JavascriptOperators::GetProperty(object, PropertyIds::toLocaleString, scriptContext, NULL);

if (JavascriptConversion::IsCallable(value))

{

RecyclableObject\* toLocaleStringFunction = RecyclableObject::FromVar(value);

Var aResult = toLocaleStringFunction->GetEntryPoint()(toLocaleStringFunction, 1, aValue);

if (JavascriptString::Is(aResult))

{

return JavascriptString::FromVar(aResult);

}

else

{

return JavascriptConversion::ToString(aResult, scriptContext);

}

}

JavascriptError::ThrowTypeError(scriptContext, JSERR\_Property\_NeedFunction, scriptContext->GetPropertyName(PropertyIds::toLocaleString)->GetBuffer());

}

}

}

//----------------------------------------------------------------------------

// ToBoolean\_Full:

// (ES3.0: S9.2):

//

// Input Output

// ----- ------

// 'undefined' 'false'

// 'null' 'false'

// Boolean Value

// Number 'false' if +0, -0, or Nan

// 'true' otherwise

// String 'false' if argument is ""

// 'true' otherwise

// Object 'true'

// Falsy Object 'false'

//----------------------------------------------------------------------------

BOOL JavascriptConversion::ToBoolean\_Full(Var aValue, ScriptContext\* scriptContext)

{

AssertMsg(!TaggedInt::Is(aValue), "Should be detected");

AssertMsg(RecyclableObject::Is(aValue), "Should be handled already");

auto type = RecyclableObject::FromVar(aValue)->GetType();

switch (type->GetTypeId())

{

case TypeIds\_Undefined:

case TypeIds\_Null:

case TypeIds\_VariantDate:

return false;

case TypeIds\_Symbol:

return true;

case TypeIds\_Boolean:

return JavascriptBoolean::FromVar(aValue)->GetValue();

#if !FLOATVAR

case TypeIds\_Number:

{

double value = JavascriptNumber::GetValue(aValue);

return (!JavascriptNumber::IsNan(value)) && (!JavascriptNumber::IsZero(value));

}

#endif

case TypeIds\_Int64Number:

{

\_\_int64 value = JavascriptInt64Number::FromVar(aValue)->GetValue();

return value != 0;

}

case TypeIds\_UInt64Number:

{

unsigned \_\_int64 value = JavascriptUInt64Number::FromVar(aValue)->GetValue();

return value != 0;

}

case TypeIds\_String:

{

JavascriptString \* pstValue = JavascriptString::FromVar(aValue);

return pstValue->GetLength() > 0;

}

case TypeIds\_SIMDFloat32x4:

case TypeIds\_SIMDFloat64x2:

case TypeIds\_SIMDInt32x4:

{ // SIMD review: may need update once spec is finalized

if (scriptContext->GetConfig()->IsSimdjsEnabled())

{

return true;

}

}

default:

{

AssertMsg(JavascriptOperators::IsObject(aValue), "bad type object in conversion ToBoolean");

// Falsy objects evaluate to false when converted to Boolean.

return !type->IsFalsy();

}

}

}

void JavascriptConversion::ToFloat\_Helper(Var aValue, float \*pResult, ScriptContext\* scriptContext)

{

\*pResult = (float)ToNumber\_Full(aValue, scriptContext);

}

void JavascriptConversion::ToNumber\_Helper(Var aValue, double \*pResult, ScriptContext\* scriptContext)

{

Assert(Js::JavascriptStackWalker::ValidateTopJitFrame(scriptContext));

\*pResult = ToNumber\_Full(aValue, scriptContext);

}

// Used for the JIT's float type specialization

// Convert aValue to double, but only allow primitives. Return false otherwise.

BOOL JavascriptConversion::ToNumber\_FromPrimitive(Var aValue, double \*pResult, BOOL allowUndefined, ScriptContext\* scriptContext)

{

Assert(Js::JavascriptStackWalker::ValidateTopJitFrame(scriptContext));

Assert(!TaggedNumber::Is(aValue));

RecyclableObject \*obj = RecyclableObject::FromVar(aValue);

// NOTE: Don't allow strings, otherwise JIT's float type specialization has to worry about concats

if (obj->GetTypeId() >= TypeIds\_String)

{

return false;

}

if (!allowUndefined && obj->GetTypeId() == TypeIds\_Undefined)

{

return false;

}

\*pResult = ToNumber\_Full(aValue, scriptContext);

return true;

}

//----------------------------------------------------------------------------

// ToNumber\_Full:

// Implements ES6 Draft Rev 26 July 18, 2014

//

// Undefined: NaN

// Null: 0

// boolean: v==true ? 1 : 0 ;

// number: v (original number)

// String: conversion by spec algorithm

// object: ToNumber(PrimitiveValue(v, hint\_number))

// Symbol: TypeError

//----------------------------------------------------------------------------

double JavascriptConversion::ToNumber\_Full(Var aValue,ScriptContext\* scriptContext)

{

AssertMsg(!TaggedInt::Is(aValue), "Should be detected");

ScriptContext \* objectScriptContext = RecyclableObject::Is(aValue) ? RecyclableObject::FromVar(aValue)->GetScriptContext() : nullptr;

BOOL fPrimitiveOnly = false;

while(true)

{

switch (JavascriptOperators::GetTypeId(aValue))

{

case TypeIds\_Symbol:

JavascriptError::TryThrowTypeError(objectScriptContext, scriptContext, JSERR\_NeedNumber);

// Fallthrough to return NaN if exceptions are disabled

case TypeIds\_Undefined:

return JavascriptNumber::GetValue(scriptContext->GetLibrary()->GetNaN());

case TypeIds\_Null:

return 0;

case TypeIds\_Integer:

return TaggedInt::ToDouble(aValue);

case TypeIds\_Boolean:

return JavascriptBoolean::FromVar(aValue)->GetValue() ? 1 : +0;

case TypeIds\_Number:

return JavascriptNumber::GetValue(aValue);

case TypeIds\_Int64Number:

return (double)JavascriptInt64Number::FromVar(aValue)->GetValue();

case TypeIds\_UInt64Number:

return (double)JavascriptUInt64Number::FromVar(aValue)->GetValue();

case TypeIds\_String:

return JavascriptString::FromVar(aValue)->ToDouble();

case TypeIds\_VariantDate:

return Js::DateImplementation::GetTvUtc(Js::DateImplementation::JsLocalTimeFromVarDate(JavascriptVariantDate::FromVar(aValue)->GetValue()), scriptContext);

case TypeIds\_SIMDFloat32x4:

case TypeIds\_SIMDInt32x4:

case TypeIds\_SIMDFloat64x2:

JavascriptError::ThrowError(scriptContext, JSERR\_NeedNumber);

default:

{

AssertMsg(JavascriptOperators::IsObject(aValue), "bad type object in conversion ToInteger");

if(fPrimitiveOnly)

{

JavascriptError::ThrowError(scriptContext, VBSERR\_OLENoPropOrMethod);

}

fPrimitiveOnly = true;

aValue = ToPrimitive(aValue, JavascriptHint::HintNumber, scriptContext);

}

}

}

}

//----------------------------------------------------------------------------

// second part of the ToInteger() implementation.(ES5.0: S9.4).

//----------------------------------------------------------------------------

double JavascriptConversion::ToInteger\_Full(Var aValue,ScriptContext\* scriptContext)

{

AssertMsg(!TaggedInt::Is(aValue), "Should be detected");

ScriptContext \* objectScriptContext = RecyclableObject::Is(aValue) ? RecyclableObject::FromVar(aValue)->GetScriptContext() : nullptr;

BOOL fPrimitiveOnly = false;

while(true)

{

switch (JavascriptOperators::GetTypeId(aValue))

{

case TypeIds\_Symbol:

JavascriptError::TryThrowTypeError(objectScriptContext, scriptContext, JSERR\_NeedNumber);

// Fallthrough to return 0 if exceptions are disabled

case TypeIds\_Undefined:

case TypeIds\_Null:

return 0;

case TypeIds\_Integer:

return TaggedInt::ToInt32(aValue);

case TypeIds\_Boolean:

return JavascriptBoolean::FromVar(aValue)->GetValue() ? 1 : +0;

case TypeIds\_Number:

return ToInteger(JavascriptNumber::GetValue(aValue));

case TypeIds\_Int64Number:

return ToInteger((double)JavascriptInt64Number::FromVar(aValue)->GetValue());

case TypeIds\_UInt64Number:

return ToInteger((double)JavascriptUInt64Number::FromVar(aValue)->GetValue());

case TypeIds\_String:

return ToInteger(JavascriptString::FromVar(aValue)->ToDouble());

case TypeIds\_VariantDate:

return ToInteger(ToNumber\_Full(aValue, scriptContext));

default:

{

AssertMsg(JavascriptOperators::IsObject(aValue), "bad type object in conversion ToInteger");

if(fPrimitiveOnly)

{

AssertMsg(FALSE, "wrong call in ToInteger\_Full, no dynamic objects should get here");

JavascriptError::ThrowError(scriptContext, VBSERR\_OLENoPropOrMethod);

}

fPrimitiveOnly = true;

aValue = ToPrimitive(aValue, JavascriptHint::HintNumber, scriptContext);

}

}

}

}

double JavascriptConversion::ToInteger(double val)

{

if(JavascriptNumber::IsNan(val))

return 0;

if(JavascriptNumber::IsPosInf(val) || JavascriptNumber::IsNegInf(val) ||

JavascriptNumber::IsZero(val))

{

return val;

}

return ( ((val < 0) ? -1 : 1 ) \* floor(fabs(val)));

}

//----------------------------------------------------------------------------

// ToInt32() converts the given Var to an Int32 value, as described in

// (ES3.0: S9.5).

//----------------------------------------------------------------------------

int32 JavascriptConversion::ToInt32\_Full(Var aValue, ScriptContext\* scriptContext)

{

Assert(Js::JavascriptStackWalker::ValidateTopJitFrame(scriptContext));

AssertMsg(!TaggedInt::Is(aValue), "Should be detected");

ScriptContext \* objectScriptContext = RecyclableObject::Is(aValue) ? RecyclableObject::FromVar(aValue)->GetScriptContext() : nullptr;

// This is used when TaggedInt's overflow but remain under int32

// so Number is our most critical case:

TypeId typeId = JavascriptOperators::GetTypeId(aValue);

if (typeId == TypeIds\_Number)

{

return JavascriptMath::ToInt32Core(JavascriptNumber::GetValue(aValue));

}

switch (typeId)

{

case TypeIds\_Symbol:

JavascriptError::TryThrowTypeError(objectScriptContext, scriptContext, JSERR\_NeedNumber);

// Fallthrough to return 0 if exceptions are disabled

case TypeIds\_Undefined:

case TypeIds\_Null:

return 0;

case TypeIds\_Integer:

return TaggedInt::ToInt32(aValue);

case TypeIds\_Boolean:

return JavascriptBoolean::FromVar(aValue)->GetValue() ? 1 : +0;

case TypeIds\_Int64Number:

// we won't lose precision if the int64 is within 32bit boundary; otherwise we need to

// treat it as double anyhow.

return JavascriptMath::ToInt32Core((double)JavascriptInt64Number::FromVar(aValue)->GetValue());

case TypeIds\_UInt64Number:

// we won't lose precision if the int64 is within 32bit boundary; otherwise we need to

// treat it as double anyhow.

return JavascriptMath::ToInt32Core((double)JavascriptUInt64Number::FromVar(aValue)->GetValue());

case TypeIds\_String:

{

double result;

if (JavascriptString::FromVar(aValue)->ToDouble(&result))

{

return JavascriptMath::ToInt32Core(result);

}

// If the string isn't a valid number, ToDouble returns NaN, and ToInt32 of that is 0

return 0;

}

case TypeIds\_VariantDate:

return ToInt32(ToNumber\_Full(aValue, scriptContext));

default:

AssertMsg(JavascriptOperators::IsObject(aValue), "bad type object in conversion ToInteger32");

aValue = ToPrimitive(aValue, JavascriptHint::HintNumber, scriptContext);

}

switch (JavascriptOperators::GetTypeId(aValue))

{

case TypeIds\_Symbol:

JavascriptError::TryThrowTypeError(objectScriptContext, scriptContext, JSERR\_NeedNumber);

// Fallthrough to return 0 if exceptions are disabled

case TypeIds\_Undefined:

case TypeIds\_Null:

return 0;

case TypeIds\_Integer:

return TaggedInt::ToInt32(aValue);

case TypeIds\_Boolean:

return JavascriptBoolean::FromVar(aValue)->GetValue() ? 1 : +0;

case TypeIds\_Number:

return ToInt32(JavascriptNumber::GetValue(aValue));

case TypeIds\_Int64Number:

// we won't lose precision if the int64 is within 32bit boundary; otherwise we need to

// treat it as double anyhow.

return JavascriptMath::ToInt32Core((double)JavascriptInt64Number::FromVar(aValue)->GetValue());

case TypeIds\_UInt64Number:

// we won't lose precision if the int64 is within 32bit boundary; otherwise we need to

// treat it as double anyhow.

return JavascriptMath::ToInt32Core((double)JavascriptUInt64Number::FromVar(aValue)->GetValue());

case TypeIds\_String:

{

double result;

if (JavascriptString::FromVar(aValue)->ToDouble(&result))

{

return ToInt32(result);

}

// If the string isn't a valid number, ToDouble returns NaN, and ToInt32 of that is 0

return 0;

}

case TypeIds\_VariantDate:

return ToInt32(ToNumber\_Full(aValue, scriptContext));

default:

AssertMsg(FALSE, "wrong call in ToInteger32\_Full, no dynamic objects should get here.");

JavascriptError::ThrowError(scriptContext, VBSERR\_OLENoPropOrMethod);

}

}

// a strict version of ToInt32 conversion that returns false for non int32 values like, inf, NaN, undef

BOOL JavascriptConversion::ToInt32Finite(Var aValue, ScriptContext\* scriptContext, int32\* result)

{

ScriptContext \* objectScriptContext = RecyclableObject::Is(aValue) ? RecyclableObject::FromVar(aValue)->GetScriptContext() : nullptr;

BOOL fPrimitiveOnly = false;

while(true)

{

switch (JavascriptOperators::GetTypeId(aValue))

{

case TypeIds\_Symbol:

JavascriptError::TryThrowTypeError(objectScriptContext, scriptContext, JSERR\_NeedNumber);

// Fallthrough to return false and set result to 0 if exceptions are disabled

case TypeIds\_Undefined:

\*result = 0;

return false;

case TypeIds\_Null:

\*result = 0;

return true;

case TypeIds\_Integer:

\*result = TaggedInt::ToInt32(aValue);

return true;

case TypeIds\_Boolean:

\*result = JavascriptBoolean::FromVar(aValue)->GetValue() ? 1 : +0;

return true;

case TypeIds\_Number:

return ToInt32Finite(JavascriptNumber::GetValue(aValue), result);

case TypeIds\_Int64Number:

// we won't lose precision if the int64 is within 32bit boundary; otherwise we need to

// treat it as double anyhow.

return ToInt32Finite((double)JavascriptInt64Number::FromVar(aValue)->GetValue(), result);

case TypeIds\_UInt64Number:

// we won't lose precision if the int64 is within 32bit boundary; otherwise we need to

// treat it as double anyhow.

return ToInt32Finite((double)JavascriptUInt64Number::FromVar(aValue)->GetValue(), result);

case TypeIds\_String:

return ToInt32Finite(JavascriptString::FromVar(aValue)->ToDouble(), result);

case TypeIds\_VariantDate:

return ToInt32Finite(ToNumber\_Full(aValue, scriptContext), result);

default:

{

AssertMsg(JavascriptOperators::IsObject(aValue), "bad type object in conversion ToInteger32");

if(fPrimitiveOnly)

{

AssertMsg(FALSE, "wrong call in ToInteger32\_Full, no dynamic objects should get here");

JavascriptError::ThrowError(scriptContext, VBSERR\_OLENoPropOrMethod);

}

fPrimitiveOnly = true;

aValue = ToPrimitive(aValue, JavascriptHint::HintNumber, scriptContext);

}

}

}

}

int32 JavascriptConversion::ToInt32(double T1)

{

return JavascriptMath::ToInt32Core(T1);

}

\_\_int64 JavascriptConversion::ToInt64(Var aValue, ScriptContext\* scriptContext)

{

switch (JavascriptOperators::GetTypeId(aValue))

{

case TypeIds\_Integer:

{

return TaggedInt::ToInt32(aValue);

}

case TypeIds\_Int64Number:

{

JavascriptInt64Number\* int64Number = JavascriptInt64Number::FromVar(aValue);

return int64Number->GetValue();

}

case TypeIds\_UInt64Number:

{

JavascriptUInt64Number\* uint64Number = JavascriptUInt64Number::FromVar(aValue);

return (\_\_int64)uint64Number->GetValue();

}

case TypeIds\_Number:

return JavascriptMath::TryToInt64(JavascriptNumber::GetValue(aValue));

default:

return (unsigned \_\_int64)JavascriptConversion::ToInt32\_Full(aValue, scriptContext);

}

}

unsigned \_\_int64 JavascriptConversion::ToUInt64(Var aValue, ScriptContext\* scriptContext)

{

switch (JavascriptOperators::GetTypeId(aValue))

{

case TypeIds\_Integer:

{

return (unsigned \_\_int64)TaggedInt::ToInt32(aValue);

}

case TypeIds\_Int64Number:

{

JavascriptInt64Number\* int64Number = JavascriptInt64Number::FromVar(aValue);

return (unsigned \_\_int64)int64Number->GetValue();

}

case TypeIds\_UInt64Number:

{

JavascriptUInt64Number\* uint64Number = JavascriptUInt64Number::FromVar(aValue);

return uint64Number->GetValue();

}

case TypeIds\_Number:

return static\_cast<unsigned \_\_int64>(JavascriptMath::TryToInt64(JavascriptNumber::GetValue(aValue)));

default:

return (unsigned \_\_int64)JavascriptConversion::ToInt32\_Full(aValue, scriptContext);

}

}

BOOL JavascriptConversion::ToInt32Finite(double value, int32\* result)

{

if((!NumberUtilities::IsFinite(value)) || JavascriptNumber::IsNan(value))

{

\*result = 0;

return false;

}

else

{

\*result = JavascriptMath::ToInt32Core(value);

return true;

}

}

//----------------------------------------------------------------------------

// (ES3.0: S9.6).

//----------------------------------------------------------------------------

uint32 JavascriptConversion::ToUInt32\_Full(Var aValue, ScriptContext\* scriptContext)

{

AssertMsg(!TaggedInt::Is(aValue), "Should be detected");

ScriptContext \* objectScriptContext = RecyclableObject::Is(aValue) ? RecyclableObject::FromVar(aValue)->GetScriptContext() : nullptr;

BOOL fPrimitiveOnly = false;

while(true)

{

switch (JavascriptOperators::GetTypeId(aValue))

{

case TypeIds\_Symbol:

JavascriptError::TryThrowTypeError(objectScriptContext, scriptContext, JSERR\_NeedNumber);

// Fallthrough to return 0 if exceptions are disabled

case TypeIds\_Undefined:

case TypeIds\_Null:

return 0;

case TypeIds\_Integer:

return TaggedInt::ToUInt32(aValue);

case TypeIds\_Boolean:

return JavascriptBoolean::FromVar(aValue)->GetValue() ? 1 : +0;

case TypeIds\_Number:

return JavascriptMath::ToUInt32(JavascriptNumber::GetValue(aValue));

case TypeIds\_Int64Number:

// we won't lose precision if the int64 is within 32bit boundary; otherwise we need to

// treat it as double anyhow.

return JavascriptMath::ToUInt32((double)JavascriptInt64Number::FromVar(aValue)->GetValue());

case TypeIds\_UInt64Number:

// we won't lose precision if the int64 is within 32bit boundary; otherwise we need to

// treat it as double anyhow.

return JavascriptMath::ToUInt32((double)JavascriptUInt64Number::FromVar(aValue)->GetValue());

case TypeIds\_String:

{

double result;

if (JavascriptString::FromVar(aValue)->ToDouble(&result))

{

return JavascriptMath::ToUInt32(result);

}

// If the string isn't a valid number, ToDouble returns NaN, and ToUInt32 of that is 0

return 0;

}

case TypeIds\_VariantDate:

return JavascriptMath::ToUInt32(ToNumber\_Full(aValue, scriptContext));

default:

{

AssertMsg(JavascriptOperators::IsObject(aValue), "bad type object in conversion ToUInt32");

if(fPrimitiveOnly)

{

AssertMsg(FALSE, "wrong call in ToUInt32\_Full, no dynamic objects should get here");

JavascriptError::ThrowError(scriptContext, VBSERR\_OLENoPropOrMethod);

}

fPrimitiveOnly = true;

aValue = ToPrimitive(aValue, JavascriptHint::HintNumber, scriptContext);

}

}

}

}

uint32 JavascriptConversion::ToUInt32(double T1)

{

// Same as doing ToInt32 and reinterpret the bits as uint32

return (uint32)JavascriptMath::ToInt32Core(T1);

}

//----------------------------------------------------------------------------

// ToUInt16() converts the given Var to a UInt16 value, as described in

// (ES3.0: S9.6).

//----------------------------------------------------------------------------

uint16 JavascriptConversion::ToUInt16\_Full(IN Var aValue, ScriptContext\* scriptContext)

{

AssertMsg(!TaggedInt::Is(aValue), "Should be detected");

ScriptContext \* objectScriptContext = RecyclableObject::Is(aValue) ? RecyclableObject::FromVar(aValue)->GetScriptContext() : nullptr;

BOOL fPrimitiveOnly = false;

while(true)

{

switch (JavascriptOperators::GetTypeId(aValue))

{

case TypeIds\_Symbol:

JavascriptError::TryThrowTypeError(objectScriptContext, scriptContext, JSERR\_NeedNumber);

// Fallthrough to return 0 if exceptions are disabled

case TypeIds\_Undefined:

case TypeIds\_Null:

return 0;

case TypeIds\_Integer:

return TaggedInt::ToUInt16(aValue);

case TypeIds\_Boolean:

return JavascriptBoolean::FromVar(aValue)->GetValue() ? 1 : +0;

case TypeIds\_Number:

return ToUInt16(JavascriptNumber::GetValue(aValue));

case TypeIds\_Int64Number:

// we won't lose precision if the int64 is within 16bit boundary; otherwise we need to

// treat it as double anyhow.

return ToUInt16((double)JavascriptInt64Number::FromVar(aValue)->GetValue());

case TypeIds\_UInt64Number:

// we won't lose precision if the int64 is within 16bit boundary; otherwise we need to

// treat it as double anyhow.

return ToUInt16((double)JavascriptUInt64Number::FromVar(aValue)->GetValue());

case TypeIds\_String:

{

double result;

if (JavascriptString::FromVar(aValue)->ToDouble(&result))

{

return ToUInt16(result);

}

// If the string isn't a valid number, ToDouble is NaN, and ToUInt16 of that is 0

return 0;

}

case TypeIds\_VariantDate:

return ToUInt16(ToNumber\_Full(aValue, scriptContext));

default:

{

AssertMsg(JavascriptOperators::IsObject(aValue), "bad type object in conversion ToUIn16");

if(fPrimitiveOnly)

{

AssertMsg(FALSE, "wrong call in ToUInt16, no dynamic objects should get here");

JavascriptError::ThrowError(scriptContext, VBSERR\_OLENoPropOrMethod);

}

fPrimitiveOnly = true;

aValue = ToPrimitive(aValue, JavascriptHint::HintNumber, scriptContext);

}

}

}

}

\_\_inline uint16 JavascriptConversion::ToUInt16(double T1)

{

//

// VC does the right thing here, if we first convert to uint32 and then to uint16

// Spec says mod should be done.

//

uint32 result = JavascriptConversion::ToUInt32(T1);

#if defined(\_M\_IX86) && \_MSC\_FULL\_VER < 160030329

// Well VC doesn't actually do the right thing... It takes (uint16)(uint32)double and removes the

// middle uint32 cast to (uint16)double, which isn't the same thing. Somehow, it only seems to be a

// problem for x86. Forcing a store to uint32 prevents the incorrect optimization.

//

// A bug has been filled in the Dev11 database: TF bug id #901495

// Fixed in compiler 16.00.30329.00

volatile uint32 volResult = result;

#endif

return (uint16) result;

}

JavascriptString \* JavascriptConversion::ToPrimitiveString(Var aValue, ScriptContext \* scriptContext)

{

return ToString(ToPrimitive(aValue, JavascriptHint::None, scriptContext), scriptContext);

}

int64 JavascriptConversion::ToLength(Var aValue, ScriptContext\* scriptContext)

{

if (TaggedInt::Is(aValue))

{

int64 length = TaggedInt::ToInt64(aValue);

return (length < 0) ? 0 : length;

}

double length = JavascriptConversion::ToInteger(aValue, scriptContext);

if (length < 0.0 || JavascriptNumber::IsNegZero(length))

{

length = 0.0;

}

else if (length > Math::MAX\_SAFE\_INTEGER)

{

length = Math::MAX\_SAFE\_INTEGER;

}

return NumberUtilities::TryToInt64(length);

}

} // namespace Js