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

// 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

namespace Js

{

class IDiagObjectAddress;

class IDiagObjectModelDisplay;

class RecyclableMethodsGroupWalker;

class RecyclableObjectWalker;

class RecyclableArrayWalker;

// Concrete type for manipulating JS Vars

struct ResolvedObject

{

ResolvedObject() : propId(Js::Constants::NoProperty), scriptContext(nullptr), address(nullptr),

objectDisplay(nullptr), obj(nullptr), originalObj(nullptr), isConst(false), name(nullptr)

{}

PropertyId propId;

ScriptContext \*scriptContext;

IDiagObjectAddress \*address;

IDiagObjectModelDisplay \*objectDisplay;

Var obj;

Var originalObj;

LPCWSTR name;

TypeId typeId;

bool isConst;

WeakArenaReference<IDiagObjectModelDisplay>\* GetObjectDisplay();

IDiagObjectModelDisplay \* CreateDisplay();

bool IsInDeadZone() const;

};

// interfaces for manipulating DataTypes

// Allow setting the value across different parent data types

class IDiagObjectAddress

{

public:

virtual BOOL Set(Var updateObject) = 0;

virtual BOOL IsWritable() { return !IsInDeadZone(); }

virtual Var GetValue(BOOL fUpdated) { return nullptr; }

virtual BOOL IsInDeadZone() const { return FALSE; };

};

class IDiagObjectModelWalkerBase

{

public:

// Get the child at i'th position.

virtual BOOL Get(int i, ResolvedObject\* pResolvedObject) = 0;

// Returns number of children for the current diag object.

virtual ulong GetChildrenCount() = 0;

virtual BOOL GetGroupObject(ResolvedObject\* pResolvedObject) = 0;

virtual IDiagObjectAddress \*FindPropertyAddress(PropertyId propId, bool& isConst) { return nullptr;}

};

enum DiagObjectModelDisplayType

{

DiagObjectModelDisplayType\_LocalsDisplay,

DiagObjectModelDisplayType\_RecyclableObjectDisplay,

DiagObjectModelDisplayType\_RecyclableCollectionObjectDisplay,

DiagObjectModelDisplayType\_RecyclableKeyValueDisplay,

};

// Allow getting information across different data types

class IDiagObjectModelDisplay

{

public:

virtual LPCWSTR Name() = 0;

virtual LPCWSTR Type() = 0;

virtual LPCWSTR Value(int radix) = 0;

virtual BOOL HasChildren() = 0;

virtual BOOL Set(Var updateObject) = 0;

virtual DBGPROP\_ATTRIB\_FLAGS GetTypeAttribute() = 0;

virtual BOOL SetDefaultTypeAttribute(DBGPROP\_ATTRIB\_FLAGS attributes) { return FALSE; };

virtual WeakArenaReference<IDiagObjectModelWalkerBase>\* CreateWalker() = 0;

virtual BOOL IsLocalsAsRoot() { return FALSE; }

virtual Var GetVarValue(BOOL fUpdated) { return nullptr; }

virtual IDiagObjectAddress \* GetDiagAddress() { return nullptr; }

virtual DiagObjectModelDisplayType GetType() = 0;

virtual bool IsFake() { return (this->GetTypeAttribute() & DBGPROP\_ATTRIB\_VALUE\_IS\_FAKE) == DBGPROP\_ATTRIB\_VALUE\_IS\_FAKE; }

virtual bool IsLiteralProperty() const = 0;

virtual bool IsSymbolProperty() { return false; }

};

//

// There are three distinct types of classes defined in order to inspect a variable on watch/locals window.

// If someone has to change or provide the support for custom types/objects (such as PixelArray etc) be displayed on the debugger, they need to aware

// of few things which are mentioned below.

// <...>Display (eg RecyclableArrayDisplay), mentions how current variable is given to debugger, and tells what walker (enumerator) to be chosen

// in order to walk to children of the current variable.

// <...>Walker (eg RecyclableArrayWalker), mentions logic of walk thru content of the current variable (the object generally acts like an enumerator). Let say for an array, it has logic to go thru each

// indices and populate values.

// <...>Address (eg RecyclableArrayAddress), associated with each child and will be used to updating that item. The object if this will be consumed by "<...>Walker"

// object when it walks thru each children of the current variable.

// In order to support the custom objects, above classes should be used (or derived) to get started.

//

enum DebuggerPropertyDisplayInfoFlags

{

DebuggerPropertyDisplayInfoFlags\_None = 0x0,

DebuggerPropertyDisplayInfoFlags\_Const = 0x1,

DebuggerPropertyDisplayInfoFlags\_InDeadZone = 0x2,

DebuggerPropertyDisplayInfoFlags\_Unscope = 0x4,

};

struct DebuggerPropertyDisplayInfo

{

PropertyId propId;

Var aVar;

DWORD flags; // DebuggerPropertyDisplayInfoFlags.

DebuggerPropertyDisplayInfo(PropertyId \_propId, Var \_aVar, DWORD \_flags) : propId(\_propId), aVar(\_aVar), flags(\_flags)

{}

bool IsUnscoped() const { return (flags & DebuggerPropertyDisplayInfoFlags\_Unscope) != 0; }

bool IsConst() const { return (flags & DebuggerPropertyDisplayInfoFlags\_Const) != 0; }

bool IsInDeadZone() const { return (flags & DebuggerPropertyDisplayInfoFlags\_InDeadZone) != 0; }

};

enum UIGroupType

{

UIGroupType\_None,

UIGroupType\_InnerScope, // variables under the innerscope (such as Block/Catch)

UIGroupType\_Scope,

UIGroupType\_Globals

};

enum FramesLocalType

{

LocalType\_None = 0x0,

LocalType\_Reg = 0x1,

LocalType\_InSlot = 0x2,

LocalType\_InObject = 0x4,

};

enum FrameWalkerFlags

{

FW\_None = 0x0,

FW\_MakeGroups = 0x1, // Make groups such as [Scope], [Globals] etc.

FW\_EnumWithScopeAlso = 0x2, // While walking include the with scope as well.

FW\_AllowLexicalThis = 0x4, // Do not filter out Js::PropertyIds::\_lexicalThisSlotSymbol

FW\_AllowSuperReference = 0x8, // Allow walking of Js::PropertyIds::\_superReferenceSymbol and Js::PropertyIds::\_superCtorReferenceSymbol

};

class VariableWalkerBase : public IDiagObjectModelWalkerBase

{

public:

DiagStackFrame\* pFrame;

Var instance;

JsUtil::List<DebuggerPropertyDisplayInfo\*, ArenaAllocator> \*pMembersList;

UIGroupType groupType;

private:

bool allowLexicalThis;

bool allowSuperReference;

public :

VariableWalkerBase(DiagStackFrame\* \_pFrame, Var \_instance, UIGroupType \_groupType, bool allowLexicalThis, bool allowSuperReference = false)

: pFrame(\_pFrame), instance(\_instance), pMembersList(nullptr), groupType(\_groupType), allowLexicalThis(allowLexicalThis), allowSuperReference(allowSuperReference)

{

}

// Defined virtual function, should be extended by type of variable scope.

virtual void PopulateMembers() { };

virtual IDiagObjectAddress \* GetObjectAddress(int index) { return nullptr; }

virtual Var GetVarObjectAt(int index);

virtual bool IsConstAt(int index);

/// IDiagObjectModelWalkerBase

virtual BOOL Get(int i, ResolvedObject\* pResolvedObject) override;

virtual ulong GetChildrenCount() override;

virtual BOOL GetGroupObject(ResolvedObject\* pResolvedObject) override sealed;

virtual IDiagObjectAddress \*FindPropertyAddress(PropertyId propId, bool& isConst) override;

static BOOL GetExceptionObject(int &index, DiagStackFrame\* frame, ResolvedObject\* pResolvedObject);

static bool HasExceptionObject(DiagStackFrame\* frame);

static BOOL GetReturnedValue(int &index, DiagStackFrame\* frame, ResolvedObject\* pResolvedObject);

static int GetReturnedValueCount(DiagStackFrame\* frame);

#ifdef ENABLE\_MUTATION\_BREAKPOINT

static BOOL GetBreakMutationBreakpointValue(int &index, DiagStackFrame\* frame, ResolvedObject\* pResolvedObject);

static uint GetBreakMutationBreakpointsCount(DiagStackFrame\* frame);

#endif

bool IsInGroup() const { return (groupType != UIGroupType::UIGroupType\_None && groupType != UIGroupType::UIGroupType\_InnerScope); }

bool IsWalkerForCurrentFrame() const { return groupType == UIGroupType::UIGroupType\_None; }

int GetAdjustedByteCodeOffset() const;

DebuggerPropertyDisplayInfo\* AllocateNewPropertyDisplayInfo(PropertyId propertyId, Var value, bool isConst, bool isInDeadZone);

protected:

int GetMemberCount() { return pMembersList ? pMembersList->Count() : 0; }

bool IsPropertyValid(PropertyId propertyId, RegSlot location, bool \*isPropertyInDebuggerScope, bool\* isConst, bool\* isInDeadZone) const;

};

class RegSlotVariablesWalker : public VariableWalkerBase

{

// This will be pointing to the inner debugger scope (block/catch)

DebuggerScope\* debuggerScope;

public:

RegSlotVariablesWalker(DiagStackFrame\* \_pFrame, DebuggerScope \*\_debuggerScope, UIGroupType \_groupType, bool allowSuperReference = false)

: VariableWalkerBase(\_pFrame, nullptr, \_groupType, /\* allowLexicalThis \*/ false, allowSuperReference), debuggerScope(\_debuggerScope)

{

}

virtual void PopulateMembers() override;

virtual IDiagObjectAddress \* GetObjectAddress(int index) override;

virtual Var GetVarObjectAt(int index) override;

private:

bool IsRegisterValid(PropertyId propertyId, RegSlot registerSlot) const;

bool IsRegisterInScope(PropertyId propertyId, RegSlot registerSlot) const;

Var GetVarObjectAndRegAt(int index, RegSlot\* reg = nullptr);

};

class SlotArrayVariablesWalker : public VariableWalkerBase

{

public:

SlotArrayVariablesWalker(DiagStackFrame\* \_pFrame, Var \_instance, UIGroupType \_groupType, bool allowLexicalThis, bool allowSuperReference = false) : VariableWalkerBase(\_pFrame, \_instance, \_groupType, allowLexicalThis, allowSuperReference) {}

virtual void PopulateMembers() override;

virtual IDiagObjectAddress \* GetObjectAddress(int index) override;

ScopeSlots GetSlotArray() {

Var \*slotArray = (Var \*) instance;

Assert(slotArray != nullptr);

return ScopeSlots(slotArray);

}

};

class ObjectVariablesWalker : public VariableWalkerBase

{

public:

ObjectVariablesWalker(DiagStackFrame\* \_pFrame, Var \_instance, UIGroupType \_groupType, bool allowLexicalThis, bool allowSuperReference = false) : VariableWalkerBase(\_pFrame, \_instance, \_groupType, allowLexicalThis, allowSuperReference) {}

virtual void PopulateMembers() override;

virtual IDiagObjectAddress \* GetObjectAddress(int index) override;

protected:

void AddObjectProperties(int count, Js::RecyclableObject\* object);

};

class RootObjectVariablesWalker : public ObjectVariablesWalker

{

public:

RootObjectVariablesWalker(DiagStackFrame\* \_pFrame, Var \_instance, UIGroupType \_groupType) : ObjectVariablesWalker(\_pFrame, \_instance, \_groupType, /\* allowLexicalThis \*/ false) {}

virtual void PopulateMembers() override;

};

class DiagScopeVariablesWalker sealed : public VariableWalkerBase

{

public:

// Represent catch/with scope objects, (ie. the representation for the diagnostics purposes.)

JsUtil::List<IDiagObjectModelWalkerBase\*, ArenaAllocator> \*pDiagScopeObjects;

ulong diagScopeVarCount;

bool scopeIsInitialized;

bool enumWithScopeAlso;

public:

DiagScopeVariablesWalker(DiagStackFrame\* \_pFrame, Var \_instance, bool \_enumWithScopeAlso)

: VariableWalkerBase(\_pFrame, \_instance, UIGroupType\_InnerScope, /\* allowLexicalThis \*/ false), pDiagScopeObjects(nullptr), diagScopeVarCount(0), scopeIsInitialized(false), enumWithScopeAlso(\_enumWithScopeAlso)

{}

DiagScopeVariablesWalker(DiagStackFrame\* \_pFrame, Var \_instance, IDiagObjectModelWalkerBase\* innerWalker);

virtual BOOL Get(int i, ResolvedObject\* pResolvedObject) override;

virtual ulong GetChildrenCount() override;

virtual IDiagObjectAddress \*FindPropertyAddress(PropertyId propId, bool& isConst) override;

};

// Display of variable on the locals window

// Also responsible for walking on the current frame and build up chain of scopes.

class LocalsWalker sealed : public IDiagObjectModelWalkerBase

{

friend class RecyclableArgumentsArrayWalker;

DiagStackFrame\* pFrame;

JsUtil::List<VariableWalkerBase \*, ArenaAllocator> \* pVarWalkers; // This includes, current frame, all scopes and globals for a current frame

uint totalLocalsCount;

DWORD frameWalkerFlags;

// true, if user has not defined the 'arguments' in the script, this is used for displaying a fake arguments object and display in the locals window.

bool hasUserNotDefinedArguments;

public:

LocalsWalker(DiagStackFrame\* \_frame, DWORD \_frameWalkerFlags);

virtual BOOL Get(int i, ResolvedObject\* pResolvedObject) override;

virtual ulong GetChildrenCount() override;

virtual BOOL GetGroupObject(ResolvedObject\* pResolvedObject) {return FALSE; }

static DWORD GetCurrentFramesLocalsType(DiagStackFrame\* frame);

IDiagObjectAddress \* FindPropertyAddress(PropertyId propId, bool& isConst) override;

// enumerateGroups will be true for the when fetching from a variable from the expression evaluation.

IDiagObjectAddress \* FindPropertyAddress(PropertyId propId, bool enumerateGroups, bool& isConst);

template <typename FnProcessResolvedObject>

DynamicObject\* CreateAndPopulateActivationObject(ScriptContext\* scriptContext, FnProcessResolvedObject processResolvedObjectFn)

{

Assert(scriptContext);

Js::DynamicObject\* activeScopeObject = nullptr;

ulong count = this->GetChildrenCount();

if (count > 0)

{

activeScopeObject = scriptContext->GetLibrary()->CreateActivationObject();

for (ulong i = 0; i < count; i++)

{

Js::ResolvedObject resolveObject;

if (this->Get(i, &resolveObject) && resolveObject.propId != Js::Constants::NoProperty)

{

if (!activeScopeObject->HasOwnProperty(resolveObject.propId))

{

OUTPUT\_TRACE(Js::ConsoleScopePhase, L"Adding '%s' property to activeScopeObject\n", resolveObject.scriptContext->GetPropertyName(resolveObject.propId)->GetBuffer());

if (resolveObject.IsInDeadZone())

{

PropertyOperationFlags flags = static\_cast<PropertyOperationFlags>(PropertyOperation\_SpecialValue | PropertyOperation\_AllowUndecl);

PropertyAttributes attributes = resolveObject.isConst ? PropertyConstDefaults : PropertyLetDefaults;

resolveObject.obj = scriptContext->GetLibrary()->GetUndeclBlockVar();

activeScopeObject->SetPropertyWithAttributes(

resolveObject.propId,

resolveObject.obj,

attributes, nullptr, flags);

}

else

{

activeScopeObject->SetPropertyWithAttributes(

resolveObject.propId,

JavascriptOperators::BoxStackInstance(resolveObject.obj, scriptContext), //The value escapes, box if necessary.

resolveObject.isConst ? PropertyConstDefaults : PropertyDynamicTypeDefaults,

nullptr);

}

processResolvedObjectFn(resolveObject);

}

}

}

}

return activeScopeObject;

}

private:

BOOL CreateArgumentsObject(ResolvedObject\* pResolvedObject);

bool ShouldMakeGroups() const { return frameWalkerFlags & FW\_MakeGroups; }

bool ShouldInsertFakeArguments();

void ExpandArgumentsObject(IDiagObjectModelDisplay \* argumentsDisplay);

};

class LocalsDisplay : public IDiagObjectModelDisplay

{

DiagStackFrame\* pFrame;

public:

LocalsDisplay(DiagStackFrame\* \_frame);

virtual LPCWSTR Name() override;

virtual LPCWSTR Type() override;

virtual LPCWSTR Value(int radix) override;

virtual BOOL HasChildren() override;

virtual BOOL Set(Var updateObject) override;

virtual DBGPROP\_ATTRIB\_FLAGS GetTypeAttribute() override;

virtual WeakArenaReference<IDiagObjectModelWalkerBase>\* CreateWalker() override;

virtual BOOL IsLocalsAsRoot() { return TRUE; }

virtual DiagObjectModelDisplayType GetType() { return DiagObjectModelDisplayType\_LocalsDisplay; }

virtual bool IsLiteralProperty() const { return false; }

};

//

// The locals var's addresses.

// A representation of a address when this Var is taken from the slot array.

class LocalObjectAddressForSlot : public IDiagObjectAddress

{

ScopeSlots slotArray;

int slotIndex;

Var value;

public:

LocalObjectAddressForSlot(ScopeSlots \_pSlotArray, int \_slotIndex, Js::Var \_value);

virtual BOOL Set(Var updateObject) override;

virtual Var GetValue(BOOL fUpdated);

virtual BOOL IsInDeadZone() const;

};

// A representation of a address when this Var is taken from the direct regslot.

class LocalObjectAddressForRegSlot : public IDiagObjectAddress

{

DiagStackFrame\* pFrame;

RegSlot regSlot;

Var value;

public:

LocalObjectAddressForRegSlot(DiagStackFrame\* \_pFrame, RegSlot \_regSlot, Js::Var \_value);

virtual BOOL Set(Var updateObject) override;

virtual Var GetValue(BOOL fUpdated);

BOOL IsInDeadZone() const;

};

class CatchScopeWalker sealed : public IDiagObjectModelWalkerBase

{

DiagStackFrame\* pFrame;

DebuggerScope \* debuggerScope;

public :

CatchScopeWalker(DiagStackFrame\* \_pFrame, DebuggerScope\* \_debuggerScope)

: pFrame(\_pFrame), debuggerScope(\_debuggerScope)

{

}

/// IDiagObjectModelWalkerBase

virtual BOOL Get(int i, ResolvedObject\* pResolvedObject) override;

virtual ulong GetChildrenCount() override;

virtual BOOL GetGroupObject(ResolvedObject\* pResolvedObject) override { return FALSE; }

virtual IDiagObjectAddress \*FindPropertyAddress(PropertyId propId, bool& isConst) override;

private:

void FetchValueAndAddress(DebuggerScopeProperty &scopeProperty, \_Out\_opt\_ Var \*pValue, \_Out\_opt\_ IDiagObjectAddress \*\* ppAddress);

};

// Concrete Classes for Objects

class RecyclableObjectWalker : public IDiagObjectModelWalkerBase

{

protected:

ScriptContext\* scriptContext;

Var instance;

Var originalInstance; // Remember original instance for prototype walk, because evaluating getters in CallGetter() if \_\_proto\_\_ instance is passed does not work

JsUtil::List<DebuggerPropertyDisplayInfo \*, ArenaAllocator> \* pMembersList;

RecyclableArrayWalker \* innerArrayObjectWalker; // Will be used for array indices on the object

JsUtil::List<IDiagObjectModelWalkerBase \*, ArenaAllocator> \* fakeGroupObjectWalkerList; // such as [prototype], [Methods] etc.

void InsertItem(Js::RecyclableObject \*pOriginalObject, Js::RecyclableObject \*pObject, PropertyId propertyId, bool isConst, bool isUnscoped, Js::RecyclableMethodsGroupWalker \*\*ppMethodsGrouptWalker, bool shouldPinProperty = false);

void InsertItem(PropertyId propertyId, bool isConst, bool isUnscoped, Var itemObj, Js::RecyclableMethodsGroupWalker \*\*ppMethodsGrouptWalker, bool shouldPinProperty = false);

void EnsureFakeGroupObjectWalkerList();

public:

RecyclableObjectWalker(ScriptContext\* pContext, Var slot);

RecyclableObjectWalker(ScriptContext\* pContext, Var slot, Var originalInstance);

virtual BOOL Get(int i, ResolvedObject\* pResolvedObject) override;

virtual ulong GetChildrenCount() override;

virtual BOOL GetGroupObject(ResolvedObject\* pResolvedObject) { return FALSE; };

virtual IDiagObjectAddress \*FindPropertyAddress(PropertyId propertyId, bool& isConst) override;

static Var GetObject(RecyclableObject\* originalInstance, RecyclableObject\* instance, PropertyId propertyId, ScriptContext\* scriptContext);

};

class RecyclableObjectAddress : public IDiagObjectAddress

{

Var parentObj;

Js::PropertyId propId;

Js::Var value;

BOOL isInDeadZone;

public:

RecyclableObjectAddress(Var parentObj, Js::PropertyId \_propId, Js::Var \_value, BOOL \_isInDeadZone);

virtual BOOL Set(Var updateObject) override;

virtual BOOL IsWritable() override;

virtual Var GetValue(BOOL fUpdated);

BOOL IsInDeadZone() const;

};

class RecyclableObjectDisplay : public IDiagObjectModelDisplay

{

protected:

ScriptContext\* scriptContext;

Var instance;

Var originalInstance;

LPCWSTR name;

IDiagObjectAddress\* pObjAddress;

DBGPROP\_ATTRIB\_FLAGS defaultAttributes;

PropertyId propertyId;

public:

RecyclableObjectDisplay(ResolvedObject\* resolvedObject, DBGPROP\_ATTRIB\_FLAGS defaultAttributes = DBGPROP\_ATTRIB\_NO\_ATTRIB);

virtual LPCWSTR Name() override;

virtual LPCWSTR Type() override;

virtual LPCWSTR Value(int radix) override;

virtual BOOL HasChildren() override;

virtual BOOL Set(Var updateObject) override;

virtual BOOL SetDefaultTypeAttribute(DBGPROP\_ATTRIB\_FLAGS attributes) override { defaultAttributes = attributes; return TRUE; };

virtual DBGPROP\_ATTRIB\_FLAGS GetTypeAttribute() override;

virtual WeakArenaReference<IDiagObjectModelWalkerBase>\* CreateWalker() override;

virtual Var GetVarValue(BOOL fUpdated) override;

virtual IDiagObjectAddress \* GetDiagAddress() override { return pObjAddress; }

virtual DiagObjectModelDisplayType GetType() { return DiagObjectModelDisplayType\_RecyclableObjectDisplay; }

virtual bool IsLiteralProperty() const;

virtual bool IsSymbolProperty() override;

static BOOL GetPropertyWithScriptEnter(RecyclableObject\* originalInstance, RecyclableObject\* instance, PropertyId propertyId, Var\* value, ScriptContext\* scriptContext);

StringBuilder<ArenaAllocator>\* GetStringBuilder();

PropertyId GetPropertyId() const;

};

// Concrete classes for Arrays

class RecyclableArrayAddress : public IDiagObjectAddress

{

protected:

Var parentArray;

unsigned int index;

public:

RecyclableArrayAddress(Var parentArray, unsigned int index);

virtual BOOL Set(Var updateObject) override;

};

class RecyclableArrayDisplay : public RecyclableObjectDisplay

{

protected:

BOOL HasChildrenInternal(Js::JavascriptArray\* arrayObj);

public:

RecyclableArrayDisplay(ResolvedObject\* resolvedObject);

virtual BOOL HasChildren() override;

virtual WeakArenaReference<IDiagObjectModelWalkerBase>\* CreateWalker() override;

};

class RecyclableArrayWalker : public RecyclableObjectWalker

{

protected:

uint32 indexedItemCount;

JsUtil::List<uint32, ArenaAllocator> \* pAbsoluteIndexList;

// Just populate the indexes only.

bool fOnlyOwnProperties;

uint32 RecyclableArrayWalker::GetItemCount(Js::JavascriptArray\* arrayObj);

// ES5Array will extend this.

virtual uint32 GetNextDescriptor(uint32 currentDescriptor) { return Js::JavascriptArray::InvalidIndex; }

LPCWSTR RecyclableArrayWalker::GetIndexName(uint32 index, StringBuilder<ArenaAllocator>\* stringBuilder);

Js::JavascriptArray\* GetArrayObject();

public:

RecyclableArrayWalker(ScriptContext\* pContext, Var slot, Var originalInstance);

void SetOnlyWalkOwnProperties(bool set) { fOnlyOwnProperties = set; }

virtual BOOL Get(int i, ResolvedObject\* pResolvedObject) override;

virtual ulong GetChildrenCount() override;

virtual BOOL FetchItemAtIndex(Js::JavascriptArray\* arrayObj, uint32 index, Var \*value);

virtual Var FetchItemAt(Js::JavascriptArray\* arrayObj, uint32 index);

virtual BOOL GetResolvedObject(Js::JavascriptArray\* arrayObj, int index, ResolvedObject\* pResolvedObject, uint32 \* pabsIndex) sealed;

StringBuilder<ArenaAllocator>\* GetBuilder();

};

// Concrete classes for Arguments object

//

class RecyclableArgumentsObjectDisplay : public RecyclableObjectDisplay

{

LocalsWalker \*pLocalsWalker;

public:

RecyclableArgumentsObjectDisplay(ResolvedObject\* resolvedObject, LocalsWalker \*localsWalker);

virtual BOOL HasChildren() override;

virtual WeakArenaReference<IDiagObjectModelWalkerBase>\* CreateWalker() override;

};

class RecyclableArgumentsObjectWalker : public RecyclableObjectWalker

{

LocalsWalker \*pLocalsWalker;

public:

RecyclableArgumentsObjectWalker(ScriptContext\* pContext, Var instance, LocalsWalker \* localsWalker);

virtual ulong GetChildrenCount() override;

};

class RecyclableArgumentsArrayAddress : public IDiagObjectAddress

{

Var parentArray;

unsigned int index;

public:

RecyclableArgumentsArrayAddress(Var parentArray, unsigned int index);

virtual BOOL Set(Var updateObject) override;

};

class RecyclableArgumentsArrayWalker : public RecyclableArrayWalker

{

JsUtil::List<IDiagObjectAddress \*, ArenaAllocator> \* pFormalsList;

public:

RecyclableArgumentsArrayWalker(ScriptContext\* pContext, Var slot, Var originalInstance);

virtual BOOL Get(int i, ResolvedObject\* pResolvedObject) override;

virtual ulong GetChildrenCount() override;

void FetchFormalsAddress (LocalsWalker \* localsWalker);

};

// Concrete classes for Typed array objects

//

class RecyclableTypedArrayAddress : public RecyclableArrayAddress

{

public:

RecyclableTypedArrayAddress(Var parentArray, unsigned int index);

virtual BOOL Set(Var updateObject) override;

};

class RecyclableTypedArrayDisplay : public RecyclableObjectDisplay

{

public:

RecyclableTypedArrayDisplay(ResolvedObject\* resolvedObject);

virtual BOOL HasChildren() override;

virtual WeakArenaReference<IDiagObjectModelWalkerBase>\* CreateWalker() override;

};

class RecyclableTypedArrayWalker : public RecyclableArrayWalker

{

public:

RecyclableTypedArrayWalker(ScriptContext\* pContext, Var slot, Var originalInstance);

virtual BOOL Get(int i, ResolvedObject\* pResolvedObject) override;

virtual ulong GetChildrenCount() override;

};

// Concrete classes for Pixel array objects

//

class RecyclablePixelArrayAddress : public RecyclableArrayAddress

{

public:

RecyclablePixelArrayAddress(Var parentArray, unsigned int index);

virtual BOOL Set(Var updateObject) override;

};

class RecyclablePixelArrayDisplay : public RecyclableObjectDisplay

{

public:

RecyclablePixelArrayDisplay(ResolvedObject\* resolvedObject);

virtual BOOL HasChildren() override;

virtual WeakArenaReference<IDiagObjectModelWalkerBase>\* CreateWalker() override;

};

class RecyclablePixelArrayWalker : public RecyclableArrayWalker

{

public:

RecyclablePixelArrayWalker(ScriptContext\* pContext, Var slot, Var originalInstance);

virtual BOOL Get(int i, ResolvedObject\* pResolvedObject) override;

virtual ulong GetChildrenCount() override;

};

// Concrete classes for ES5 array objects

//

class RecyclableES5ArrayAddress : public RecyclableArrayAddress

{

public:

RecyclableES5ArrayAddress(Var parentArray, unsigned int index);

virtual BOOL Set(Var updateObject) override;

};

class RecyclableES5ArrayDisplay : public RecyclableArrayDisplay

{

public:

RecyclableES5ArrayDisplay(ResolvedObject\* resolvedObject);

virtual BOOL HasChildren() override;

virtual WeakArenaReference<IDiagObjectModelWalkerBase>\* CreateWalker() override;

};

class RecyclableES5ArrayWalker sealed : public RecyclableArrayWalker

{

public:

RecyclableES5ArrayWalker(ScriptContext\* pContext, Var slot, Var originalInstance);

virtual uint32 GetNextDescriptor(uint32 currentDescriptor) override;

virtual BOOL FetchItemAtIndex(Js::JavascriptArray\* arrayObj, uint32 index, Var \*value) override;

virtual Var FetchItemAt(Js::JavascriptArray\* arrayObj, uint32 index) override;

};

// Concrete classes for Proto group object

//

class RecyclableProtoObjectWalker : public RecyclableObjectWalker

{

public:

RecyclableProtoObjectWalker(ScriptContext\* pContext, Var slot, Var originalInstance);

virtual BOOL GetGroupObject(ResolvedObject\* pResolvedObject) override;

virtual IDiagObjectAddress \*FindPropertyAddress(PropertyId propId, bool& isConst) override;

};

class RecyclableProtoObjectAddress : public RecyclableObjectAddress

{

public:

RecyclableProtoObjectAddress(Var \_parentObj, Js::PropertyId \_propId, Js::Var \_value);

};

// Concrete classes for Map, Set, and WeakMap group objects

//

template <typename TData>

struct RecyclableCollectionObjectWalkerPropertyData

{

RecyclableCollectionObjectWalkerPropertyData():key(nullptr), value(nullptr) { }

RecyclableCollectionObjectWalkerPropertyData(Var key, Var value):key(key), value(value) { }

Var key;

Var value;

};

template<>

struct RecyclableCollectionObjectWalkerPropertyData<JavascriptSet>

{

RecyclableCollectionObjectWalkerPropertyData():value(nullptr) { }

RecyclableCollectionObjectWalkerPropertyData(Var value):value(value) { }

Var value;

};

template<>

struct RecyclableCollectionObjectWalkerPropertyData<JavascriptWeakSet>

{

RecyclableCollectionObjectWalkerPropertyData():value(nullptr) { }

RecyclableCollectionObjectWalkerPropertyData(Var value):value(value) { }

Var value;

};

template <typename TData>

class RecyclableCollectionObjectWalker : public IDiagObjectModelWalkerBase

{

ScriptContext\* scriptContext;

Var instance;

JsUtil::List<RecyclableCollectionObjectWalkerPropertyData<TData>, ArenaAllocator>\* propertyList;

const wchar\_t\* Name();

IDiagObjectModelDisplay\* CreateTDataDisplay(ResolvedObject\* resolvedObject, int i);

void GetChildren();

public:

RecyclableCollectionObjectWalker(ScriptContext\* scriptContext, Var instance):scriptContext(scriptContext), instance(instance), propertyList(nullptr) { }

virtual BOOL GetGroupObject(ResolvedObject\* pResolvedObject) override;

virtual BOOL Get(int i, ResolvedObject\* pResolvedObject) override;

virtual ulong GetChildrenCount() override;

};

typedef RecyclableCollectionObjectWalker<JavascriptMap> RecyclableMapObjectWalker;

typedef RecyclableCollectionObjectWalker<JavascriptSet> RecyclableSetObjectWalker;

typedef RecyclableCollectionObjectWalker<JavascriptWeakMap> RecyclableWeakMapObjectWalker;

typedef RecyclableCollectionObjectWalker<JavascriptWeakSet> RecyclableWeakSetObjectWalker;

template <typename TData>

class RecyclableCollectionObjectDisplay : public IDiagObjectModelDisplay

{

ScriptContext\* scriptContext;

const wchar\_t\* name;

RecyclableCollectionObjectWalker<TData>\* walker;

public:

RecyclableCollectionObjectDisplay(ScriptContext\* scriptContext, const wchar\_t\* name, RecyclableCollectionObjectWalker<TData>\* walker) : scriptContext(scriptContext), name(name), walker(walker) { }

virtual LPCWSTR Name() override { return name; }

virtual LPCWSTR Type() override { return L""; }

virtual LPCWSTR Value(int radix) override;

virtual BOOL HasChildren() override { return walker->GetChildrenCount() > 0; }

virtual BOOL Set(Var updateObject) override { return FALSE; }

virtual BOOL SetDefaultTypeAttribute(DBGPROP\_ATTRIB\_FLAGS attributes) override { return FALSE; }

virtual DBGPROP\_ATTRIB\_FLAGS GetTypeAttribute() override { return DBGPROP\_ATTRIB\_VALUE\_READONLY | DBGPROP\_ATTRIB\_VALUE\_IS\_FAKE | (HasChildren() ? DBGPROP\_ATTRIB\_VALUE\_IS\_EXPANDABLE : 0); }

virtual WeakArenaReference<IDiagObjectModelWalkerBase>\* CreateWalker() override;

virtual Var GetVarValue(BOOL fUpdated) override { return nullptr; }

virtual IDiagObjectAddress \* GetDiagAddress() override { return nullptr; }

virtual DiagObjectModelDisplayType GetType() { return DiagObjectModelDisplayType\_RecyclableCollectionObjectDisplay; }

virtual bool IsLiteralProperty() const { return false; }

};

class RecyclableKeyValueDisplay : public IDiagObjectModelDisplay

{

ScriptContext\* scriptContext;

Var key;

Var value;

const wchar\_t\* name;

public:

RecyclableKeyValueDisplay(ScriptContext\* scriptContext, Var key, Var value, const wchar\_t\* name) : scriptContext(scriptContext), key(key), value(value), name(name) { }

virtual LPCWSTR Name() override { return name; }

virtual LPCWSTR Type() override { return L""; }

virtual LPCWSTR Value(int radix) override;

virtual BOOL HasChildren() override { return TRUE; }

virtual BOOL Set(Var updateObject) override { return FALSE; }

virtual DBGPROP\_ATTRIB\_FLAGS GetTypeAttribute() override { return DBGPROP\_ATTRIB\_VALUE\_IS\_EXPANDABLE | DBGPROP\_ATTRIB\_VALUE\_IS\_FAKE | DBGPROP\_ATTRIB\_VALUE\_READONLY; }

virtual WeakArenaReference<IDiagObjectModelWalkerBase>\* CreateWalker() override;

virtual DiagObjectModelDisplayType GetType() { return DiagObjectModelDisplayType\_RecyclableKeyValueDisplay; }

virtual bool IsLiteralProperty() const { return false; }

};

class RecyclableKeyValueWalker : public IDiagObjectModelWalkerBase

{

ScriptContext\* scriptContext;

Var key;

Var value;

public:

RecyclableKeyValueWalker(ScriptContext\* scriptContext, Var key, Var value):scriptContext(scriptContext), key(key), value(value) { }

virtual BOOL Get(int i, ResolvedObject\* pResolvedObject) override;

virtual ulong GetChildrenCount() override { return 2; }

virtual BOOL GetGroupObject(ResolvedObject\* pResolvedObject) override { return FALSE; }

};

class RecyclableProxyObjectDisplay : public RecyclableObjectDisplay

{

public:

RecyclableProxyObjectDisplay(ResolvedObject\* resolvedObject);

virtual BOOL HasChildren() override { return TRUE; }

virtual WeakArenaReference<IDiagObjectModelWalkerBase>\* CreateWalker() override;

};

class RecyclableProxyObjectWalker : public RecyclableObjectWalker

{

public:

RecyclableProxyObjectWalker(ScriptContext\* pContext, Var instance);

virtual BOOL Get(int i, ResolvedObject\* pResolvedObject) override;

virtual ulong GetChildrenCount() override { return 2; }

virtual BOOL GetGroupObject(ResolvedObject\* pResolvedObject) override;

};

// Concrete classes for Methods group object

//

class RecyclableMethodsGroupWalker : public RecyclableObjectWalker

{

public:

RecyclableMethodsGroupWalker(ScriptContext\* pContext, Var slot);

void AddItem(Js::PropertyId propertyId, Var obj);

virtual BOOL Get(int i, ResolvedObject\* pResolvedObject) override;

virtual ulong GetChildrenCount() override;

virtual BOOL GetGroupObject(ResolvedObject\* pResolvedObject) override;

void Sort();

};

class RecyclableMethodsGroupDisplay : public RecyclableObjectDisplay

{

public:

RecyclableMethodsGroupWalker \*methodGroupWalker;

RecyclableMethodsGroupDisplay(RecyclableMethodsGroupWalker \*\_methodGroupWalker, ResolvedObject\* resolvedObject);

virtual LPCWSTR Type() override;

virtual LPCWSTR Value(int radix) override;

virtual BOOL HasChildren() override;

virtual DBGPROP\_ATTRIB\_FLAGS GetTypeAttribute() override;

virtual WeakArenaReference<IDiagObjectModelWalkerBase>\* CreateWalker() override;

};

// Concrete classes for Scope group object

//

class ScopeVariablesGroupDisplay : public RecyclableObjectDisplay

{

public:

VariableWalkerBase \*scopeGroupWalker;

ScopeVariablesGroupDisplay(VariableWalkerBase \*walker, ResolvedObject\* resolvedObject);

virtual LPCWSTR Type() override;

virtual LPCWSTR Value(int radix) override;

virtual BOOL HasChildren() override;

virtual DBGPROP\_ATTRIB\_FLAGS GetTypeAttribute() override;

virtual WeakArenaReference<IDiagObjectModelWalkerBase>\* CreateWalker() override;

};

// Concrete classes for Globals group object

//

class GlobalsScopeVariablesGroupDisplay sealed : public RecyclableObjectDisplay

{

public:

VariableWalkerBase \*globalsGroupWalker;

GlobalsScopeVariablesGroupDisplay(VariableWalkerBase \*walker, ResolvedObject\* resolvedObject);

virtual LPCWSTR Type() override;

virtual LPCWSTR Value(int radix) override;

virtual BOOL HasChildren() override;

virtual DBGPROP\_ATTRIB\_FLAGS GetTypeAttribute() override;

virtual WeakArenaReference<IDiagObjectModelWalkerBase>\* CreateWalker() override;

};

#ifdef ENABLE\_MUTATION\_BREAKPOINT

// For Pending Mutation breakpoint

class PendingMutationBreakpointDisplay : public RecyclableObjectDisplay

{

MutationType mutationType;

public:

PendingMutationBreakpointDisplay(ResolvedObject\* resolvedObject, MutationType mutationType);

virtual LPCWSTR Value(int radix) override { return L""; }

virtual BOOL HasChildren() override { return TRUE; }

virtual WeakArenaReference<IDiagObjectModelWalkerBase>\* CreateWalker() override;

};

class PendingMutationBreakpointWalker : public RecyclableObjectWalker

{

MutationType mutationType;

public:

PendingMutationBreakpointWalker(ScriptContext\* pContext, Var instance, MutationType mutationType);

virtual BOOL Get(int i, ResolvedObject\* pResolvedObject) override;

virtual ulong GetChildrenCount() override;

};

#endif

}

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

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

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

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

#include "RuntimeDebugPch.h"

// Parser includes

#include "CharClassifier.h"

// TODO: clean up the need of these regex related header here just for GroupInfo needed in JavascriptRegExpConstructor

#include "RegexCommon.h"

// Runtime includes

#include "Library\ObjectPrototypeObject.h"

#include "Library\JavascriptNumberObject.h"

#include "Library\BoundFunction.h"

#include "Library\JavascriptRegExpConstructor.h"

#include "Library\SameValueComparer.h"

#include "Library\MapOrSetDataList.h"

#include "Library\JavascriptProxy.h"

#include "Library\JavascriptMap.h"

#include "Library\JavascriptSet.h"

#include "Library\JavascriptWeakMap.h"

#include "Library\JavascriptWeakSet.h"

#include "Library\ArgumentsObject.h"

#include "Types\DynamicObjectEnumerator.h"

#include "Types\DynamicObjectSnapshotEnumerator.h"

#include "Types\DynamicObjectSnapshotEnumeratorWPCache.h"

#include "Library\ForInObjectEnumerator.h"

#include "Library\ES5Array.h"

// Other includes

#include <shlwapi.h>

#include <strsafe.h>

namespace Js

{

#define RETURN\_VALUE\_MAX\_NAME 255

#define PENDING\_MUTATION\_VALUE\_MAX\_NAME 255

//

// Some helper routines

int \_\_cdecl ElementsComparer(\_\_in void\* context, \_\_in const void\* item1, \_\_in const void\* item2)

{

ScriptContext \*scriptContext = (ScriptContext \*)context;

Assert(scriptContext);

const DWORD\_PTR \*p1 = reinterpret\_cast<const DWORD\_PTR\*>(item1);

const DWORD\_PTR \*p2 = reinterpret\_cast<const DWORD\_PTR\*>(item2);

DebuggerPropertyDisplayInfo \* pPVItem1 = (DebuggerPropertyDisplayInfo \*)(\*p1);

DebuggerPropertyDisplayInfo \* pPVItem2 = (DebuggerPropertyDisplayInfo \*)(\*p2);

const Js::PropertyRecord \*propertyRecord1 = scriptContext->GetPropertyName(pPVItem1->propId);

const Js::PropertyRecord \*propertyRecord2 = scriptContext->GetPropertyName(pPVItem2->propId);

const wchar\_t \*str1 = propertyRecord1->GetBuffer();

const wchar\_t \*str2 = propertyRecord2->GetBuffer();

// Do the natural comparison, for example test2 comes before test11.

return StrCmpLogicalW(str1, str2);

}

ArenaAllocator \*GetArenaFromContext(ScriptContext \*scriptContext)

{

Assert(scriptContext);

return scriptContext->GetThreadContext()->GetDebugManager()->GetDiagnosticArena()->Arena();

}

template <class T>

WeakArenaReference<IDiagObjectModelWalkerBase>\* CreateAWalker(ScriptContext \* scriptContext, Var instance, Var originalInstance)

{

ReferencedArenaAdapter\* pRefArena = scriptContext->GetThreadContext()->GetDebugManager()->GetDiagnosticArena();

if (pRefArena)

{

IDiagObjectModelWalkerBase\* pOMWalker = Anew(pRefArena->Arena(), T, scriptContext, instance, originalInstance);

return HeapNew(WeakArenaReference<IDiagObjectModelWalkerBase>,pRefArena, pOMWalker);

}

return nullptr;

}

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

// ResolvedObject

WeakArenaReference<IDiagObjectModelDisplay>\* ResolvedObject::GetObjectDisplay()

{

AssertMsg(typeId != TypeIds\_HostDispatch, "Bad usage of ResolvedObject::GetObjectDisplay");

IDiagObjectModelDisplay\* pOMDisplay = (this->objectDisplay != nullptr) ? this->objectDisplay : CreateDisplay();

Assert(pOMDisplay);

return HeapNew(WeakArenaReference<IDiagObjectModelDisplay>, scriptContext->GetThreadContext()->GetDebugManager()->GetDiagnosticArena(), pOMDisplay);

}

IDiagObjectModelDisplay \* ResolvedObject::CreateDisplay()

{

IDiagObjectModelDisplay\* pOMDisplay = nullptr;

ReferencedArenaAdapter\* pRefArena = scriptContext->GetThreadContext()->GetDebugManager()->GetDiagnosticArena();

if (Js::TypedArrayBase::Is(obj))

{

pOMDisplay = Anew(pRefArena->Arena(), RecyclableTypedArrayDisplay, this);

}

else if (Js::ES5Array::Is(obj))

{

pOMDisplay = Anew(pRefArena->Arena(), RecyclableES5ArrayDisplay, this);

}

else if (Js::JavascriptArray::Is(obj))

{

// DisableJIT-TODO: Review- is this correct?

#if ENABLE\_COPYONACCESS\_ARRAY

// Make sure any NativeIntArrays are converted

Js::JavascriptLibrary::CheckAndConvertCopyOnAccessNativeIntArray<Var>(obj);

#endif

pOMDisplay = Anew(pRefArena->Arena(), RecyclableArrayDisplay, this);

}

else

{

pOMDisplay = Anew(pRefArena->Arena(), RecyclableObjectDisplay, this);

}

if (this->isConst || this->propId == Js::PropertyIds::\_superReferenceSymbol || this->propId == Js::PropertyIds::\_superReferenceSymbol)

{

pOMDisplay->SetDefaultTypeAttribute(DBGPROP\_ATTRIB\_VALUE\_READONLY);

}

return pOMDisplay;

}

bool ResolvedObject::IsInDeadZone() const

{

Assert(scriptContext);

return this->obj == scriptContext->GetLibrary()->GetDebuggerDeadZoneBlockVariableString();

}

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

// LocalsDisplay

LocalsDisplay::LocalsDisplay(DiagStackFrame\* \_frame)

: pFrame(\_frame)

{

}

LPCWSTR LocalsDisplay::Name()

{

return L"Locals";

}

LPCWSTR LocalsDisplay::Type()

{

return L"";

}

LPCWSTR LocalsDisplay::Value(int radix)

{

return L"Locals";

}

BOOL LocalsDisplay::HasChildren()

{

Js::JavascriptFunction\* func = pFrame->GetJavascriptFunction();

FunctionBody\* function = func->GetFunctionBody();

return function && function->GetLocalsCount() != 0;

}

DBGPROP\_ATTRIB\_FLAGS LocalsDisplay::GetTypeAttribute()

{

return DBGPROP\_ATTRIB\_NO\_ATTRIB;

}

BOOL LocalsDisplay::Set(Var updateObject)

{

// This is the hidden root object for Locals it doesn't get updated.

return FALSE;

}

WeakArenaReference<IDiagObjectModelWalkerBase>\* LocalsDisplay::CreateWalker()

{

ReferencedArenaAdapter\* pRefArena = pFrame->GetScriptContext()->GetThreadContext()->GetDebugManager()->GetDiagnosticArena();

if (pRefArena)

{

IDiagObjectModelWalkerBase \* pOMWalker = nullptr;

BEGIN\_JS\_RUNTIME\_CALL\_EX(pFrame->GetScriptContext(), false);

{

IGNORE\_STACKWALK\_EXCEPTION(scriptContext);

pOMWalker = Anew(pRefArena->Arena(), LocalsWalker, pFrame, FrameWalkerFlags::FW\_MakeGroups);

}

END\_JS\_RUNTIME\_CALL(scriptContext);

return HeapNew(WeakArenaReference<IDiagObjectModelWalkerBase>,pRefArena, pOMWalker);

}

return nullptr;

}

// Variables on the scope or in current function.

/\*static\*/

BOOL VariableWalkerBase::GetExceptionObject(int &index, DiagStackFrame\* frame, ResolvedObject\* pResolvedObject)

{

Assert(pResolvedObject);

Assert(pResolvedObject->scriptContext);

Assert(frame);

Assert(index >= 0);

if (HasExceptionObject(frame))

{

if (index == 0)

{

pResolvedObject->name = L"{exception}";

pResolvedObject->typeId = TypeIds\_Error;

pResolvedObject->address = nullptr;

pResolvedObject->obj = pResolvedObject->scriptContext->GetDebugContext()->GetProbeContainer()->GetExceptionObject();

if (pResolvedObject->obj == nullptr)

{

Assert(false);

pResolvedObject->obj = pResolvedObject->scriptContext->GetLibrary()->GetUndefined();

}

return TRUE;

}

// Adjust the index

index -= 1;

}

return FALSE;

}

/\*static\*/

bool VariableWalkerBase::HasExceptionObject(DiagStackFrame\* frame)

{

Assert(frame);

Assert(frame->GetScriptContext());

return frame->GetScriptContext()->GetDebugContext()->GetProbeContainer()->GetExceptionObject() != nullptr;

}

/\*static\*/

BOOL VariableWalkerBase::GetReturnedValue(int &index, DiagStackFrame\* frame, ResolvedObject\* pResolvedObject)

{

Assert(pResolvedObject);

Assert(pResolvedObject->scriptContext);

Assert(frame);

Assert(index >= 0);

ReturnedValueList \*returnedValueList = frame->GetScriptContext()->GetDebugContext()->GetProbeContainer()->GetReturnedValueList();

if (returnedValueList != nullptr && returnedValueList->Count() > 0 && frame->IsTopFrame())

{

if (index < returnedValueList->Count())

{

DBGPROP\_ATTRIB\_FLAGS defaultAttributes = DBGPROP\_ATTRIB\_VALUE\_IS\_RETURN\_VALUE | DBGPROP\_ATTRIB\_VALUE\_IS\_FAKE;

WCHAR \* finalName = AnewArray(GetArenaFromContext(pResolvedObject->scriptContext), WCHAR, RETURN\_VALUE\_MAX\_NAME);

ReturnedValue \* returnValue = returnedValueList->Item(index);

if (returnValue->isValueOfReturnStatement)

{

swprintf\_s(finalName, RETURN\_VALUE\_MAX\_NAME, L"[Return value]");

pResolvedObject->obj = frame->GetRegValue(Js::FunctionBody::ReturnValueRegSlot);

pResolvedObject->address = Anew(frame->GetArena(), LocalObjectAddressForRegSlot, frame, Js::FunctionBody::ReturnValueRegSlot, pResolvedObject->obj);

}

else

{

if (returnValue->calledFunction->IsScriptFunction())

{

swprintf\_s(finalName, RETURN\_VALUE\_MAX\_NAME, L"[%s returned]", returnValue->calledFunction->GetFunctionBody()->GetDisplayName());

}

else

{

Js::JavascriptString \*builtInName = returnValue->calledFunction->GetDisplayName();

swprintf\_s(finalName, RETURN\_VALUE\_MAX\_NAME, L"[%s returned]", builtInName->GetSz());

}

pResolvedObject->obj = returnValue->returnedValue;

defaultAttributes |= DBGPROP\_ATTRIB\_VALUE\_READONLY;

pResolvedObject->address = nullptr;

}

Assert(pResolvedObject->obj != nullptr);

pResolvedObject->name = finalName;

pResolvedObject->typeId = TypeIds\_Object;

pResolvedObject->objectDisplay = pResolvedObject->CreateDisplay();

pResolvedObject->objectDisplay->SetDefaultTypeAttribute(defaultAttributes);

return TRUE;

}

// Adjust the index

index -= returnedValueList->Count();

}

return FALSE;

}

/\*static\*/

int VariableWalkerBase::GetReturnedValueCount(DiagStackFrame\* frame)

{

Assert(frame);

Assert(frame->GetScriptContext());

ReturnedValueList \*returnedValueList = frame->GetScriptContext()->GetDebugContext()->GetProbeContainer()->GetReturnedValueList();

return returnedValueList != nullptr && frame->IsTopFrame() ? returnedValueList->Count() : 0;

}

#ifdef ENABLE\_MUTATION\_BREAKPOINT

BOOL VariableWalkerBase::GetBreakMutationBreakpointValue(int &index, DiagStackFrame\* frame, ResolvedObject\* pResolvedObject)

{

Assert(pResolvedObject);

Assert(pResolvedObject->scriptContext);

Assert(frame);

Assert(index >= 0);

Js::MutationBreakpoint \*mutationBreakpoint = frame->GetScriptContext()->GetDebugContext()->GetProbeContainer()->GetDebugManager()->GetActiveMutationBreakpoint();

if (mutationBreakpoint != nullptr)

{

if (index == 0)

{

pResolvedObject->name = L"[Pending Mutation]";

pResolvedObject->typeId = TypeIds\_Object;

pResolvedObject->address = nullptr;

pResolvedObject->obj = mutationBreakpoint->GetMutationObjectVar();

ReferencedArenaAdapter\* pRefArena = pResolvedObject->scriptContext->GetThreadContext()->GetDebugManager()->GetDiagnosticArena();

pResolvedObject->objectDisplay = Anew(pRefArena->Arena(), PendingMutationBreakpointDisplay, pResolvedObject, mutationBreakpoint->GetBreakMutationType());

pResolvedObject->objectDisplay->SetDefaultTypeAttribute(DBGPROP\_ATTRIB\_VALUE\_PENDING\_MUTATION | DBGPROP\_ATTRIB\_VALUE\_READONLY | DBGPROP\_ATTRIB\_VALUE\_IS\_FAKE);

return TRUE;

}

index -= 1; // Adjust the index

}

return FALSE;

}

uint VariableWalkerBase::GetBreakMutationBreakpointsCount(DiagStackFrame\* frame)

{

Assert(frame);

Assert(frame->GetScriptContext());

return frame->GetScriptContext()->GetDebugContext()->GetProbeContainer()->GetDebugManager()->GetActiveMutationBreakpoint() != nullptr ? 1 : 0;

}

#endif

BOOL VariableWalkerBase::Get(int i, ResolvedObject\* pResolvedObject)

{

AssertMsg(pResolvedObject, "Bad usage of VariableWalkerBase::Get");

Assert(pFrame);

pResolvedObject->scriptContext = pFrame->GetScriptContext();

if (i < 0)

{

return FALSE;

}

if (GetMemberCount() > i)

{

pResolvedObject->propId = pMembersList->Item(i)->propId;

Assert(pResolvedObject->propId != Js::Constants::NoProperty);

Assert(!Js::IsInternalPropertyId(pResolvedObject->propId));

if (pResolvedObject->propId == Js::PropertyIds::\_superReferenceSymbol || pResolvedObject->propId == Js::PropertyIds::\_superCtorReferenceSymbol)

{

pResolvedObject->name = L"super";

}

else

{

const Js::PropertyRecord\* propertyRecord = pResolvedObject->scriptContext->GetPropertyName(pResolvedObject->propId);

pResolvedObject->name = propertyRecord->GetBuffer();

}

pResolvedObject->obj = GetVarObjectAt(i);

Assert(pResolvedObject->obj);

pResolvedObject->typeId = JavascriptOperators::GetTypeId(pResolvedObject->obj);

pResolvedObject->address = GetObjectAddress(i);

pResolvedObject->isConst = IsConstAt(i);

pResolvedObject->objectDisplay = nullptr;

return TRUE;

}

return FALSE;

}

Var VariableWalkerBase::GetVarObjectAt(int index)

{

Assert(index < pMembersList->Count());

return pMembersList->Item(index)->aVar;

}

bool VariableWalkerBase::IsConstAt(int index)

{

Assert(index < pMembersList->Count());

DebuggerPropertyDisplayInfo\* displayInfo = pMembersList->Item(index);

// Dead zone variables are also displayed as read only.

return displayInfo->IsConst() || displayInfo->IsInDeadZone();

}

ulong VariableWalkerBase::GetChildrenCount()

{

PopulateMembers();

return GetMemberCount();

}

BOOL VariableWalkerBase::GetGroupObject(ResolvedObject\* pResolvedObject)

{

if (!IsInGroup()) return FALSE;

Assert(pResolvedObject);

// This is fake [Methods] object.

pResolvedObject->name = groupType == UIGroupType\_Scope ? L"[Scope]" : L"[Globals]";

pResolvedObject->obj = Js::RecyclableObject::FromVar(instance);

pResolvedObject->typeId = TypeIds\_Function;

pResolvedObject->address = nullptr; // Scope object should not be editable

ArenaAllocator \*arena = GetArenaFromContext(pResolvedObject->scriptContext);

Assert(arena);

if (groupType == UIGroupType\_Scope)

{

pResolvedObject->objectDisplay = Anew(arena, ScopeVariablesGroupDisplay, this, pResolvedObject);

}

else

{

pResolvedObject->objectDisplay = Anew(arena, GlobalsScopeVariablesGroupDisplay, this, pResolvedObject);

}

return TRUE;

}

IDiagObjectAddress \*VariableWalkerBase::FindPropertyAddress(PropertyId propId, bool& isConst)

{

PopulateMembers();

if (pMembersList)

{

for (int i = 0; i < pMembersList->Count(); i++)

{

DebuggerPropertyDisplayInfo \*pair = pMembersList->Item(i);

Assert(pair);

if (pair->propId == propId)

{

isConst = pair->IsConst();

return GetObjectAddress(i);

}

}

}

return nullptr;

}

// Determines if the given property is valid for display in the locals window.

// Cases in which the property is valid are:

// 1. It is not represented by an internal property.

// 2. It is a var property.

// 3. It is a let/const property in scope and is not in a dead zone (assuming isInDeadZone is nullptr).

// (Determines if the given property is currently in block scope and not in a dead zone.)

bool VariableWalkerBase::IsPropertyValid(PropertyId propertyId, RegSlot location, bool \*isPropertyInDebuggerScope, bool\* isConst, bool\* isInDeadZone) const

{

Assert(isPropertyInDebuggerScope);

Assert(isConst);

\*isPropertyInDebuggerScope = false;

// Default to writable (for the case of vars and internal properties).

\*isConst = false;

if (!allowLexicalThis && propertyId == Js::PropertyIds::\_lexicalThisSlotSymbol)

{

return false;

}

if (!allowSuperReference && (propertyId == Js::PropertyIds::\_superReferenceSymbol || propertyId == Js::PropertyIds::\_superCtorReferenceSymbol))

{

return false;

}

if (Js::IsInternalPropertyId(propertyId))

{

return false;

}

Assert(pFrame);

Js::FunctionBody \*pFBody = pFrame->GetJavascriptFunction()->GetFunctionBody();

if (pFBody && pFBody->GetScopeObjectChain())

{

int offset = GetAdjustedByteCodeOffset();

if (pFBody->GetScopeObjectChain()->TryGetDebuggerScopePropertyInfo(

propertyId,

location,

offset,

isPropertyInDebuggerScope,

isConst,

isInDeadZone))

{

return true;

}

}

// If the register was not found in any scopes, then it's a var and should be in scope.

return !\*isPropertyInDebuggerScope;

}

// Gets an adjusted offset for the current bytecode location based on which stack frame we're in.

// If we're in the top frame (leaf node), then the byte code offset should remain as is, to reflect

// the current position of the instruction pointer. If we're not in the top frame, we need to subtract

// 1 as the byte code location will be placed at the next statement to be executed at the top frame.

// In the case of block scoping, this is an inaccurate location for viewing variables since the next

// statement could be beyond the current block scope. For inspection, we want to remain in the

// current block that the function was called from.

// An example is this:

// function foo() { ... } // Frame 0 (with breakpoint inside)

// function bar() { // Frame 1

// {

// let a = 0;

// foo(); // <-- Inspecting here, foo is already evaluated.

// }

// foo(); // <-- Byte code offset is now here, so we need to -1 to get back in the block scope.

int VariableWalkerBase::GetAdjustedByteCodeOffset() const

{

Assert(pFrame);

int offset = pFrame->GetByteCodeOffset();

if (!pFrame->IsTopFrame() && pFrame->IsInterpreterFrame())

{

// Native frames are already adjusted so just need to adjust interpreted

// frames that are not the top frame.

--offset;

}

return offset;

}

// Allocates and returns a property display info.

DebuggerPropertyDisplayInfo\* VariableWalkerBase::AllocateNewPropertyDisplayInfo(PropertyId propertyId, Var value, bool isConst, bool isInDeadZone)

{

Assert(pFrame);

Assert(value);

Assert(isInDeadZone || !pFrame->GetScriptContext()->IsUndeclBlockVar(value));

DWORD flags = DebuggerPropertyDisplayInfoFlags\_None;

flags |= isConst ? DebuggerPropertyDisplayInfoFlags\_Const : 0;

flags |= isInDeadZone ? DebuggerPropertyDisplayInfoFlags\_InDeadZone : 0;

ArenaAllocator \*arena = pFrame->GetArena();

if (isInDeadZone)

{

value = pFrame->GetScriptContext()->GetLibrary()->GetDebuggerDeadZoneBlockVariableString();

}

return Anew(arena, DebuggerPropertyDisplayInfo, propertyId, value, flags);

}

/// Slot array

void SlotArrayVariablesWalker::PopulateMembers()

{

if (pMembersList == nullptr && instance != nullptr)

{

ArenaAllocator \*arena = pFrame->GetArena();

ScopeSlots slotArray = GetSlotArray();

if (slotArray.IsFunctionScopeSlotArray())

{

Js::FunctionBody \*pFBody = slotArray.GetFunctionBody();

if (pFBody->GetPropertyIdsForScopeSlotArray() != nullptr)

{

uint slotArrayCount = slotArray.GetCount();

pMembersList = JsUtil::List<DebuggerPropertyDisplayInfo \*, ArenaAllocator>::New(arena, slotArrayCount);

for (ulong i = 0; i < slotArrayCount; i++)

{

Js::PropertyId propertyId = pFBody->GetPropertyIdsForScopeSlotArray()[i];

bool isConst = false;

bool isPropertyInDebuggerScope = false;

bool isInDeadZone = false;

if (propertyId != Js::Constants::NoProperty && IsPropertyValid(propertyId, i, &isPropertyInDebuggerScope, &isConst, &isInDeadZone))

{

Var value = slotArray.Get(i);

if (pFrame->GetScriptContext()->IsUndeclBlockVar(value))

{

isInDeadZone = true;

}

DebuggerPropertyDisplayInfo \*pair = AllocateNewPropertyDisplayInfo(

propertyId,

value,

isConst,

isInDeadZone);

Assert(pair != nullptr);

pMembersList->Add(pair);

}

}

}

}

else

{

DebuggerScope\* debuggerScope = slotArray.GetDebuggerScope();

AssertMsg(debuggerScope, "Slot array debugger scope is missing but should be created.");

pMembersList = JsUtil::List<DebuggerPropertyDisplayInfo \*, ArenaAllocator>::New(arena);

if (debuggerScope->HasProperties())

{

debuggerScope->scopeProperties->Map([&] (int i, Js::DebuggerScopeProperty& scopeProperty)

{

Var value = slotArray.Get(scopeProperty.location);

bool isConst = scopeProperty.IsConst();

bool isInDeadZone = false;

if (pFrame->GetScriptContext()->IsUndeclBlockVar(value))

{

isInDeadZone = true;

}

DebuggerPropertyDisplayInfo \*pair = AllocateNewPropertyDisplayInfo(

scopeProperty.propId,

value,

isConst,

isInDeadZone);

Assert(pair != nullptr);

pMembersList->Add(pair);

});

}

}

}

}

IDiagObjectAddress \* SlotArrayVariablesWalker::GetObjectAddress(int index)

{

Assert(index < pMembersList->Count());

ScopeSlots slotArray = GetSlotArray();

return Anew(pFrame->GetArena(), LocalObjectAddressForSlot, slotArray, index, pMembersList->Item(index)->aVar);

}

// Regslot

void RegSlotVariablesWalker::PopulateMembers()

{

if (pMembersList == nullptr)

{

Js::FunctionBody \*pFBody = pFrame->GetJavascriptFunction()->GetFunctionBody();

ArenaAllocator \*arena = pFrame->GetArena();

PropertyIdOnRegSlotsContainer \*propIdContainer = pFBody->GetPropertyIdOnRegSlotsContainer();

// this container can be nullptr if there is no locals in current function.

if (propIdContainer != nullptr)

{

pMembersList = JsUtil::List<DebuggerPropertyDisplayInfo \*, ArenaAllocator>::New(arena);

for (uint i = 0; i < propIdContainer->length; i++)

{

Js::PropertyId propertyId;

RegSlot reg;

propIdContainer->FetchItemAt(i, pFBody, &propertyId, &reg);

bool shouldInsert = false;

bool isConst = false;

bool isInDeadZone = false;

if (this->debuggerScope)

{

DebuggerScopeProperty debuggerScopeProperty;

if (this->debuggerScope->TryGetValidProperty(propertyId, reg, GetAdjustedByteCodeOffset(), &debuggerScopeProperty, &isInDeadZone))

{

isConst = debuggerScopeProperty.IsConst();

shouldInsert = true;

}

}

else

{

bool isPropertyInDebuggerScope = false;

shouldInsert = IsPropertyValid(propertyId, reg, &isPropertyInDebuggerScope, &isConst, &isInDeadZone) && !isPropertyInDebuggerScope;

}

if (shouldInsert)

{

Var value = pFrame->GetRegValue(reg);

// If the user didn't supply an arguments object, a fake one will

// be created when evaluating LocalsWalker::ShouldInsertFakeArguments().

if (!(propertyId == PropertyIds::arguments && value == nullptr))

{

if (pFrame->GetScriptContext()->IsUndeclBlockVar(value))

{

isInDeadZone = true;

}

DebuggerPropertyDisplayInfo \*info = AllocateNewPropertyDisplayInfo(

propertyId,

(Var)reg,

isConst,

isInDeadZone);

Assert(info != nullptr);

pMembersList->Add(info);

}

}

}

}

}

}

Var RegSlotVariablesWalker::GetVarObjectAndRegAt(int index, RegSlot\* reg /\*= nullptr\*/)

{

Assert(index < pMembersList->Count());

Var returnedVar = nullptr;

RegSlot returnedReg = Js::Constants::NoRegister;

DebuggerPropertyDisplayInfo\* displayInfo = pMembersList->Item(index);

if (displayInfo->IsInDeadZone())

{

// The uninitialized string is already set in the var for the dead zone display.

Assert(JavascriptString::Is(displayInfo->aVar));

returnedVar = displayInfo->aVar;

}

else

{

returnedReg = ::Math::PointerCastToIntegral<RegSlot>(displayInfo->aVar);

returnedVar = pFrame->GetRegValue(returnedReg);

}

if (reg != nullptr)

{

\*reg = returnedReg;

}

AssertMsg(returnedVar, "Var should be replaced with the dead zone string object.");

return returnedVar;

}

Var RegSlotVariablesWalker::GetVarObjectAt(int index)

{

return GetVarObjectAndRegAt(index);

}

IDiagObjectAddress \* RegSlotVariablesWalker::GetObjectAddress(int index)

{

RegSlot reg = Js::Constants::NoRegister;

Var obj = GetVarObjectAndRegAt(index, &reg);

return Anew(pFrame->GetArena(), LocalObjectAddressForRegSlot, pFrame, reg, obj);

}

// For an activation object.

void ObjectVariablesWalker::PopulateMembers()

{

if (pMembersList == nullptr && instance != nullptr)

{

ScriptContext \* scriptContext = pFrame->GetScriptContext();

ArenaAllocator \*arena = GetArenaFromContext(scriptContext);

Assert(Js::RecyclableObject::Is(instance));

Js::RecyclableObject\* object = Js::RecyclableObject::FromVar(instance);

Assert(JavascriptOperators::IsObject(object));

int count = object->GetPropertyCount();

pMembersList = JsUtil::List<DebuggerPropertyDisplayInfo \*, ArenaAllocator>::New(arena, count);

AddObjectProperties(count, object);

}

}

void ObjectVariablesWalker::AddObjectProperties(int count, Js::RecyclableObject\* object)

{

ScriptContext \* scriptContext = pFrame->GetScriptContext();

// For the scopes and locals only enumerable properties will be shown.

for (int i = 0; i < count; i++)

{

Js::PropertyId propertyId = object->GetPropertyId((PropertyIndex)i);

bool isConst = false;

bool isPropertyInDebuggerScope = false;

bool isInDeadZone = false;

if (propertyId != Js::Constants::NoProperty

&& IsPropertyValid(propertyId, Js::Constants::NoRegister, &isPropertyInDebuggerScope, &isConst, &isInDeadZone)

&& object->IsEnumerable(propertyId))

{

Var itemObj = RecyclableObjectWalker::GetObject(object, object, propertyId, scriptContext);

if (itemObj == nullptr)

{

itemObj = scriptContext->GetLibrary()->GetUndefined();

}

AssertMsg(!RootObjectBase::Is(object) || !isConst, "root object shouldn't produce const properties through IsPropertyValid");

DebuggerPropertyDisplayInfo \*info = AllocateNewPropertyDisplayInfo(

propertyId,

itemObj,

isConst,

isInDeadZone);

Assert(info);

pMembersList->Add(info);

}

}

}

IDiagObjectAddress \* ObjectVariablesWalker::GetObjectAddress(int index)

{

Assert(index < pMembersList->Count());

DebuggerPropertyDisplayInfo\* info = pMembersList->Item(index);

return Anew(pFrame->GetArena(), RecyclableObjectAddress, instance, info->propId, info->aVar, info->IsInDeadZone() ? TRUE : FALSE);

}

// For root access on the Global object (adds let/const variables before properties)

void RootObjectVariablesWalker::PopulateMembers()

{

if (pMembersList == nullptr && instance != nullptr)

{

ScriptContext \* scriptContext = pFrame->GetScriptContext();

ArenaAllocator \*arena = GetArenaFromContext(scriptContext);

Assert(Js::RootObjectBase::Is(instance));

Js::RootObjectBase\* object = Js::RootObjectBase::FromVar(instance);

int count = object->GetPropertyCount();

pMembersList = JsUtil::List<DebuggerPropertyDisplayInfo \*, ArenaAllocator>::New(arena, count);

// Add let/const globals first so that they take precedence over the global properties. Then

// VariableWalkerBase::FindPropertyAddress will correctly find let/const globals that shadow

// global properties of the same name.

object->MapLetConstGlobals([&](const PropertyRecord\* propertyRecord, Var value, bool isConst) {

if (!scriptContext->IsUndeclBlockVar(value))

{

// Let/const are always enumerable and valid

DebuggerPropertyDisplayInfo \*info = AllocateNewPropertyDisplayInfo(propertyRecord->GetPropertyId(), value, isConst, false /\*isInDeadZone\*/);

pMembersList->Add(info);

}

});

AddObjectProperties(count, object);

}

}

// DiagScopeVariablesWalker

DiagScopeVariablesWalker::DiagScopeVariablesWalker(DiagStackFrame\* \_pFrame, Var \_instance, IDiagObjectModelWalkerBase\* innerWalker)

: VariableWalkerBase(\_pFrame, \_instance, UIGroupType\_InnerScope, /\* allowLexicalThis \*/ false)

{

ScriptContext \* scriptContext = \_pFrame->GetScriptContext();

ArenaAllocator \*arena = GetArenaFromContext(scriptContext);

pDiagScopeObjects = JsUtil::List<IDiagObjectModelWalkerBase \*, ArenaAllocator>::New(arena);

pDiagScopeObjects->Add(innerWalker);

diagScopeVarCount = innerWalker->GetChildrenCount();

scopeIsInitialized = true;

}

ulong DiagScopeVariablesWalker::GetChildrenCount()

{

if (scopeIsInitialized)

{

return diagScopeVarCount;

}

Assert(pFrame);

Js::FunctionBody \*pFBody = pFrame->GetJavascriptFunction()->GetFunctionBody();

if (pFBody->GetScopeObjectChain())

{

int bytecodeOffset = GetAdjustedByteCodeOffset();

ScriptContext \* scriptContext = pFrame->GetScriptContext();

ArenaAllocator \*arena = GetArenaFromContext(scriptContext);

pDiagScopeObjects = JsUtil::List<IDiagObjectModelWalkerBase \*, ArenaAllocator>::New(arena);

// Look for catch/with/block scopes which encompass current offset (skip block scopes as

// they are only used for lookup within the RegSlotVariablesWalker).

// Go the reverse way so that we find the innermost scope first;

Js::ScopeObjectChain \* pScopeObjectChain = pFBody->GetScopeObjectChain();

for (int i = pScopeObjectChain->pScopeChain->Count() - 1 ; i >= 0; i--)

{

Js::DebuggerScope \*debuggerScope = pScopeObjectChain->pScopeChain->Item(i);

bool isScopeInRange = debuggerScope->IsOffsetInScope(bytecodeOffset);

if (isScopeInRange && (debuggerScope->IsOwnScope() || debuggerScope->scopeType == DiagBlockScopeDirect))

{

switch (debuggerScope->scopeType)

{

case DiagWithScope:

{

if (enumWithScopeAlso)

{

RecyclableObjectWalker\* recylableObjectWalker = Anew(arena, RecyclableObjectWalker, scriptContext,

(Var)pFrame->GetRegValue(debuggerScope->GetLocation(), true));

pDiagScopeObjects->Add(recylableObjectWalker);

diagScopeVarCount += recylableObjectWalker->GetChildrenCount();

}

}

break;

case DiagCatchScopeDirect:

case DiagCatchScopeInObject:

{

CatchScopeWalker\* catchScopeWalker = Anew(arena, CatchScopeWalker, pFrame, debuggerScope);

pDiagScopeObjects->Add(catchScopeWalker);

diagScopeVarCount += catchScopeWalker->GetChildrenCount();

}

break;

case DiagCatchScopeInSlot:

case DiagBlockScopeInSlot:

{

SlotArrayVariablesWalker\* blockScopeWalker = Anew(arena, SlotArrayVariablesWalker, pFrame,

(Var)pFrame->GetInnerScopeFromRegSlot(debuggerScope->GetLocation()), UIGroupType\_InnerScope, /\* allowLexicalThis \*/ false);

pDiagScopeObjects->Add(blockScopeWalker);

diagScopeVarCount += blockScopeWalker->GetChildrenCount();

}

break;

case DiagBlockScopeDirect:

{

RegSlotVariablesWalker \*pObjWalker = Anew(arena, RegSlotVariablesWalker, pFrame, debuggerScope, UIGroupType\_InnerScope);

pDiagScopeObjects->Add(pObjWalker);

diagScopeVarCount += pObjWalker->GetChildrenCount();

}

break;

case DiagBlockScopeInObject:

{

ObjectVariablesWalker\* objectVariablesWalker = Anew(arena, ObjectVariablesWalker, pFrame, pFrame->GetInnerScopeFromRegSlot(debuggerScope->GetLocation()), UIGroupType\_InnerScope, /\* allowLexicalThis \*/ false);

pDiagScopeObjects->Add(objectVariablesWalker);

diagScopeVarCount += objectVariablesWalker->GetChildrenCount();

}

break;

default:

Assert(false);

}

}

}

}

scopeIsInitialized = true;

return diagScopeVarCount;

}

BOOL DiagScopeVariablesWalker::Get(int i, ResolvedObject\* pResolvedObject)

{

if (i >= 0 && i < (int)diagScopeVarCount)

{

for (int j = 0; j < pDiagScopeObjects->Count(); j++)

{

IDiagObjectModelWalkerBase \*pObjWalker = pDiagScopeObjects->Item(j);

if (i < (int)pObjWalker->GetChildrenCount())

{

return pObjWalker->Get(i, pResolvedObject);

}

i -= (int)pObjWalker->GetChildrenCount();

Assert(i >=0);

}

}

return FALSE;

}

IDiagObjectAddress \* DiagScopeVariablesWalker::FindPropertyAddress(PropertyId propId, bool& isConst)

{

IDiagObjectAddress \* address = nullptr;

// Ensure that children are fetched.

GetChildrenCount();

if (pDiagScopeObjects)

{

for (int j = 0; j < pDiagScopeObjects->Count(); j++)

{

IDiagObjectModelWalkerBase \*pObjWalker = pDiagScopeObjects->Item(j);

Assert(pObjWalker);

address = pObjWalker->FindPropertyAddress(propId, isConst);

if (address != nullptr)

{

break;

}

}

}

return address;

}

// Locals walker

LocalsWalker::LocalsWalker(DiagStackFrame\* \_frame, DWORD \_frameWalkerFlags)

: pFrame(\_frame), frameWalkerFlags(\_frameWalkerFlags), pVarWalkers(nullptr), totalLocalsCount(0), hasUserNotDefinedArguments(false)

{

Js::FunctionBody \*pFBody = pFrame->GetJavascriptFunction()->GetFunctionBody();

if (pFBody && !pFBody->GetUtf8SourceInfo()->GetIsLibraryCode())

{

// Allocate the container of all walkers.

ArenaAllocator \*arena = pFrame->GetArena();

pVarWalkers = JsUtil::List<VariableWalkerBase \*, ArenaAllocator>::New(arena);

VariableWalkerBase \*pVarWalker = nullptr;

// Top most function will have one of these regslot, slotarray or activation object.

FrameDisplay \* pDisplay = pFrame->GetFrameDisplay();

uint scopeCount = (uint)(pDisplay ? pDisplay->GetLength() : 0);

uint nextStartIndex = 0;

// Add the catch/with/block expression scope objects.

if (pFBody->GetScopeObjectChain())

{

pVarWalkers->Add(Anew(arena, DiagScopeVariablesWalker, pFrame, nullptr, !!(frameWalkerFlags & FrameWalkerFlags::FW\_EnumWithScopeAlso)));

}

// In the eval function, we will not show global items directly, instead they should go as a group node.

bool shouldAddGlobalItemsDirectly = pFBody->GetIsGlobalFunc() && !pFBody->IsEval();

if (shouldAddGlobalItemsDirectly)

{

// Global properties will be enumerated using RootObjectVariablesWalker

pVarWalkers->Add(Anew(arena, RootObjectVariablesWalker, pFrame, pFrame->GetRootObject(), UIGroupType\_None));

}

DWORD localsType = GetCurrentFramesLocalsType(pFrame);

if (localsType & FramesLocalType::LocalType\_Reg)

{

pVarWalkers->Add(Anew(arena, RegSlotVariablesWalker, pFrame, nullptr /\*not debugger scope\*/, UIGroupType\_None, !!(frameWalkerFlags & FrameWalkerFlags::FW\_AllowSuperReference)));

}

if (localsType & FramesLocalType::LocalType\_InObject)

{

Assert(scopeCount > 0);

pVarWalker = Anew(arena, ObjectVariablesWalker, pFrame, pDisplay->GetItem(nextStartIndex++), UIGroupType\_None, !!(frameWalkerFlags & FrameWalkerFlags::FW\_AllowLexicalThis), !!(frameWalkerFlags & FrameWalkerFlags::FW\_AllowSuperReference));

}

else if (localsType & FramesLocalType::LocalType\_InSlot)

{

Assert(scopeCount > 0);

pVarWalker = Anew(arena, SlotArrayVariablesWalker, pFrame, (Js::Var \*)pDisplay->GetItem(nextStartIndex++), UIGroupType\_None, !!(frameWalkerFlags & FrameWalkerFlags::FW\_AllowLexicalThis), !!(frameWalkerFlags & FrameWalkerFlags::FW\_AllowSuperReference));

}

else if (scopeCount > 0 && pFBody->GetFrameDisplayRegister() != 0)

{

Assert((Var)pDisplay->GetItem(0) == pFrame->GetScriptContext()->GetLibrary()->GetNull());

// A dummy scope with nullptr register is created. Skip this.

nextStartIndex++;

}

if (pVarWalker)

{

pVarWalkers->Add(pVarWalker);

}

const Js::Var nullVar = pFrame->GetScriptContext()->GetLibrary()->GetNull();

for (uint i = nextStartIndex; i < (uint)scopeCount; i++)

{

Var currentScopeObject = pDisplay->GetItem(i);

if (currentScopeObject != nullptr && currentScopeObject != nullVar) // Skip nullptr (dummy scope)

{

ScopeType scopeType = FrameDisplay::GetScopeType(currentScopeObject);

switch(scopeType)

{

case ScopeType\_ActivationObject:

pVarWalker = Anew(arena, ObjectVariablesWalker, pFrame, currentScopeObject, UIGroupType\_Scope, !!(frameWalkerFlags & FrameWalkerFlags::FW\_AllowLexicalThis), !!(frameWalkerFlags & FrameWalkerFlags::FW\_AllowSuperReference));

pVarWalkers->Add(pVarWalker);

break;

case ScopeType\_SlotArray:

pVarWalker = Anew(arena, SlotArrayVariablesWalker, pFrame, currentScopeObject, UIGroupType\_Scope, !!(frameWalkerFlags & FrameWalkerFlags::FW\_AllowLexicalThis), !!(frameWalkerFlags & FrameWalkerFlags::FW\_AllowSuperReference));

pVarWalkers->Add(pVarWalker);

break;

case ScopeType\_WithScope:

if( (frameWalkerFlags & FrameWalkerFlags::FW\_EnumWithScopeAlso) == FrameWalkerFlags::FW\_EnumWithScopeAlso)

{

RecyclableObjectWalker\* withScopeWalker = Anew(arena, RecyclableObjectWalker, pFrame->GetScriptContext(), currentScopeObject);

pVarWalker = Anew(arena, DiagScopeVariablesWalker, pFrame, currentScopeObject, withScopeWalker);

pVarWalkers->Add(pVarWalker);

}

break;

default:

Assert(false);

}

}

}

// No need to add global properties if this is a global function, as it is already done above.

if (!shouldAddGlobalItemsDirectly)

{

pVarWalker = Anew(arena, RootObjectVariablesWalker, pFrame, pFrame->GetRootObject(), UIGroupType\_Globals);

pVarWalkers->Add(pVarWalker);

}

}

}

BOOL LocalsWalker::CreateArgumentsObject(ResolvedObject\* pResolvedObject)

{

Assert(pResolvedObject);

Assert(pResolvedObject->scriptContext);

Assert(hasUserNotDefinedArguments);

pResolvedObject->name = L"arguments";

pResolvedObject->propId = Js::PropertyIds::arguments;

pResolvedObject->typeId = TypeIds\_Arguments;

Js::FunctionBody \*pFBody = pFrame->GetJavascriptFunction()->GetFunctionBody();

Assert(pFBody);

pResolvedObject->obj = pFrame->GetArgumentsObject();

if (pResolvedObject->obj == nullptr)

{

pResolvedObject->obj = pFrame->CreateHeapArguments();

Assert(pResolvedObject->obj);

pResolvedObject->objectDisplay = Anew(pFrame->GetArena(), RecyclableArgumentsObjectDisplay, pResolvedObject, this);

ExpandArgumentsObject(pResolvedObject->objectDisplay);

}

pResolvedObject->address = Anew(GetArenaFromContext(pResolvedObject->scriptContext),

RecyclableObjectAddress,

pResolvedObject->scriptContext->GetGlobalObject(),

Js::PropertyIds::arguments,

pResolvedObject->obj,

false /\*isInDeadZone\*/);

return TRUE;

}

BOOL LocalsWalker::Get(int i, ResolvedObject\* pResolvedObject)

{

if (i >= (int)totalLocalsCount)

{

return FALSE;

}

pResolvedObject->scriptContext = pFrame->GetScriptContext();

if (VariableWalkerBase::GetExceptionObject(i, pFrame, pResolvedObject))

{

return TRUE;

}

#ifdef ENABLE\_MUTATION\_BREAKPOINT

// Pending mutation display should be before any return value

if (VariableWalkerBase::GetBreakMutationBreakpointValue(i, pFrame, pResolvedObject))

{

return TRUE;

}

#endif

if (VariableWalkerBase::GetReturnedValue(i, pFrame, pResolvedObject))

{

return TRUE;

}

if (hasUserNotDefinedArguments)

{

if (i == 0)

{

return CreateArgumentsObject(pResolvedObject);

}

i--;

}

if (!pVarWalkers || pVarWalkers->Count() == 0)

{

return FALSE;

}

// In the case of not making groups, all variables will be arranged

// as one long list in the locals window.

if (!ShouldMakeGroups())

{

for (int j = 0; j < pVarWalkers->Count(); j++)

{

int count = pVarWalkers->Item(j)->GetChildrenCount();

if (i < count)

{

return pVarWalkers->Item(j)->Get(i, pResolvedObject);

}

i-= count;

}

Assert(FALSE);

return FALSE;

}

int startScopeIndex = 0;

// Need to determine what range of local variables we're in for the requested index.

// Non-grouped local variables are organized with reg slot coming first, then followed by

// scope slot/activation object variables. Catch and with variables follow next

// and group variables are stored last which come from upper scopes that

// are accessed in this function (those passed down as part of a closure).

// Note that all/any/none of these walkers may be present.

// Example variable layout:

// [0-2] - Reg slot vars.

// [3-4] - Scope slot array vars.

// [5-8] - Global vars (stored on the global object as properties).

for (int j = 0; j < pVarWalkers->Count(); ++j)

{

VariableWalkerBase \*variableWalker = pVarWalkers->Item(j);

if (!variableWalker->IsInGroup())

{

int count = variableWalker->GetChildrenCount();

if (i < count)

{

return variableWalker->Get(i, pResolvedObject);

}

i-= count;

startScopeIndex++;

}

else

{

// We've finished with all walkers for the current locals level so

// break out in order to handle the groups.

break;

}

}

// Handle groups.

Assert((i + startScopeIndex) < pVarWalkers->Count());

VariableWalkerBase \*variableWalker = pVarWalkers->Item(i + startScopeIndex);

return variableWalker->GetGroupObject(pResolvedObject);

}

bool LocalsWalker::ShouldInsertFakeArguments()

{

JavascriptFunction\* func = pFrame->GetJavascriptFunction();

if (func->IsScriptFunction()

&& !func->GetFunctionBody()->GetUtf8SourceInfo()->GetIsLibraryCode()

&& !func->GetFunctionBody()->GetIsGlobalFunc())

{

bool isConst = false;

hasUserNotDefinedArguments = (nullptr == FindPropertyAddress(PropertyIds::arguments, false /\*walkers on the current frame\*/, isConst));

}

return hasUserNotDefinedArguments;

}

ulong LocalsWalker::GetChildrenCount()

{

if (totalLocalsCount == 0)

{

if (pVarWalkers)

{

int groupWalkersStartIndex = 0;

for (int i = 0; i < pVarWalkers->Count(); i++)

{

VariableWalkerBase\* variableWalker = pVarWalkers->Item(i);

// In the case of making groups, we want to include any variables that aren't

// part of a group as part of the local variable count.

if (!ShouldMakeGroups() || !variableWalker->IsInGroup())

{

++groupWalkersStartIndex;

totalLocalsCount += variableWalker->GetChildrenCount();

}

}

// Add on the number of groups to display in locals

// (group walkers come after function local walkers).

totalLocalsCount += (pVarWalkers->Count() - groupWalkersStartIndex);

}

if (VariableWalkerBase::HasExceptionObject(pFrame))

{

totalLocalsCount++;

}

#ifdef ENABLE\_MUTATION\_BREAKPOINT

totalLocalsCount += VariableWalkerBase::GetBreakMutationBreakpointsCount(pFrame);

#endif

totalLocalsCount += VariableWalkerBase::GetReturnedValueCount(pFrame);

// Check if needed to add fake arguments.

if (ShouldInsertFakeArguments())

{

// In this case we need to create arguments object explicitly.

totalLocalsCount++;

}

}

return totalLocalsCount;

}

/\*static\*/

DWORD LocalsWalker::GetCurrentFramesLocalsType(DiagStackFrame\* frame)

{

Assert(frame);

FunctionBody \*pFBody = frame->GetJavascriptFunction()->GetFunctionBody();

Assert(pFBody);

DWORD localType = FramesLocalType::LocalType\_None;

if (pFBody->GetFrameDisplayRegister() != 0)

{

if (pFBody->GetObjectRegister() != 0)

{

// current scope is activation object

localType = FramesLocalType::LocalType\_InObject;

}

else

{

if (pFBody->scopeSlotArraySize > 0)

{

localType = FramesLocalType::LocalType\_InSlot;

}

}

}

if (pFBody->GetPropertyIdOnRegSlotsContainer() && pFBody->GetPropertyIdOnRegSlotsContainer()->length > 0)

{

localType |= FramesLocalType::LocalType\_Reg;

}

return localType;

}

IDiagObjectAddress \* LocalsWalker::FindPropertyAddress(PropertyId propId, bool& isConst)

{

return FindPropertyAddress(propId, true, isConst);

}

IDiagObjectAddress \* LocalsWalker::FindPropertyAddress(PropertyId propId, bool enumerateGroups, bool& isConst)

{

isConst = false;

if (propId == PropertyIds::arguments && hasUserNotDefinedArguments)

{

ResolvedObject resolveObject;

resolveObject.scriptContext = pFrame->GetScriptContext();

if (CreateArgumentsObject(&resolveObject))

{

return resolveObject.address;

}

}

if (pVarWalkers)

{

for (int i = 0; i < pVarWalkers->Count(); i++)

{

VariableWalkerBase \*pVarWalker = pVarWalkers->Item(i);

if (!enumerateGroups && !pVarWalker->IsWalkerForCurrentFrame())

{

continue;

}

IDiagObjectAddress \*address = pVarWalkers->Item(i)->FindPropertyAddress(propId, isConst);

if (address != nullptr)

{

return address;

}

}

}

return nullptr;

}

void LocalsWalker::ExpandArgumentsObject(IDiagObjectModelDisplay \* argumentsDisplay)

{

Assert(argumentsDisplay != nullptr);

WeakArenaReference<Js::IDiagObjectModelWalkerBase>\* argumentsObjectWalkerRef = argumentsDisplay->CreateWalker();

Assert(argumentsObjectWalkerRef != nullptr);

IDiagObjectModelWalkerBase \* walker = argumentsObjectWalkerRef->GetStrongReference();

int count = (int)walker->GetChildrenCount();

Js::ResolvedObject tempResolvedObj;

for (int i = 0; i < count; i++)

{

walker->Get(i, &tempResolvedObj);

}

argumentsObjectWalkerRef->ReleaseStrongReference();

HeapDelete(argumentsObjectWalkerRef);

}

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

// LocalObjectAddressForSlot

LocalObjectAddressForSlot::LocalObjectAddressForSlot(ScopeSlots \_pSlotArray, int \_slotIndex, Js::Var \_value)

: slotArray(\_pSlotArray),

slotIndex(\_slotIndex),

value(\_value)

{

}

BOOL LocalObjectAddressForSlot::Set(Var updateObject)

{

if (IsInDeadZone())

{

AssertMsg(FALSE, "Should not be able to set the value of a slot in a dead zone.");

return FALSE;

}

slotArray.Set(slotIndex, updateObject);

return TRUE;

}

Var LocalObjectAddressForSlot::GetValue(BOOL fUpdated)

{

if (!fUpdated || IsInDeadZone())

{

#if DBG

if (IsInDeadZone())

{

// If we're in a dead zone, the value will be the

// [Uninitialized block variable] string.

Assert(JavascriptString::Is(value));

}

#endif // DBG

return value;

}

return slotArray.Get(slotIndex);

}

BOOL LocalObjectAddressForSlot::IsInDeadZone() const

{

Var value = slotArray.Get(slotIndex);

if (!RecyclableObject::Is(value))

{

return FALSE;

}

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

ScriptContext\* scriptContext = obj->GetScriptContext();

return scriptContext->IsUndeclBlockVar(obj) ? TRUE : FALSE;

}

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

// LocalObjectAddressForSlot

LocalObjectAddressForRegSlot::LocalObjectAddressForRegSlot(DiagStackFrame\* \_pFrame, RegSlot \_regSlot, Js::Var \_value)

: pFrame(\_pFrame),

regSlot(\_regSlot),

value(\_value)

{

}

BOOL LocalObjectAddressForRegSlot::IsInDeadZone() const

{

return regSlot == Js::Constants::NoRegister;

}

BOOL LocalObjectAddressForRegSlot::Set(Var updateObject)

{

Assert(pFrame);

if (IsInDeadZone())

{

AssertMsg(FALSE, "Should not be able to set the value of a register in a dead zone.");

return FALSE;

}

pFrame->SetRegValue(regSlot, updateObject);

return TRUE;

}

Var LocalObjectAddressForRegSlot::GetValue(BOOL fUpdated)

{

if (!fUpdated || IsInDeadZone())

{

#if DBG

if (IsInDeadZone())

{

// If we're in a dead zone, the value will be the

// [Uninitialized block variable] string.

Assert(JavascriptString::Is(value));

}

#endif // DBG

return value;

}

Assert(pFrame);

return pFrame->GetRegValue(regSlot);

}

//

// CatchScopeWalker

BOOL CatchScopeWalker::Get(int i, ResolvedObject\* pResolvedObject)

{

Assert(pResolvedObject);

Assert(pFrame);

pResolvedObject->scriptContext = pFrame->GetScriptContext();

Assert(i < (int)GetChildrenCount());

Js::DebuggerScopeProperty scopeProperty = debuggerScope->scopeProperties->Item(i);

pResolvedObject->propId = scopeProperty.propId;

const Js::PropertyRecord\* propertyRecord = pResolvedObject->scriptContext->GetPropertyName(pResolvedObject->propId);

// TODO: If this is a symbol-keyed property, we should indicate that in the name - "Symbol (description)"

pResolvedObject->name = propertyRecord->GetBuffer();

FetchValueAndAddress(scopeProperty, &pResolvedObject->obj, &pResolvedObject->address);

Assert(pResolvedObject->obj);

pResolvedObject->typeId = JavascriptOperators::GetTypeId(pResolvedObject->obj);

pResolvedObject->objectDisplay = Anew(pFrame->GetArena(), RecyclableObjectDisplay, pResolvedObject);

return TRUE;

}

ulong CatchScopeWalker::GetChildrenCount()

{

return debuggerScope->scopeProperties->Count();

}

void CatchScopeWalker::FetchValueAndAddress(DebuggerScopeProperty &scopeProperty, \_Out\_opt\_ Var \*pValue, \_Out\_opt\_ IDiagObjectAddress \*\* ppAddress)

{

Assert(pValue != nullptr || ppAddress != nullptr);

ArenaAllocator\* arena = pFrame->GetArena();

Var outValue;

IDiagObjectAddress \* pAddress = nullptr;

ScriptContext\* scriptContext = pFrame->GetScriptContext();

if (debuggerScope->scopeType == Js::DiagCatchScopeInObject)

{

Var obj = pFrame->GetInnerScopeFromRegSlot(debuggerScope->GetLocation());

Assert(RecyclableObject::Is(obj));

outValue = RecyclableObjectWalker::GetObject(RecyclableObject::FromVar(obj), RecyclableObject::FromVar(obj), scopeProperty.propId, scriptContext);

bool isInDeadZone = scriptContext->IsUndeclBlockVar(outValue);

if (isInDeadZone)

{

outValue = scriptContext->GetLibrary()->GetDebuggerDeadZoneBlockVariableString();

}

pAddress = Anew(arena, RecyclableObjectAddress, obj, scopeProperty.propId, outValue, isInDeadZone);

}

else

{

outValue = pFrame->GetRegValue(scopeProperty.location);

bool isInDeadZone = scriptContext->IsUndeclBlockVar(outValue);

if (isInDeadZone)

{

outValue = scriptContext->GetLibrary()->GetDebuggerDeadZoneBlockVariableString();

}

pAddress = Anew(arena, LocalObjectAddressForRegSlot, pFrame, scopeProperty.location, outValue);

}

if (pValue)

{

\*pValue = outValue;

}

if (ppAddress)

{

\*ppAddress = pAddress;

}

}

IDiagObjectAddress \*CatchScopeWalker::FindPropertyAddress(PropertyId \_propId, bool& isConst)

{

isConst = false;

IDiagObjectAddress \* address = nullptr;

auto properties = debuggerScope->scopeProperties;

for (int i = 0; i < properties->Count(); i++)

{

if (properties->Item(i).propId == \_propId)

{

FetchValueAndAddress(properties->Item(i), nullptr, &address);

break;

}

}

return address;

}

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

// RecyclableObjectAddress

RecyclableObjectAddress::RecyclableObjectAddress(Var \_parentObj, Js::PropertyId \_propId, Js::Var \_value, BOOL \_isInDeadZone)

: parentObj(\_parentObj),

propId(\_propId),

value(\_value),

isInDeadZone(\_isInDeadZone)

{

parentObj = ((RecyclableObject\*)parentObj)->GetThisObjectOrUnWrap();

}

BOOL RecyclableObjectAddress::IsInDeadZone() const

{

return isInDeadZone;

}

BOOL RecyclableObjectAddress::Set(Var updateObject)

{

if (Js::RecyclableObject::Is(parentObj))

{

Js::RecyclableObject\* obj = Js::RecyclableObject::FromVar(parentObj);

ScriptContext\* requestContext = obj->GetScriptContext(); //TODO: real requestContext

return Js::JavascriptOperators::SetProperty(obj, obj, propId, updateObject, requestContext);

}

return FALSE;

}

BOOL RecyclableObjectAddress::IsWritable()

{

if (Js::RecyclableObject::Is(parentObj))

{

Js::RecyclableObject\* obj = Js::RecyclableObject::FromVar(parentObj);

return obj->IsWritable(propId);

}

return TRUE;

}

Var RecyclableObjectAddress::GetValue(BOOL fUpdated)

{

if (!fUpdated)

{

return value;

}

if (Js::RecyclableObject::Is(parentObj))

{

Js::RecyclableObject\* obj = Js::RecyclableObject::FromVar(parentObj);

ScriptContext\* requestContext = obj->GetScriptContext();

Var objValue = nullptr;

if (Js::JavascriptOperators::GetProperty(obj, propId, &objValue, requestContext))

{

return objValue;

}

}

return nullptr;

}

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

// RecyclableObjectDisplay

RecyclableObjectDisplay::RecyclableObjectDisplay(ResolvedObject\* resolvedObject, DBGPROP\_ATTRIB\_FLAGS defaultAttributes)

: scriptContext(resolvedObject->scriptContext),

instance(resolvedObject->obj),

originalInstance(resolvedObject->originalObj != nullptr ? resolvedObject->originalObj : resolvedObject->obj), // If we don't have it set it means originalInstance should point to object itself

name(resolvedObject->name),

pObjAddress(resolvedObject->address),

defaultAttributes(defaultAttributes),

propertyId(resolvedObject->propId)

{

}

bool RecyclableObjectDisplay::IsLiteralProperty() const

{

Assert(this->scriptContext);

if (this->propertyId != Constants::NoProperty)

{

Js::PropertyRecord const \* propertyRecord = this->scriptContext->GetThreadContext()->GetPropertyName(this->propertyId);

const WCHAR\* startOfPropertyName = propertyRecord->GetBuffer();

const WCHAR\* endOfIdentifier = this->scriptContext->GetCharClassifier()->SkipIdentifier((LPCOLESTR)propertyRecord->GetBuffer());

return (charcount\_t)(endOfIdentifier - startOfPropertyName) == propertyRecord->GetLength();

}

else

{

return true;

}

}

bool RecyclableObjectDisplay::IsSymbolProperty()

{

Assert(this->scriptContext);

if (this->propertyId != Constants::NoProperty)

{

Js::PropertyRecord const \* propertyRecord = this->scriptContext->GetThreadContext()->GetPropertyName(this->propertyId);

return propertyRecord->IsSymbol();

}

return false;

}

LPCWSTR RecyclableObjectDisplay::Name()

{

return name;

}

LPCWSTR RecyclableObjectDisplay::Type()

{

LPCWSTR typeStr;

if(Js::TaggedInt::Is(instance) || Js::JavascriptNumber::Is(instance))

{

typeStr = L"Number";

}

else

{

Js::RecyclableObject\* obj = Js::RecyclableObject::FromVar(instance);

StringBuilder<ArenaAllocator>\* builder = scriptContext->GetThreadContext()->GetDebugManager()->pCurrentInterpreterLocation->stringBuilder;

builder->Reset();

// For the RecyclableObject try to find out the constructor, which will be shown as type for the object.

// This case is to handle the user defined function, built in objects have dedicated classes to handle.

Var value = nullptr;

TypeId typeId = obj->GetTypeId();

if (typeId == TypeIds\_Object && GetPropertyWithScriptEnter(obj, obj, PropertyIds::constructor, &value, scriptContext))

{

builder->AppendCppLiteral(L"Object");

if (Js::JavascriptFunction::Is(value))

{

Js::JavascriptFunction \*pfunction = Js::JavascriptFunction::FromVar(value);

// For an odd chance that the constructor wasn't called to create the object.

Js::ParseableFunctionInfo \*pFuncBody = pfunction->GetFunctionProxy() != nullptr ? pfunction->GetFunctionProxy()->EnsureDeserialized() : nullptr;

if (pFuncBody)

{

const wchar\_t\* pDisplayName = pFuncBody->GetDisplayName();

if (pDisplayName)

{

builder->AppendCppLiteral(L", (");

builder->AppendSz(pDisplayName);

builder->Append(L')');

}

}

}

typeStr = builder->Detach();

}

else if (obj->GetDiagTypeString(builder, scriptContext))

{

typeStr = builder->Detach();

}

else

{

typeStr = L"Undefined";

}

}

return typeStr;

}

Var RecyclableObjectDisplay::GetVarValue(BOOL fUpdated)

{

if (pObjAddress)

{

return pObjAddress->GetValue(fUpdated);

}

return instance;

}

LPCWSTR RecyclableObjectDisplay::Value(int radix)

{

LPCWSTR valueStr = L"";

if(Js::TaggedInt::Is(instance)

|| Js::JavascriptNumber::Is(instance)

|| Js::JavascriptNumberObject::Is(instance)

|| Js::JavascriptOperators::GetTypeId(instance) == TypeIds\_Int64Number

|| Js::JavascriptOperators::GetTypeId(instance) == TypeIds\_UInt64Number)

{

double value;

if (Js::TaggedInt::Is(instance))

{

value = TaggedInt::ToDouble(instance);

}

else if (Js::JavascriptNumber::Is(instance))

{

value = Js::JavascriptNumber::GetValue(instance);

}

else if (Js::JavascriptOperators::GetTypeId(instance) == TypeIds\_Int64Number)

{

value = (double)JavascriptInt64Number::FromVar(instance)->GetValue();

}

else if (Js::JavascriptOperators::GetTypeId(instance) == TypeIds\_UInt64Number)

{

value = (double)JavascriptUInt64Number::FromVar(instance)->GetValue();

}

else

{

Js::JavascriptNumberObject\* numobj = Js::JavascriptNumberObject::FromVar(instance);

value = numobj->GetValue();

}

// For fractional values, radix is ignored.

long l = (long)value;

bool isZero = JavascriptNumber::IsZero(value - (double)l);

if (radix == 10 || !isZero)

{

if (Js::JavascriptNumber::IsNegZero(value))

{

// In debugger, we wanted to show negative zero explicitly

valueStr = L"-0";

}

else

{

valueStr = Js::JavascriptNumber::ToStringRadix10(value, scriptContext)->GetSz();

}

}

else if (radix >= 2 && radix <= 36)

{

if (radix == 16)

{

if (value < 0)

{

// On the tools side we show unsigned value.

ulong ul = (ulong)(long)value; // ARM: casting negative value to ulong gives 0

value = (double)ul;

}

valueStr = Js::JavascriptString::Concat(scriptContext->GetLibrary()->CreateStringFromCppLiteral(L"0x"),

Js::JavascriptNumber::ToStringRadixHelper(value, radix, scriptContext))->GetSz();

}

else

{

valueStr = Js::JavascriptNumber::ToStringRadixHelper(value, radix, scriptContext)->GetSz();

}

}

}

else

{

Js::RecyclableObject\* obj = Js::RecyclableObject::FromVar(instance);

StringBuilder<ArenaAllocator>\* builder = scriptContext->GetThreadContext()->GetDebugManager()->pCurrentInterpreterLocation->stringBuilder;

builder->Reset();

if (obj->GetDiagValueString(builder, scriptContext))

{

valueStr = builder->Detach();

}

else

{

valueStr = L"undefined";

}

}

return valueStr;

}

BOOL RecyclableObjectDisplay::HasChildren()

{

if (Js::RecyclableObject::Is(instance))

{

Js::RecyclableObject\* object = Js::RecyclableObject::FromVar(instance);

if (JavascriptOperators::IsObject(object))

{

if (JavascriptOperators::GetTypeId(object) == TypeIds\_HostDispatch)

{

return TRUE;

}

try

{

BEGIN\_JS\_RUNTIME\_CALL\_EX(scriptContext, false)

{

IGNORE\_STACKWALK\_EXCEPTION(scriptContext);

if (object->CanHaveInterceptors())

{

Js::ForInObjectEnumerator enumerator(object, object->GetScriptContext(), /\* enumSymbols \*/ true);

if (enumerator.MoveNext())

{

enumerator.Clear();

return TRUE;

}

}

else if (object->GetPropertyCount() > 0 || (JavascriptOperators::GetTypeId(object->GetPrototype()) != TypeIds\_Null))

{

return TRUE;

}

}

END\_JS\_RUNTIME\_CALL(scriptContext);

}

catch (Js::JavascriptExceptionObject\* exception)

{

// The For in enumerator can throw an exception and we will use the error object as a child in that case.

Var error = exception->GetThrownObject(scriptContext);

if (error != nullptr && Js::JavascriptError::Is(error))

{

return TRUE;

}

return FALSE;

}

}

}

return FALSE;

}

BOOL RecyclableObjectDisplay::Set(Var updateObject)

{

if (pObjAddress)

{

return pObjAddress->Set(updateObject);

}

return FALSE;

}

DBGPROP\_ATTRIB\_FLAGS RecyclableObjectDisplay::GetTypeAttribute()

{

DBGPROP\_ATTRIB\_FLAGS flag = defaultAttributes;

if (Js::RecyclableObject::Is(instance))

{

if (instance == scriptContext->GetLibrary()->GetDebuggerDeadZoneBlockVariableString())

{

flag |= DBGPROP\_ATTRIB\_VALUE\_IS\_INVALID;

}

else if (JavascriptOperators::GetTypeId(instance) == TypeIds\_Function)

{

flag |= DBGPROP\_ATTRIB\_VALUE\_IS\_METHOD;

}

else if (JavascriptOperators::GetTypeId(instance) == TypeIds\_String

|| JavascriptOperators::GetTypeId(instance) == TypeIds\_StringObject)

{

flag |= DBGPROP\_ATTRIB\_VALUE\_IS\_RAW\_STRING;

}

}

auto checkWriteableFunction = [&]()

{

if (pObjAddress && !pObjAddress->IsWritable())

{

flag |= DBGPROP\_ATTRIB\_VALUE\_READONLY;

}

};

if (!scriptContext->GetThreadContext()->IsScriptActive())

{

BEGIN\_JS\_RUNTIME\_CALL\_EX(scriptContext, false);

{

IGNORE\_STACKWALK\_EXCEPTION(scriptContext);

checkWriteableFunction();

}

END\_JS\_RUNTIME\_CALL(scriptContext);

}

else

{

checkWriteableFunction();

}

// TODO : need to identify Events explicitly for fastDOM

return flag;

}

/\* static \*/

BOOL RecyclableObjectDisplay::GetPropertyWithScriptEnter(RecyclableObject\* originalInstance, RecyclableObject\* instance, PropertyId propertyId, Var\* value, ScriptContext\* scriptContext)

{

BOOL retValue = FALSE;

if(!scriptContext->GetThreadContext()->IsScriptActive())

{

BEGIN\_JS\_RUNTIME\_CALL\_EX(scriptContext, false)

{

IGNORE\_STACKWALK\_EXCEPTION(scriptContext);

retValue = Js::JavascriptOperators::GetProperty(originalInstance, instance, propertyId, value, scriptContext);

}

END\_JS\_RUNTIME\_CALL(scriptContext);

}

else

{

retValue = Js::JavascriptOperators::GetProperty(originalInstance, instance, propertyId, value, scriptContext);

}

return retValue;

}

WeakArenaReference<IDiagObjectModelWalkerBase>\* RecyclableObjectDisplay::CreateWalker()

{

return CreateAWalker<RecyclableObjectWalker>(scriptContext, instance, originalInstance);

}

StringBuilder<ArenaAllocator>\* RecyclableObjectDisplay::GetStringBuilder()

{

return scriptContext->GetThreadContext()->GetDebugManager()->pCurrentInterpreterLocation->stringBuilder;

}

PropertyId RecyclableObjectDisplay::GetPropertyId() const

{

return this->propertyId;

}

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

// RecyclableObjectWalker

RecyclableObjectWalker::RecyclableObjectWalker(ScriptContext\* \_scriptContext, Var \_slot)

: scriptContext(\_scriptContext),

instance(\_slot),

originalInstance(\_slot),

pMembersList(nullptr),

innerArrayObjectWalker(nullptr),

fakeGroupObjectWalkerList(nullptr)

{

}

RecyclableObjectWalker::RecyclableObjectWalker(ScriptContext\* \_scriptContext, Var \_slot, Var \_originalInstance)

: scriptContext(\_scriptContext),

instance(\_slot),

originalInstance(\_originalInstance),

pMembersList(nullptr),

innerArrayObjectWalker(nullptr),

fakeGroupObjectWalkerList(nullptr)

{

}

BOOL RecyclableObjectWalker::Get(int index, ResolvedObject\* pResolvedObject)

{

AssertMsg(pResolvedObject, "Bad usage of RecyclableObjectWalker::Get");

int fakeObjCount = fakeGroupObjectWalkerList ? fakeGroupObjectWalkerList->Count() : 0;

int nonArrayElementCount = Js::RecyclableObject::Is(instance) ? pMembersList->Count() : 0;

int arrayItemCount = innerArrayObjectWalker ? innerArrayObjectWalker->GetChildrenCount() : 0;

if (index < 0 || !pMembersList || index >= (pMembersList->Count() + arrayItemCount + fakeObjCount))

{

return FALSE;

}

// First the virtual groups

if (index < fakeObjCount)

{

Assert(fakeGroupObjectWalkerList);

return fakeGroupObjectWalkerList->Item(index)->GetGroupObject(pResolvedObject);

}

index -= fakeObjCount;

if (index < nonArrayElementCount)

{

Assert(Js::RecyclableObject::Is(instance));

pResolvedObject->propId = pMembersList->Item(index)->propId;

if (pResolvedObject->propId == Js::Constants::NoProperty || Js::IsInternalPropertyId(pResolvedObject->propId))

{

Assert(FALSE);

return FALSE;

}

Js::DebuggerPropertyDisplayInfo\* displayInfo = pMembersList->Item(index);

const Js::PropertyRecord\* propertyRecord = scriptContext->GetPropertyName(pResolvedObject->propId);

pResolvedObject->name = propertyRecord->GetBuffer();

pResolvedObject->obj = displayInfo->aVar;

Assert(pResolvedObject->obj);

pResolvedObject->scriptContext = scriptContext;

pResolvedObject->typeId = JavascriptOperators::GetTypeId(pResolvedObject->obj);

pResolvedObject->address = Anew(GetArenaFromContext(scriptContext),

RecyclableObjectAddress,

instance,

pResolvedObject->propId,

pResolvedObject->obj,

displayInfo->IsInDeadZone() ? TRUE : FALSE);

pResolvedObject->isConst = displayInfo->IsConst();

return TRUE;

}

index -= nonArrayElementCount;

if (index < arrayItemCount)

{

Assert(innerArrayObjectWalker);

return innerArrayObjectWalker->Get(index, pResolvedObject);

}

Assert(false);

return FALSE;

}

void RecyclableObjectWalker::EnsureFakeGroupObjectWalkerList()

{

if (fakeGroupObjectWalkerList == nullptr)

{

ArenaAllocator \*arena = GetArenaFromContext(scriptContext);

fakeGroupObjectWalkerList = JsUtil::List<IDiagObjectModelWalkerBase \*, ArenaAllocator>::New(arena);

}

}

IDiagObjectAddress \*RecyclableObjectWalker::FindPropertyAddress(PropertyId propertyId, bool& isConst)

{

GetChildrenCount(); // Ensure to populate members

if (pMembersList != nullptr)

{

for (int i = 0; i < pMembersList->Count(); i++)

{

DebuggerPropertyDisplayInfo \*pair = pMembersList->Item(i);

Assert(pair);

if (pair->propId == propertyId)

{

isConst = pair->IsConst();

return Anew(GetArenaFromContext(scriptContext),

RecyclableObjectAddress,

instance,

propertyId,

pair->aVar,

pair->IsInDeadZone() ? TRUE : FALSE);

}

}

}

// Following is for "with object" scope lookup. We may have members in [Methods] group or prototype chain that need to

// be exposed to expression evaluation.

if (fakeGroupObjectWalkerList != nullptr)

{

// WARNING: Following depends on [Methods] group being before [prototype] group. We need to check local [Methods] group

// first for local properties before going to prototype chain.

for (int i = 0; i < fakeGroupObjectWalkerList->Count(); i++)

{

IDiagObjectAddress\* address = fakeGroupObjectWalkerList->Item(i)->FindPropertyAddress(propertyId, isConst);

if (address != nullptr)

{

return address;

}

}

}

return nullptr;

}

ulong RecyclableObjectWalker::GetChildrenCount()

{

if (pMembersList == nullptr)

{

ArenaAllocator \*arena = GetArenaFromContext(scriptContext);

pMembersList = JsUtil::List<DebuggerPropertyDisplayInfo \*, ArenaAllocator>::New(arena);

RecyclableMethodsGroupWalker \*pMethodsGroupWalker = nullptr;

if (Js::RecyclableObject::Is(instance))

{

Js::RecyclableObject\* object = Js::RecyclableObject::FromVar(instance);

// If we are walking a prototype, we'll use its instance for property names enumeration, but originalInstance to get values

Js::RecyclableObject\* originalObject = (originalInstance != nullptr) ? Js::RecyclableObject::FromVar(originalInstance) : object;

const Js::TypeId typeId = JavascriptOperators::GetTypeId(instance);

if (JavascriptOperators::IsObject(object))

{

if (object->CanHaveInterceptors() || JavascriptOperators::GetTypeId(object) == TypeIds\_Proxy)

{

try

{

JavascriptEnumerator\* enumerator;

if (object->GetEnumerator(true/\*enumNonEnumable\*/, (Var\*)&enumerator, scriptContext, false/\*preferSnapshotSyntax\*/, true/\*enumSymbols\*/))

{

Js::PropertyId propertyId;

Var obj;

while ((obj = enumerator->GetCurrentAndMoveNext(propertyId)) != nullptr)

{

if (!JavascriptString::Is(obj))

{

continue;

}

if (propertyId == Constants::NoProperty)

{

JavascriptString \*pString = JavascriptString::FromVar(obj);

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

{

// If we have a property string, it is assumed that the propertyId is being

// kept alive with the object

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

propertyId = propertyString->GetPropertyRecord()->GetPropertyId();

}

else

{

const PropertyRecord\* propertyRecord;

scriptContext->GetOrAddPropertyRecord(pString->GetSz(), pString->GetLength(), &propertyRecord);

propertyId = propertyRecord->GetPropertyId();

}

}

// GetCurrentAndMoveNext shouldn't return an internal property id

Assert(!Js::IsInternalPropertyId(propertyId));

uint32 indexVal;

Var varValue;

if (scriptContext->IsNumericPropertyId(propertyId, &indexVal) && object->GetItem(object, indexVal, &varValue, scriptContext))

{

InsertItem(propertyId, false /\*isConst\*/, false /\*isUnscoped\*/, varValue, &pMethodsGroupWalker, true /\*shouldPinProperty\*/);

}

else

{

InsertItem(originalObject, object, propertyId, false /\*isConst\*/, false /\*isUnscoped\*/, &pMethodsGroupWalker, true /\*shouldPinProperty\*/);

}

}

}

}

catch (JavascriptExceptionObject\* exception)

{

Var error = exception->GetThrownObject(scriptContext);

if (error != nullptr && Js::JavascriptError::Is(error))

{

Js::PropertyId propertyId = scriptContext->GetOrAddPropertyIdTracked(L"{error}");

InsertItem(propertyId, false /\*isConst\*/, false /\*isUnscoped\*/, error, &pMethodsGroupWalker);

}

}

if (typeId == TypeIds\_Proxy)

{

// Provide [Proxy] group object

EnsureFakeGroupObjectWalkerList();

JavascriptProxy\* proxy = JavascriptProxy::FromVar(object);

RecyclableProxyObjectWalker\* proxyWalker = Anew(arena, RecyclableProxyObjectWalker, scriptContext, proxy);

fakeGroupObjectWalkerList->Add(proxyWalker);

}

// If current object has internal proto object then provide [prototype] group object.

if (JavascriptOperators::GetTypeId(object->GetPrototype()) != TypeIds\_Null)

{

// Has [prototype] object.

EnsureFakeGroupObjectWalkerList();

RecyclableProtoObjectWalker \*pProtoWalker = Anew(arena, RecyclableProtoObjectWalker, scriptContext, instance, (originalInstance == nullptr) ? instance : originalInstance);

fakeGroupObjectWalkerList->Add(pProtoWalker);

}

}

else

{

RecyclableObject\* wrapperObject = nullptr;

if (JavascriptOperators::GetTypeId(object) == TypeIds\_WithScopeObject)

{

wrapperObject = object;

object = object->GetThisObjectOrUnWrap();

}

int count = object->GetPropertyCount();

for (int i = 0; i < count; i++)

{

Js::PropertyId propertyId = object->GetPropertyId((PropertyIndex)i);

bool isUnscoped = false;

if (wrapperObject && JavascriptOperators::IsPropertyUnscopable(object, propertyId))

{

isUnscoped = true;

}

if (propertyId != Js::Constants::NoProperty && !Js::IsInternalPropertyId(propertyId))

{

InsertItem(originalObject, object, propertyId, false /\*isConst\*/, isUnscoped, &pMethodsGroupWalker);

}

}

if (CONFIG\_FLAG(EnumerateSpecialPropertiesInDebugger))

{

count = object->GetSpecialPropertyCount();

PropertyId const \* specialPropertyIds = object->GetSpecialPropertyIds();

for (int i = 0; i < count; i++)

{

Js::PropertyId propertyId = specialPropertyIds[i];

bool isUnscoped = false;

if (wrapperObject && JavascriptOperators::IsPropertyUnscopable(object, propertyId))

{

isUnscoped = true;

}

if (propertyId != Js::Constants::NoProperty)

{

bool isConst = true;

if (propertyId == PropertyIds::length && Js::JavascriptArray::Is(object))

{

// For JavascriptArrays, we allow resetting the length special property.

isConst = false;

}

auto containsPredicate = [&](Js::DebuggerPropertyDisplayInfo\* info) { return info->propId == propertyId; };

if (Js::BoundFunction::Is(object)

&& this->pMembersList->Any(containsPredicate))

{

// Bound functions can already contain their special properties,

// so we need to check for that (caller and arguments). This occurs

// when JavascriptFunction::EntryBind() is called. Arguments can similarly

// already display caller in compat mode 8.

continue;

}

AssertMsg(!this->pMembersList->Any(containsPredicate), "Special property already on the object, no need to insert.");

InsertItem(originalObject, object, propertyId, isConst, isUnscoped, &pMethodsGroupWalker);

}

}

if (Js::JavascriptFunction::Is(object))

{

// We need to special-case RegExp constructor here because it has some special properties (above) and some

// special enumerable properties which should all show up in the debugger.

JavascriptRegExpConstructor\* regExp = scriptContext->GetLibrary()->GetRegExpConstructor();

if (regExp == object)

{

bool isUnscoped = false;

bool isConst = true;

count = regExp->GetSpecialEnumerablePropertyCount();

PropertyId const \* specialPropertyIds = regExp->GetSpecialEnumerablePropertyIds();

for (int i = 0; i < count; i++)

{

Js::PropertyId propertyId = specialPropertyIds[i];

InsertItem(originalObject, object, propertyId, isConst, isUnscoped, &pMethodsGroupWalker);

}

}

else if (Js::JavascriptFunction::FromVar(object)->IsScriptFunction() || Js::JavascriptFunction::FromVar(object)->IsBoundFunction())

{

// Adding special property length for the ScriptFunction, like it is done in JavascriptFunction::GetSpecialNonEnumerablePropertyName

InsertItem(originalObject, object, PropertyIds::length, true/\*not editable\*/, false /\*isUnscoped\*/, &pMethodsGroupWalker);

}

}

}

// If current object has internal proto object then provide [prototype] group object.

if (JavascriptOperators::GetTypeId(object->GetPrototype()) != TypeIds\_Null)

{

// Has [prototype] object.

EnsureFakeGroupObjectWalkerList();

RecyclableProtoObjectWalker \*pProtoWalker = Anew(arena, RecyclableProtoObjectWalker, scriptContext, instance, originalInstance);

fakeGroupObjectWalkerList->Add(pProtoWalker);

}

}

// If the object contains array indices.

if (typeId == TypeIds\_Arguments)

{

// Create ArgumentsArray walker for a arguments object

Js::ArgumentsObject \* argObj = static\_cast<Js::ArgumentsObject\*>(instance);

Assert(argObj);

if (argObj->GetNumberOfArguments() > 0 || argObj->HasNonEmptyObjectArray())

{

innerArrayObjectWalker = Anew(arena, RecyclableArgumentsArrayWalker, scriptContext, (Var)instance, originalInstance);

}

}

else if (typeId == TypeIds\_Map)

{

// Provide [Map] group object.

EnsureFakeGroupObjectWalkerList();

JavascriptMap\* map = JavascriptMap::FromVar(object);

RecyclableMapObjectWalker \*pMapWalker = Anew(arena, RecyclableMapObjectWalker, scriptContext, map);

fakeGroupObjectWalkerList->Add(pMapWalker);

}

else if (typeId == TypeIds\_Set)

{

// Provide [Set] group object.

EnsureFakeGroupObjectWalkerList();

JavascriptSet\* set = JavascriptSet::FromVar(object);

RecyclableSetObjectWalker \*pSetWalker = Anew(arena, RecyclableSetObjectWalker, scriptContext, set);

fakeGroupObjectWalkerList->Add(pSetWalker);

}

else if (typeId == TypeIds\_WeakMap)

{

// Provide [WeakMap] group object.

EnsureFakeGroupObjectWalkerList();

JavascriptWeakMap\* weakMap = JavascriptWeakMap::FromVar(object);

RecyclableWeakMapObjectWalker \*pWeakMapWalker = Anew(arena, RecyclableWeakMapObjectWalker, scriptContext, weakMap);

fakeGroupObjectWalkerList->Add(pWeakMapWalker);

}

else if (typeId == TypeIds\_WeakSet)

{

// Provide [WeakSet] group object.

EnsureFakeGroupObjectWalkerList();

JavascriptWeakSet\* weakSet = JavascriptWeakSet::FromVar(object);

RecyclableWeakSetObjectWalker \*pWeakSetWalker = Anew(arena, RecyclableWeakSetObjectWalker, scriptContext, weakSet);

fakeGroupObjectWalkerList->Add(pWeakSetWalker);

}

else if (Js::DynamicType::Is(typeId))

{

DynamicObject \*const dynamicObject = Js::DynamicObject::FromVar(instance);

if (dynamicObject->HasNonEmptyObjectArray())

{

ArrayObject\* objectArray = dynamicObject->GetObjectArray();

if (Js::ES5Array::Is(objectArray))

{

innerArrayObjectWalker = Anew(arena, RecyclableES5ArrayWalker, scriptContext, objectArray, originalInstance);

}

else if (Js::JavascriptArray::Is(objectArray))

{

innerArrayObjectWalker = Anew(arena, RecyclableArrayWalker, scriptContext, objectArray, originalInstance);

}

else

{

innerArrayObjectWalker = Anew(arena, RecyclableTypedArrayWalker, scriptContext, objectArray, originalInstance);

}

innerArrayObjectWalker->SetOnlyWalkOwnProperties(true);

}

}

}

}

// Sort the members of the methods group

if (pMethodsGroupWalker)

{

pMethodsGroupWalker->Sort();

}

// Sort current pMembersList.

pMembersList->Sort(ElementsComparer, scriptContext);

}

ulong childrenCount =

pMembersList->Count()

+ (innerArrayObjectWalker ? innerArrayObjectWalker->GetChildrenCount() : 0)

+ (fakeGroupObjectWalkerList ? fakeGroupObjectWalkerList->Count() : 0);

return childrenCount;

}

void RecyclableObjectWalker::InsertItem(

Js::RecyclableObject \*pOriginalObject,

Js::RecyclableObject \*pObject,

PropertyId propertyId,

bool isReadOnly,

bool isUnscoped,

Js::RecyclableMethodsGroupWalker \*\*ppMethodsGroupWalker,

bool shouldPinProperty /\* = false\*/)

{

Assert(pOriginalObject);

Assert(pObject);

Assert(propertyId);

Assert(ppMethodsGroupWalker);

if (propertyId != PropertyIds::\_\_proto\_\_)

{

InsertItem(propertyId, isReadOnly, isUnscoped, RecyclableObjectWalker::GetObject(pOriginalObject, pObject, propertyId, scriptContext), ppMethodsGroupWalker, shouldPinProperty);

}

else // Since \_\_proto\_\_ defined as a Getter we should always evaluate it against object itself instead of walking prototype chain

{

InsertItem(propertyId, isReadOnly, isUnscoped, RecyclableObjectWalker::GetObject(pObject, pObject, propertyId, scriptContext), ppMethodsGroupWalker, shouldPinProperty);

}

}

void RecyclableObjectWalker::InsertItem(

PropertyId propertyId,

bool isConst,

bool isUnscoped,

Var itemObj,

Js:: RecyclableMethodsGroupWalker \*\*ppMethodsGroupWalker,

bool shouldPinProperty /\* = false\*/)

{

Assert(propertyId);

Assert(ppMethodsGroupWalker);

if (itemObj == nullptr)

{

itemObj = scriptContext->GetLibrary()->GetUndefined();

}

if (shouldPinProperty)

{

const Js::PropertyRecord \* propertyRecord = scriptContext->GetPropertyName(propertyId);

if (propertyRecord)

{

// Pin this record so that it will not go away till we are done with this break.

scriptContext->GetDebugContext()->GetProbeContainer()->PinPropertyRecord(propertyRecord);

}

}

ArenaAllocator \*arena = GetArenaFromContext(scriptContext);

if (JavascriptOperators::GetTypeId(itemObj) == TypeIds\_Function)

{

EnsureFakeGroupObjectWalkerList();

if (\*ppMethodsGroupWalker == nullptr)

{

\*ppMethodsGroupWalker = Anew(arena, RecyclableMethodsGroupWalker, scriptContext, instance);

fakeGroupObjectWalkerList->Add(\*ppMethodsGroupWalker);

}

(\*ppMethodsGroupWalker)->AddItem(propertyId, itemObj);

}

else

{

DWORD flags = DebuggerPropertyDisplayInfoFlags\_None;

flags |= isConst ? DebuggerPropertyDisplayInfoFlags\_Const : 0;

flags |= isUnscoped ? DebuggerPropertyDisplayInfoFlags\_Unscope : 0;

DebuggerPropertyDisplayInfo \*info = Anew(arena, DebuggerPropertyDisplayInfo, propertyId, itemObj, flags);

pMembersList->Add(info);

}

}

/\*static\*/

Var RecyclableObjectWalker::GetObject(RecyclableObject\* originalInstance, RecyclableObject\* instance, PropertyId propertyId, ScriptContext\* scriptContext)

{

Assert(instance);

Assert(!Js::IsInternalPropertyId(propertyId));

Var obj = nullptr;

try

{

if (!RecyclableObjectDisplay::GetPropertyWithScriptEnter(originalInstance, instance, propertyId, &obj, scriptContext))

{

return instance->GetScriptContext()->GetMissingPropertyResult(instance, propertyId);

}

}

catch(Js::JavascriptExceptionObject \* exceptionObject)

{

Var error = exceptionObject->GetThrownObject(instance->GetScriptContext());

if (error != nullptr && Js::JavascriptError::Is(error))

{

obj = error;

}

}

return obj;

}

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

// RecyclableArrayAddress

RecyclableArrayAddress::RecyclableArrayAddress(Var \_parentArray, unsigned int \_index)

: parentArray(\_parentArray),

index(\_index)

{

}

BOOL RecyclableArrayAddress::Set(Var updateObject)

{

if (Js::JavascriptArray::Is(parentArray))

{

Js::JavascriptArray\* jsArray = Js::JavascriptArray::FromVar(parentArray);

return jsArray->SetItem(index, updateObject, PropertyOperation\_None);

}

return FALSE;

}

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

// RecyclableArrayDisplay

RecyclableArrayDisplay::RecyclableArrayDisplay(ResolvedObject\* resolvedObject)

: RecyclableObjectDisplay(resolvedObject)

{

}

BOOL RecyclableArrayDisplay::HasChildrenInternal(Js::JavascriptArray\* arrayObj)

{

Assert(arrayObj);

if (JavascriptOperators::GetTypeId(arrayObj->GetPrototype()) != TypeIds\_Null)

{

return TRUE;

}

uint32 index = arrayObj->GetNextIndex(Js::JavascriptArray::InvalidIndex);

return index != Js::JavascriptArray::InvalidIndex && index < arrayObj->GetLength();

}

BOOL RecyclableArrayDisplay::HasChildren()

{

if (Js::JavascriptArray::Is(instance))

{

Js::JavascriptArray\* arrayObj = Js::JavascriptArray::FromVar(instance);

if (HasChildrenInternal(arrayObj))

{

return TRUE;

}

}

return RecyclableObjectDisplay::HasChildren();

}

WeakArenaReference<IDiagObjectModelWalkerBase>\* RecyclableArrayDisplay::CreateWalker()

{

return CreateAWalker<RecyclableArrayWalker>(scriptContext, instance, originalInstance);

}

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

// RecyclableArrayWalker

uint32 RecyclableArrayWalker::GetItemCount(Js::JavascriptArray\* arrayObj)

{

if (pAbsoluteIndexList == nullptr)

{

Assert(arrayObj);

pAbsoluteIndexList = JsUtil::List<uint32, ArenaAllocator>::New(GetArenaFromContext(scriptContext));

Assert(pAbsoluteIndexList);

uint32 dataIndex = Js::JavascriptArray::InvalidIndex;

uint32 descriptorIndex = Js::JavascriptArray::InvalidIndex;

uint32 absIndex = Js::JavascriptArray::InvalidIndex;

do

{

if (absIndex == dataIndex)

{

dataIndex = arrayObj->GetNextIndex(dataIndex);

}

if (absIndex == descriptorIndex)

{

descriptorIndex = GetNextDescriptor(descriptorIndex);

}

absIndex = min(dataIndex, descriptorIndex);

if (absIndex == Js::JavascriptArray::InvalidIndex || absIndex >= arrayObj->GetLength())

{

break;

}

pAbsoluteIndexList->Add(absIndex);

} while (absIndex < arrayObj->GetLength());

}

return (uint32)pAbsoluteIndexList->Count();

}

BOOL RecyclableArrayWalker::FetchItemAtIndex(Js::JavascriptArray\* arrayObj, uint32 index, Var \* value)

{

Assert(arrayObj);

Assert(value);

return arrayObj->DirectGetItemAt(index, value);

}

Var RecyclableArrayWalker::FetchItemAt(Js::JavascriptArray\* arrayObj, uint32 index)

{

Assert(arrayObj);

return arrayObj->DirectGetItem(index);

}

LPCWSTR RecyclableArrayWalker::GetIndexName(uint32 index, StringBuilder<ArenaAllocator>\* stringBuilder)

{

stringBuilder->Append(L'[');

if (stringBuilder->AppendUint64(index) != 0)

{

return L"[.]";

}

stringBuilder->Append(L']');

return stringBuilder->Detach();

}

RecyclableArrayWalker::RecyclableArrayWalker(ScriptContext\* scriptContext, Var instance, Var originalInstance)

: indexedItemCount(0),

pAbsoluteIndexList(nullptr),

fOnlyOwnProperties(false),

RecyclableObjectWalker(scriptContext,instance,originalInstance)

{

}

BOOL RecyclableArrayWalker::GetResolvedObject(Js::JavascriptArray\* arrayObj, int index, ResolvedObject\* pResolvedObject, uint32 \* pabsIndex)

{

Assert(arrayObj);

Assert(pResolvedObject);

Assert(pAbsoluteIndexList);

Assert(pAbsoluteIndexList->Count() > index);

// translate i'th Item to the correct array index and return

uint32 absIndex = pAbsoluteIndexList->Item(index);

pResolvedObject->obj = FetchItemAt(arrayObj, absIndex);

pResolvedObject->scriptContext = scriptContext;

pResolvedObject->typeId = JavascriptOperators::GetTypeId(pResolvedObject->obj);

pResolvedObject->address = nullptr;

StringBuilder<ArenaAllocator>\* builder = GetBuilder();

Assert(builder);

builder->Reset();

pResolvedObject->name = GetIndexName(absIndex, builder);

if (pabsIndex)

{

\*pabsIndex = absIndex;

}

return TRUE;

}

BOOL RecyclableArrayWalker::Get(int i, ResolvedObject\* pResolvedObject)

{

AssertMsg(pResolvedObject, "Bad usage of RecyclableArrayWalker::Get");

if (Js::JavascriptArray::Is(instance) || Js::ES5Array::Is(instance))

{

Js::JavascriptArray\* arrayObj = GetArrayObject();

int nonArrayElementCount = (!fOnlyOwnProperties ? RecyclableObjectWalker::GetChildrenCount() : 0);

if (i < nonArrayElementCount)

{

return RecyclableObjectWalker::Get(i, pResolvedObject);

}

else

{

i -= nonArrayElementCount;

uint32 absIndex; // Absolute index

GetResolvedObject(arrayObj, i, pResolvedObject, &absIndex);

pResolvedObject->address = Anew(GetArenaFromContext(scriptContext),

RecyclableArrayAddress,

instance,

absIndex);

return TRUE;

}

}

return FALSE;

}

Js::JavascriptArray\* RecyclableArrayWalker::GetArrayObject()

{

Assert(Js::JavascriptArray::Is(instance) || Js::ES5Array::Is(instance));

return Js::ES5Array::Is(instance) ?

static\_cast<Js::JavascriptArray \*>(RecyclableObject::FromVar(instance)) :

Js::JavascriptArray::FromVar(instance);

}

ulong RecyclableArrayWalker::GetChildrenCount()

{

if (Js::JavascriptArray::Is(instance) || Js::ES5Array::Is(instance))

{

ulong count = (!fOnlyOwnProperties ? RecyclableObjectWalker::GetChildrenCount() : 0);

Js::JavascriptArray\* arrayObj = GetArrayObject();

return GetItemCount(arrayObj) + count;

}

return 0;

}

StringBuilder<ArenaAllocator>\* RecyclableArrayWalker::GetBuilder()

{

return scriptContext->GetThreadContext()->GetDebugManager()->pCurrentInterpreterLocation->stringBuilder;

}

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

// RecyclableArgumentsArrayAddress

RecyclableArgumentsArrayAddress::RecyclableArgumentsArrayAddress(Var \_parentArray, unsigned int \_index)

: parentArray(\_parentArray),

index(\_index)

{

}

BOOL RecyclableArgumentsArrayAddress::Set(Var updateObject)

{

if (Js::ArgumentsObject::Is(parentArray))

{

Js::ArgumentsObject\* argObj = static\_cast<Js::ArgumentsObject\*>(parentArray);

return argObj->SetItem(index, updateObject, PropertyOperation\_None);

}

return FALSE;

}

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

// RecyclableArgumentsObjectDisplay

RecyclableArgumentsObjectDisplay::RecyclableArgumentsObjectDisplay(ResolvedObject\* resolvedObject, LocalsWalker \*localsWalker)

: RecyclableObjectDisplay(resolvedObject), pLocalsWalker(localsWalker)

{

}

BOOL RecyclableArgumentsObjectDisplay::HasChildren()

{

// It must have children otherwise object itself was not created in first place.

return TRUE;

}

WeakArenaReference<IDiagObjectModelWalkerBase>\* RecyclableArgumentsObjectDisplay::CreateWalker()

{

ReferencedArenaAdapter\* pRefArena = scriptContext->GetThreadContext()->GetDebugManager()->GetDiagnosticArena();

if (pRefArena)

{

IDiagObjectModelWalkerBase\* pOMWalker = Anew(pRefArena->Arena(), RecyclableArgumentsObjectWalker, scriptContext, instance, pLocalsWalker);

return HeapNew(WeakArenaReference<IDiagObjectModelWalkerBase>,pRefArena, pOMWalker);

}

return nullptr;

}

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

// RecyclableArgumentsObjectWalker

RecyclableArgumentsObjectWalker::RecyclableArgumentsObjectWalker(ScriptContext\* pContext, Var \_instance, LocalsWalker \* localsWalker)

: RecyclableObjectWalker(pContext, \_instance), pLocalsWalker(localsWalker)

{

}

ulong RecyclableArgumentsObjectWalker::GetChildrenCount()

{

if (innerArrayObjectWalker == nullptr)

{

ulong count = RecyclableObjectWalker::GetChildrenCount();

if (innerArrayObjectWalker != nullptr)

{

RecyclableArgumentsArrayWalker \*pWalker = static\_cast<RecyclableArgumentsArrayWalker \*> (innerArrayObjectWalker);

pWalker->FetchFormalsAddress(pLocalsWalker);

}

return count;

}

return RecyclableObjectWalker::GetChildrenCount();

}

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

// RecyclableArgumentsArrayWalker

RecyclableArgumentsArrayWalker::RecyclableArgumentsArrayWalker(ScriptContext\* \_scriptContext, Var \_instance, Var \_originalInstance)

: RecyclableArrayWalker(\_scriptContext, \_instance, \_originalInstance), pFormalsList(nullptr)

{

}

ulong RecyclableArgumentsArrayWalker::GetChildrenCount()

{

if (pMembersList == nullptr)

{

Assert(Js::ArgumentsObject::Is(instance));

Js::ArgumentsObject \* argObj = static\_cast<Js::ArgumentsObject\*>(instance);

pMembersList = JsUtil::List<DebuggerPropertyDisplayInfo \*, ArenaAllocator>::New(GetArenaFromContext(scriptContext));

Assert(pMembersList);

uint32 totalCount = argObj->GetNumberOfArguments();

Js::ArrayObject \* objectArray = argObj->GetObjectArray();

if (objectArray != nullptr && objectArray->GetLength() > totalCount)

{

totalCount = objectArray->GetLength();

}

for (uint32 index = 0; index < totalCount; index++)

{

Var itemObj;

if (argObj->GetItem(argObj, index, &itemObj, scriptContext))

{

DebuggerPropertyDisplayInfo \*info = Anew(GetArenaFromContext(scriptContext), DebuggerPropertyDisplayInfo, index, itemObj, DebuggerPropertyDisplayInfoFlags\_None);

Assert(info);

pMembersList->Add(info);

}

}

}

return pMembersList ? pMembersList->Count() : 0;

}

void RecyclableArgumentsArrayWalker::FetchFormalsAddress(LocalsWalker \* localsWalker)

{

Assert(localsWalker);

Assert(localsWalker->pFrame);

Js::FunctionBody \*pFBody = localsWalker->pFrame->GetJavascriptFunction()->GetFunctionBody();

Assert(pFBody);

PropertyIdOnRegSlotsContainer \* container = pFBody->GetPropertyIdOnRegSlotsContainer();

if (container && container->propertyIdsForFormalArgs)

{

for (uint32 i = 0; i < container->propertyIdsForFormalArgs->count; i++)

{

if (container->propertyIdsForFormalArgs->elements[i] != Js::Constants::NoRegister)

{

bool isConst = false;

IDiagObjectAddress \* address = localsWalker->FindPropertyAddress(container->propertyIdsForFormalArgs->elements[i], false, isConst);

if (address)

{

if (pFormalsList == nullptr)

{

pFormalsList = JsUtil::List<IDiagObjectAddress \*, ArenaAllocator>::New(GetArenaFromContext(scriptContext));

}

pFormalsList->Add(address);

}

}

}

}

}

BOOL RecyclableArgumentsArrayWalker::Get(int i, ResolvedObject\* pResolvedObject)

{

AssertMsg(pResolvedObject, "Bad usage of RecyclableArgumentsArrayWalker::Get");

Assert(i >= 0);

Assert(Js::ArgumentsObject::Is(instance));

if (pMembersList && i < pMembersList->Count())

{

Assert(pMembersList->Item(i) != nullptr);

pResolvedObject->address = nullptr;

if (pFormalsList && i < pFormalsList->Count())

{

pResolvedObject->address = pFormalsList->Item(i);

pResolvedObject->obj = pResolvedObject->address->GetValue(FALSE);

if (pResolvedObject->obj == nullptr)

{

// Temp workaround till the arguments (In jit code) work is ready.

Assert(Js::Configuration::Global.EnableJitInDebugMode());

pResolvedObject->obj = pMembersList->Item(i)->aVar;

}

else if (pResolvedObject->obj != pMembersList->Item(i)->aVar)

{

// We set the formals value in the object itself, so that expression evaluation can reflect them correctly

Js::HeapArgumentsObject\* argObj = static\_cast<Js::HeapArgumentsObject\*>(instance);

JavascriptOperators::SetItem(instance, argObj, (uint32)pMembersList->Item(i)->propId, pResolvedObject->obj, scriptContext, PropertyOperation\_None);

}

}

else

{

pResolvedObject->obj = pMembersList->Item(i)->aVar;

}

Assert(pResolvedObject->obj);

pResolvedObject->scriptContext = scriptContext;

pResolvedObject->typeId = JavascriptOperators::GetTypeId(pResolvedObject->obj);

StringBuilder<ArenaAllocator>\* builder = GetBuilder();

Assert(builder);

builder->Reset();

pResolvedObject->name = GetIndexName(pMembersList->Item(i)->propId, builder);

if (pResolvedObject->typeId != TypeIds\_HostDispatch && pResolvedObject->address == nullptr)

{

pResolvedObject->address = Anew(GetArenaFromContext(scriptContext),

RecyclableArgumentsArrayAddress,

instance,

pMembersList->Item(i)->propId);

}

return TRUE;

}

return FALSE;

}

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

// RecyclableTypedArrayAddress

RecyclableTypedArrayAddress::RecyclableTypedArrayAddress(Var \_parentArray, unsigned int \_index)

: RecyclableArrayAddress(\_parentArray, \_index)

{

}

BOOL RecyclableTypedArrayAddress::Set(Var updateObject)

{

if (Js::TypedArrayBase::Is(parentArray))

{

Js::TypedArrayBase\* typedArrayObj = Js::TypedArrayBase::FromVar(parentArray);

return typedArrayObj->SetItem(index, updateObject, PropertyOperation\_None);

}

return FALSE;

}

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

// RecyclableTypedArrayDisplay

RecyclableTypedArrayDisplay::RecyclableTypedArrayDisplay(ResolvedObject\* resolvedObject)

: RecyclableObjectDisplay(resolvedObject)

{

}

BOOL RecyclableTypedArrayDisplay::HasChildren()

{

if (Js::TypedArrayBase::Is(instance))

{

Js::TypedArrayBase\* typedArrayObj = Js::TypedArrayBase::FromVar(instance);

if (typedArrayObj->GetLength() > 0)

{

return TRUE;

}

}

return RecyclableObjectDisplay::HasChildren();

}

WeakArenaReference<IDiagObjectModelWalkerBase>\* RecyclableTypedArrayDisplay::CreateWalker()

{

return CreateAWalker<RecyclableTypedArrayWalker>(scriptContext, instance, originalInstance);

}

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

// RecyclableTypedArrayWalker

RecyclableTypedArrayWalker::RecyclableTypedArrayWalker(ScriptContext\* \_scriptContext, Var \_instance, Var \_originalInstance)

: RecyclableArrayWalker(\_scriptContext, \_instance, \_originalInstance)

{

}

ulong RecyclableTypedArrayWalker::GetChildrenCount()

{

if (!indexedItemCount)

{

Assert(Js::TypedArrayBase::Is(instance));

Js::TypedArrayBase \* typedArrayObj = Js::TypedArrayBase::FromVar(instance);

indexedItemCount = typedArrayObj->GetLength() + (!fOnlyOwnProperties ? RecyclableObjectWalker::GetChildrenCount() : 0);

}

return indexedItemCount;

}

BOOL RecyclableTypedArrayWalker::Get(int i, ResolvedObject\* pResolvedObject)

{

AssertMsg(pResolvedObject, "Bad usage of RecyclableTypedArrayWalker::Get");

Assert(Js::TypedArrayBase::Is(instance));

Js::TypedArrayBase \* typedArrayObj = Js::TypedArrayBase::FromVar(instance);

int nonArrayElementCount = (!fOnlyOwnProperties ? RecyclableObjectWalker::GetChildrenCount() : 0);

if (i < nonArrayElementCount)

{

return RecyclableObjectWalker::Get(i, pResolvedObject);

}

else

{

i -= nonArrayElementCount;

pResolvedObject->scriptContext = scriptContext;

pResolvedObject->obj = typedArrayObj->DirectGetItem(i);

pResolvedObject->typeId = JavascriptOperators::GetTypeId(pResolvedObject->obj);

StringBuilder<ArenaAllocator>\* builder = GetBuilder();

Assert(builder);

builder->Reset();

pResolvedObject->name = GetIndexName(i, builder);

Assert(pResolvedObject->typeId != TypeIds\_HostDispatch);

pResolvedObject->address = Anew(GetArenaFromContext(scriptContext),

RecyclableTypedArrayAddress,

instance,

i);

}

return TRUE;

}

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

// RecyclableES5ArrayAddress

RecyclableES5ArrayAddress::RecyclableES5ArrayAddress(Var \_parentArray, unsigned int \_index)

: RecyclableArrayAddress(\_parentArray, \_index)

{

}

BOOL RecyclableES5ArrayAddress::Set(Var updateObject)

{

if (Js::ES5Array::Is(parentArray))

{

Js::ES5Array\* arrayObj = Js::ES5Array::FromVar(parentArray);

return arrayObj->SetItem(index, updateObject, PropertyOperation\_None);

}

return FALSE;

}

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

// RecyclableES5ArrayDisplay

RecyclableES5ArrayDisplay::RecyclableES5ArrayDisplay(ResolvedObject\* resolvedObject)

: RecyclableArrayDisplay(resolvedObject)

{

}

BOOL RecyclableES5ArrayDisplay::HasChildren()

{

if (Js::ES5Array::Is(instance))

{

Js::JavascriptArray\* arrayObj = static\_cast<Js::JavascriptArray \*>(RecyclableObject::FromVar(instance));

if (HasChildrenInternal(arrayObj))

{

return TRUE;

}

}

return RecyclableObjectDisplay::HasChildren();

}

WeakArenaReference<IDiagObjectModelWalkerBase>\* RecyclableES5ArrayDisplay::CreateWalker()

{

return CreateAWalker<RecyclableES5ArrayWalker>(scriptContext, instance, originalInstance);

}

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

// RecyclableES5ArrayWalker

RecyclableES5ArrayWalker::RecyclableES5ArrayWalker(ScriptContext\* \_scriptContext, Var \_instance, Var \_originalInstance)

: RecyclableArrayWalker(\_scriptContext, \_instance, \_originalInstance)

{

}

uint32 RecyclableES5ArrayWalker::GetNextDescriptor(uint32 currentDescriptor)

{

Js::ES5Array \*es5Array = static\_cast<Js::ES5Array \*>(RecyclableObject::FromVar(instance));

IndexPropertyDescriptor\* descriptor = nullptr;

void \* descriptorValidationToken = nullptr;

return es5Array->GetNextDescriptor(currentDescriptor, &descriptor, &descriptorValidationToken);

}

BOOL RecyclableES5ArrayWalker::FetchItemAtIndex(Js::JavascriptArray\* arrayObj, uint32 index, Var \*value)

{

Assert(arrayObj);

Assert(value);

return arrayObj->GetItem(arrayObj, index, value, scriptContext);

}

Var RecyclableES5ArrayWalker::FetchItemAt(Js::JavascriptArray\* arrayObj, uint32 index)

{

Assert(arrayObj);

Var value = nullptr;

if (FetchItemAtIndex(arrayObj, index, &value))

{

return value;

}

return nullptr;

}

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

// RecyclableProtoObjectWalker

RecyclableProtoObjectWalker::RecyclableProtoObjectWalker(ScriptContext\* pContext, Var instance, Var originalInstance)

: RecyclableObjectWalker(pContext, instance)

{

this->originalInstance = originalInstance;

}

BOOL RecyclableProtoObjectWalker::GetGroupObject(ResolvedObject\* pResolvedObject)

{

Assert(pResolvedObject);

DBGPROP\_ATTRIB\_FLAGS defaultAttributes = DBGPROP\_ATTRIB\_NO\_ATTRIB;

if (scriptContext->GetLibrary()->GetObjectPrototypeObject()->is\_\_proto\_\_Enabled())

{

pResolvedObject->name = L"\_\_proto\_\_";

pResolvedObject->propId = PropertyIds::\_\_proto\_\_;

}

else

{

pResolvedObject->name = L"[prototype]";

pResolvedObject->propId = Constants::NoProperty; // This property will not be editable.

defaultAttributes = DBGPROP\_ATTRIB\_VALUE\_IS\_FAKE;

}

RecyclableObject \*obj = Js::RecyclableObject::FromVar(instance);

Assert(obj->GetPrototype() != nullptr);

//withscopeObjects prototype is null

Assert(obj->GetPrototype()->GetTypeId() != TypeIds\_Null || (obj->GetPrototype()->GetTypeId() == TypeIds\_Null && obj->GetTypeId() == TypeIds\_WithScopeObject));

pResolvedObject->obj = obj->GetPrototype();

pResolvedObject->originalObj = (originalInstance != nullptr) ? Js::RecyclableObject::FromVar(originalInstance) : pResolvedObject->obj;

pResolvedObject->scriptContext = scriptContext;

pResolvedObject->typeId = JavascriptOperators::GetTypeId(pResolvedObject->obj);

ArenaAllocator \* arena = GetArenaFromContext(scriptContext);

pResolvedObject->objectDisplay = pResolvedObject->CreateDisplay();

pResolvedObject->objectDisplay->SetDefaultTypeAttribute(defaultAttributes);

pResolvedObject->address = Anew(arena,

RecyclableProtoObjectAddress,

instance,

PropertyIds::prototype,

pResolvedObject->obj);

return TRUE;

}

IDiagObjectAddress\* RecyclableProtoObjectWalker::FindPropertyAddress(PropertyId propId, bool& isConst)

{

ResolvedObject resolvedProto;

GetGroupObject(&resolvedProto);

struct AutoCleanup

{

WeakArenaReference<Js::IDiagObjectModelWalkerBase> \* walkerRef;

IDiagObjectModelWalkerBase \* walker;

AutoCleanup() : walkerRef(nullptr), walker(nullptr) {};

~AutoCleanup()

{

if (walker)

{

walkerRef->ReleaseStrongReference();

}

if (walkerRef)

{

HeapDelete(walkerRef);

}

}

} autoCleanup;

Assert(resolvedProto.objectDisplay);

autoCleanup.walkerRef = resolvedProto.objectDisplay->CreateWalker();

autoCleanup.walker = autoCleanup.walkerRef->GetStrongReference();

return autoCleanup.walker ? autoCleanup.walker->FindPropertyAddress(propId, isConst) : nullptr;

}

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

// RecyclableProtoObjectAddress

RecyclableProtoObjectAddress::RecyclableProtoObjectAddress(Var \_parentObj, Js::PropertyId \_propId, Js::Var \_value)

: RecyclableObjectAddress(\_parentObj, \_propId, \_value, false /\*isInDeadZone\*/)

{

}

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

// RecyclableCollectionObjectWalker

template <typename TData> const wchar\_t\* RecyclableCollectionObjectWalker<TData>::Name() { static\_assert(false, L"Must use specialization"); }

template <> const wchar\_t\* RecyclableCollectionObjectWalker<JavascriptMap>::Name() { return L"[Map]"; }

template <> const wchar\_t\* RecyclableCollectionObjectWalker<JavascriptSet>::Name() { return L"[Set]"; }

template <> const wchar\_t\* RecyclableCollectionObjectWalker<JavascriptWeakMap>::Name() { return L"[WeakMap]"; }

template <> const wchar\_t\* RecyclableCollectionObjectWalker<JavascriptWeakSet>::Name() { return L"[WeakSet]"; }

template <typename TData>

BOOL RecyclableCollectionObjectWalker<TData>::GetGroupObject(ResolvedObject\* pResolvedObject)

{

pResolvedObject->name = Name();

pResolvedObject->propId = Constants::NoProperty;

pResolvedObject->obj = instance;

pResolvedObject->scriptContext = scriptContext;

pResolvedObject->typeId = JavascriptOperators::GetTypeId(pResolvedObject->obj);

pResolvedObject->address = nullptr;

typedef RecyclableCollectionObjectDisplay<TData> RecyclableDataObjectDisplay;

pResolvedObject->objectDisplay = Anew(GetArenaFromContext(scriptContext), RecyclableDataObjectDisplay, scriptContext, pResolvedObject->name, this);

return TRUE;

}

template <typename TData>

BOOL RecyclableCollectionObjectWalker<TData>::Get(int i, ResolvedObject\* pResolvedObject)

{

auto builder = scriptContext->GetThreadContext()->GetDebugManager()->pCurrentInterpreterLocation->stringBuilder;

builder->Reset();

builder->AppendUint64(i);

pResolvedObject->name = builder->Detach();

pResolvedObject->propId = Constants::NoProperty;

pResolvedObject->obj = instance;

pResolvedObject->scriptContext = scriptContext;

pResolvedObject->typeId = JavascriptOperators::GetTypeId(pResolvedObject->obj);

pResolvedObject->address = nullptr;

pResolvedObject->objectDisplay = CreateTDataDisplay(pResolvedObject, i);

return TRUE;

}

template <typename TData>

IDiagObjectModelDisplay\* RecyclableCollectionObjectWalker<TData>::CreateTDataDisplay(ResolvedObject\* resolvedObject, int i)

{

Var key = propertyList->Item(i).key;

Var value = propertyList->Item(i).value;

return Anew(GetArenaFromContext(scriptContext), RecyclableKeyValueDisplay, resolvedObject->scriptContext, key, value, resolvedObject->name);

}

template <>

IDiagObjectModelDisplay\* RecyclableCollectionObjectWalker<JavascriptSet>::CreateTDataDisplay(ResolvedObject\* resolvedObject, int i)

{

resolvedObject->obj = propertyList->Item(i).value;

IDiagObjectModelDisplay\* display = resolvedObject->CreateDisplay();

display->SetDefaultTypeAttribute(DBGPROP\_ATTRIB\_VALUE\_READONLY | DBGPROP\_ATTRIB\_VALUE\_IS\_FAKE);

return display;

}

template <>

IDiagObjectModelDisplay\* RecyclableCollectionObjectWalker<JavascriptWeakSet>::CreateTDataDisplay(ResolvedObject\* resolvedObject, int i)

{

resolvedObject->obj = propertyList->Item(i).value;

IDiagObjectModelDisplay\* display = resolvedObject->CreateDisplay();

display->SetDefaultTypeAttribute(DBGPROP\_ATTRIB\_VALUE\_READONLY | DBGPROP\_ATTRIB\_VALUE\_IS\_FAKE);

return display;

}

template <typename TData>

ulong RecyclableCollectionObjectWalker<TData>::GetChildrenCount()

{

TData\* data = TData::FromVar(instance);

if (data->Size() > 0 && propertyList == nullptr)

{

propertyList = JsUtil::List<RecyclableCollectionObjectWalkerPropertyData<TData>, ArenaAllocator>::New(GetArenaFromContext(scriptContext));

GetChildren();

}

return data->Size();

}

template <>

void RecyclableCollectionObjectWalker<JavascriptMap>::GetChildren()

{

JavascriptMap\* data = JavascriptMap::FromVar(instance);

auto iterator = data->GetIterator();

while (iterator.Next())

{

Var key = iterator.Current().Key();

Var value = iterator.Current().Value();

propertyList->Add(RecyclableCollectionObjectWalkerPropertyData<JavascriptMap>(key, value));

}

}

template <>

void RecyclableCollectionObjectWalker<JavascriptSet>::GetChildren()

{

JavascriptSet\* data = JavascriptSet::FromVar(instance);

auto iterator = data->GetIterator();

while (iterator.Next())

{

Var value = iterator.Current();

propertyList->Add(RecyclableCollectionObjectWalkerPropertyData<JavascriptSet>(value));

}

}

template <>

void RecyclableCollectionObjectWalker<JavascriptWeakMap>::GetChildren()

{

JavascriptWeakMap\* data = JavascriptWeakMap::FromVar(instance);

data->Map([&](Var key, Var value)

{

propertyList->Add(RecyclableCollectionObjectWalkerPropertyData<JavascriptWeakMap>(key, value));

});

}

template <>

void RecyclableCollectionObjectWalker<JavascriptWeakSet>::GetChildren()

{

JavascriptWeakSet\* data = JavascriptWeakSet::FromVar(instance);

data->Map([&](Var value)

{

propertyList->Add(RecyclableCollectionObjectWalkerPropertyData<JavascriptWeakSet>(value));

});

}

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

// RecyclableCollectionObjectDisplay

template <typename TData>

LPCWSTR RecyclableCollectionObjectDisplay<TData>::Value(int radix)

{

StringBuilder<ArenaAllocator>\* builder = scriptContext->GetThreadContext()->GetDebugManager()->pCurrentInterpreterLocation->stringBuilder;

builder->Reset();

builder->AppendCppLiteral(L"size = ");

builder->AppendUint64(walker->GetChildrenCount());

return builder->Detach();

}

template <typename TData>

WeakArenaReference<IDiagObjectModelWalkerBase>\* RecyclableCollectionObjectDisplay<TData>::CreateWalker()

{

if (walker)

{

ReferencedArenaAdapter\* pRefArena = scriptContext->GetThreadContext()->GetDebugManager()->GetDiagnosticArena();

if (pRefArena)

{

return HeapNew(WeakArenaReference<IDiagObjectModelWalkerBase>, pRefArena, walker);

}

}

return nullptr;

}

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

// RecyclableKeyValueDisplay

WeakArenaReference<IDiagObjectModelWalkerBase>\* RecyclableKeyValueDisplay::CreateWalker()

{

ReferencedArenaAdapter\* pRefArena = scriptContext->GetThreadContext()->GetDebugManager()->GetDiagnosticArena();

if (pRefArena)

{

IDiagObjectModelWalkerBase\* pOMWalker = Anew(pRefArena->Arena(), RecyclableKeyValueWalker, scriptContext, key, value);

return HeapNew(WeakArenaReference<IDiagObjectModelWalkerBase>, pRefArena, pOMWalker);

}

return nullptr;

}

LPCWSTR RecyclableKeyValueDisplay::Value(int radix)

{

ResolvedObject ro;

ro.scriptContext = scriptContext;

ro.obj = key;

RecyclableObjectDisplay keyDisplay(&ro);

ro.obj = value;

RecyclableObjectDisplay valueDisplay(&ro);

// Note, RecyclableObjectDisplay::Value(int) uses the shared string builder

// so we cannot call it while building our string below. Call both before hand.

const wchar\_t\* keyValue = keyDisplay.Value(radix);

const wchar\_t\* valueValue = valueDisplay.Value(radix);

StringBuilder<ArenaAllocator>\* builder = scriptContext->GetThreadContext()->GetDebugManager()->pCurrentInterpreterLocation->stringBuilder;

builder->Reset();

builder->Append('[');

builder->AppendSz(keyValue);

builder->AppendCppLiteral(L", ");

builder->AppendSz(valueValue);

builder->Append(']');

return builder->Detach();

}

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

// RecyclableKeyValueWalker

BOOL RecyclableKeyValueWalker::Get(int i, ResolvedObject\* pResolvedObject)

{

if (i == 0)

{

pResolvedObject->name = L"key";

pResolvedObject->obj = key;

}

else if (i == 1)

{

pResolvedObject->name = L"value";

pResolvedObject->obj = value;

}

else

{

Assert(false);

return FALSE;

}

pResolvedObject->propId = Constants::NoProperty;

pResolvedObject->scriptContext = scriptContext;

pResolvedObject->typeId = JavascriptOperators::GetTypeId(pResolvedObject->obj);

pResolvedObject->objectDisplay = pResolvedObject->CreateDisplay();

pResolvedObject->objectDisplay->SetDefaultTypeAttribute(DBGPROP\_ATTRIB\_VALUE\_READONLY | DBGPROP\_ATTRIB\_VALUE\_IS\_FAKE);

pResolvedObject->address = nullptr;

return TRUE;

}

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

// RecyclableProxyObjectDisplay

RecyclableProxyObjectDisplay::RecyclableProxyObjectDisplay(ResolvedObject\* resolvedObject)

: RecyclableObjectDisplay(resolvedObject)

{

}

WeakArenaReference<IDiagObjectModelWalkerBase>\* RecyclableProxyObjectDisplay::CreateWalker()

{

ReferencedArenaAdapter\* pRefArena = scriptContext->GetThreadContext()->GetDebugManager()->GetDiagnosticArena();

if (pRefArena)

{

IDiagObjectModelWalkerBase\* pOMWalker = Anew(pRefArena->Arena(), RecyclableProxyObjectWalker, scriptContext, instance);

return HeapNew(WeakArenaReference<IDiagObjectModelWalkerBase>, pRefArena, pOMWalker);

}

return nullptr;

}

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

// RecyclableProxyObjectWalker

RecyclableProxyObjectWalker::RecyclableProxyObjectWalker(ScriptContext\* pContext, Var \_instance)

: RecyclableObjectWalker(pContext, \_instance)

{

}

BOOL RecyclableProxyObjectWalker::GetGroupObject(ResolvedObject\* pResolvedObject)

{

pResolvedObject->name = L"[Proxy]";

pResolvedObject->propId = Constants::NoProperty;

pResolvedObject->obj = instance;

pResolvedObject->scriptContext = scriptContext;

pResolvedObject->typeId = JavascriptOperators::GetTypeId(pResolvedObject->obj);

pResolvedObject->address = nullptr;

pResolvedObject->objectDisplay = Anew(GetArenaFromContext(scriptContext), RecyclableProxyObjectDisplay, pResolvedObject);

pResolvedObject->objectDisplay->SetDefaultTypeAttribute(DBGPROP\_ATTRIB\_VALUE\_READONLY | DBGPROP\_ATTRIB\_VALUE\_IS\_FAKE);

return TRUE;

}

BOOL RecyclableProxyObjectWalker::Get(int i, ResolvedObject\* pResolvedObject)

{

JavascriptProxy\* proxy = JavascriptProxy::FromVar(instance);

if (i == 0)

{

pResolvedObject->name = L"[target]";

pResolvedObject->obj = proxy->GetTarget();

}

else if (i == 1)

{

pResolvedObject->name = L"[handler]";

pResolvedObject->obj = proxy->GetHandler();

}

else

{

Assert(false);

return FALSE;

}

pResolvedObject->propId = Constants::NoProperty;

pResolvedObject->scriptContext = scriptContext;

pResolvedObject->typeId = JavascriptOperators::GetTypeId(pResolvedObject->obj);

pResolvedObject->objectDisplay = pResolvedObject->CreateDisplay();

pResolvedObject->objectDisplay->SetDefaultTypeAttribute(DBGPROP\_ATTRIB\_VALUE\_READONLY | DBGPROP\_ATTRIB\_VALUE\_IS\_FAKE);

pResolvedObject->address = Anew(GetArenaFromContext(pResolvedObject->scriptContext),

RecyclableObjectAddress,

pResolvedObject->scriptContext->GetGlobalObject(),

Js::PropertyIds::Proxy,

pResolvedObject->obj,

false /\*isInDeadZone\*/);

return TRUE;

}

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

// RecyclableMethodsGroupWalker

RecyclableMethodsGroupWalker::RecyclableMethodsGroupWalker(ScriptContext\* scriptContext, Var instance)

: RecyclableObjectWalker(scriptContext,instance)

{

}

void RecyclableMethodsGroupWalker::AddItem(Js::PropertyId propertyId, Var obj)

{

if (pMembersList == nullptr)

{

pMembersList = JsUtil::List<DebuggerPropertyDisplayInfo \*, ArenaAllocator>::New(GetArenaFromContext(scriptContext));

}

Assert(pMembersList);

DebuggerPropertyDisplayInfo \*info = Anew(GetArenaFromContext(scriptContext), DebuggerPropertyDisplayInfo, propertyId, obj, DebuggerPropertyDisplayInfoFlags\_Const);

Assert(info);

pMembersList->Add(info);

}

ulong RecyclableMethodsGroupWalker::GetChildrenCount()

{

return pMembersList ? pMembersList->Count() : 0;

}

BOOL RecyclableMethodsGroupWalker::Get(int i, ResolvedObject\* pResolvedObject)

{

AssertMsg(pResolvedObject, "Bad usage of RecyclableMethodsGroupWalker::Get");

return RecyclableObjectWalker::Get(i, pResolvedObject);

}

BOOL RecyclableMethodsGroupWalker::GetGroupObject(ResolvedObject\* pResolvedObject)

{

Assert(pResolvedObject);

// This is fake [Methods] object.

pResolvedObject->name = L"[Methods]";

pResolvedObject->obj = Js::RecyclableObject::FromVar(instance);

pResolvedObject->scriptContext = scriptContext;

pResolvedObject->typeId = JavascriptOperators::GetTypeId(pResolvedObject->obj);

pResolvedObject->address = nullptr; // Methods object will not be editable

pResolvedObject->objectDisplay = Anew(GetArenaFromContext(scriptContext), RecyclableMethodsGroupDisplay, this, pResolvedObject);

return TRUE;

}

void RecyclableMethodsGroupWalker::Sort()

{

pMembersList->Sort(ElementsComparer, scriptContext);

}

RecyclableMethodsGroupDisplay::RecyclableMethodsGroupDisplay(RecyclableMethodsGroupWalker \*\_methodGroupWalker, ResolvedObject\* resolvedObject)

: methodGroupWalker(\_methodGroupWalker),

RecyclableObjectDisplay(resolvedObject)

{

}

LPCWSTR RecyclableMethodsGroupDisplay::Type()

{

return L"";

}

LPCWSTR RecyclableMethodsGroupDisplay::Value(int radix)

{

return L"{...}";

}

BOOL RecyclableMethodsGroupDisplay::HasChildren()

{

return methodGroupWalker ? TRUE : FALSE;

}

DBGPROP\_ATTRIB\_FLAGS RecyclableMethodsGroupDisplay::GetTypeAttribute()

{

return DBGPROP\_ATTRIB\_VALUE\_READONLY | DBGPROP\_ATTRIB\_VALUE\_IS\_FAKE | DBGPROP\_ATTRIB\_VALUE\_IS\_METHOD | DBGPROP\_ATTRIB\_VALUE\_IS\_EXPANDABLE;

}

WeakArenaReference<IDiagObjectModelWalkerBase>\* RecyclableMethodsGroupDisplay::CreateWalker()

{

if (methodGroupWalker)

{

ReferencedArenaAdapter\* pRefArena = scriptContext->GetThreadContext()->GetDebugManager()->GetDiagnosticArena();

if (pRefArena)

{

return HeapNew(WeakArenaReference<IDiagObjectModelWalkerBase>, pRefArena, methodGroupWalker);

}

}

return nullptr;

}

ScopeVariablesGroupDisplay::ScopeVariablesGroupDisplay(VariableWalkerBase \*walker, ResolvedObject\* resolvedObject)

: scopeGroupWalker(walker),

RecyclableObjectDisplay(resolvedObject)

{

}

LPCWSTR ScopeVariablesGroupDisplay::Type()

{

return L"";

}

LPCWSTR ScopeVariablesGroupDisplay::Value(int radix)

{

if (ActivationObject::Is(instance))

{

// The scope is defined by the activation object.

Js::RecyclableObject \*object = Js::RecyclableObject::FromVar(instance);

try

{

// Trying to find out the JavascriptFunction from the scope.

Var value = nullptr;

if (object->GetTypeId() == TypeIds\_ActivationObject && GetPropertyWithScriptEnter(object, object, PropertyIds::arguments, &value, scriptContext))

{

if (Js::RecyclableObject::Is(value))

{

Js::RecyclableObject \*argObject = Js::RecyclableObject::FromVar(value);

Var calleeFunc = nullptr;

if (GetPropertyWithScriptEnter(argObject, argObject, PropertyIds::callee, &calleeFunc, scriptContext) && Js::JavascriptFunction::Is(calleeFunc))

{

Js::JavascriptFunction \*calleeFunction = Js::JavascriptFunction::FromVar(calleeFunc);

Js::FunctionBody \*pFuncBody = calleeFunction->GetFunctionBody();

if (pFuncBody)

{

const wchar\_t\* pDisplayName = pFuncBody->GetDisplayName();

if (pDisplayName)

{

StringBuilder<ArenaAllocator>\* builder = GetStringBuilder();

builder->Reset();

builder->AppendSz(pDisplayName);

return builder->Detach();

}

}

}

}

}

}

catch(Js::JavascriptExceptionObject \*exceptionObject)

{

exceptionObject;

// Not doing anything over here.

}

return L"";

}

else

{

// The scope is defined by a slot array object so grab the function body out to get the function name.

ScopeSlots slotArray = ScopeSlots(reinterpret\_cast<Var\*>(instance));

if(slotArray.IsFunctionScopeSlotArray())

{

Js::FunctionBody \*functionBody = slotArray.GetFunctionBody();

return functionBody->GetDisplayName();

}

else

{

// handling for block/catch scope

return L"";

}

}

}

BOOL ScopeVariablesGroupDisplay::HasChildren()

{

return scopeGroupWalker ? TRUE : FALSE;

}

DBGPROP\_ATTRIB\_FLAGS ScopeVariablesGroupDisplay::GetTypeAttribute()

{

return DBGPROP\_ATTRIB\_VALUE\_READONLY | DBGPROP\_ATTRIB\_VALUE\_IS\_FAKE | DBGPROP\_ATTRIB\_VALUE\_IS\_EXPANDABLE;

}

WeakArenaReference<IDiagObjectModelWalkerBase>\* ScopeVariablesGroupDisplay::CreateWalker()

{

if (scopeGroupWalker)

{

ReferencedArenaAdapter\* pRefArena = scriptContext->GetThreadContext()->GetDebugManager()->GetDiagnosticArena();

if (pRefArena)

{

return HeapNew(WeakArenaReference<IDiagObjectModelWalkerBase>, pRefArena, scopeGroupWalker);

}

}

return nullptr;

}

GlobalsScopeVariablesGroupDisplay::GlobalsScopeVariablesGroupDisplay(VariableWalkerBase \*walker, ResolvedObject\* resolvedObject)

: globalsGroupWalker(walker),

RecyclableObjectDisplay(resolvedObject)

{

}

LPCWSTR GlobalsScopeVariablesGroupDisplay::Type()

{

return L"";

}

LPCWSTR GlobalsScopeVariablesGroupDisplay::Value(int radix)

{

return L"";

}

BOOL GlobalsScopeVariablesGroupDisplay::HasChildren()

{

return globalsGroupWalker ? globalsGroupWalker->GetChildrenCount() > 0 : FALSE;

}

DBGPROP\_ATTRIB\_FLAGS GlobalsScopeVariablesGroupDisplay::GetTypeAttribute()

{

return DBGPROP\_ATTRIB\_VALUE\_READONLY | DBGPROP\_ATTRIB\_VALUE\_IS\_FAKE | (HasChildren() ? DBGPROP\_ATTRIB\_VALUE\_IS\_EXPANDABLE : 0);

}

WeakArenaReference<IDiagObjectModelWalkerBase>\* GlobalsScopeVariablesGroupDisplay::CreateWalker()

{

if (globalsGroupWalker)

{

ReferencedArenaAdapter\* pRefArena = scriptContext->GetThreadContext()->GetDebugManager()->GetDiagnosticArena();

if (pRefArena)

{

return HeapNew(WeakArenaReference<IDiagObjectModelWalkerBase>, pRefArena, globalsGroupWalker);

}

}

return nullptr;

}

#ifdef ENABLE\_MUTATION\_BREAKPOINT

PendingMutationBreakpointDisplay::PendingMutationBreakpointDisplay(ResolvedObject\* resolvedObject, MutationType \_mutationType)

: RecyclableObjectDisplay(resolvedObject), mutationType(\_mutationType)

{

AssertMsg(\_mutationType > MutationTypeNone && \_mutationType < MutationTypeAll, "Invalid mutationType value passed to PendingMutationBreakpointDisplay");

}

WeakArenaReference<IDiagObjectModelWalkerBase>\* PendingMutationBreakpointDisplay::CreateWalker()

{

ReferencedArenaAdapter\* pRefArena = scriptContext->GetThreadContext()->GetDebugManager()->GetDiagnosticArena();

if (pRefArena)

{

IDiagObjectModelWalkerBase\* pOMWalker = Anew(pRefArena->Arena(), PendingMutationBreakpointWalker, scriptContext, instance, this->mutationType);

return HeapNew(WeakArenaReference<IDiagObjectModelWalkerBase>, pRefArena, pOMWalker);

}

return nullptr;

}

ulong PendingMutationBreakpointWalker::GetChildrenCount()

{

switch (this->mutationType)

{

case MutationTypeUpdate:

return 3;

case MutationTypeDelete:

case MutationTypeAdd:

return 2;

default:

AssertMsg(false, "Invalid mutationType");

return 0;

}

}

PendingMutationBreakpointWalker::PendingMutationBreakpointWalker(ScriptContext\* pContext, Var \_instance, MutationType mutationType)

: RecyclableObjectWalker(pContext, \_instance)

{

this->mutationType = mutationType;

}

BOOL PendingMutationBreakpointWalker::Get(int i, ResolvedObject\* pResolvedObject)

{

Js::MutationBreakpoint \*mutationBreakpoint = scriptContext->GetDebugContext()->GetProbeContainer()->GetDebugManager()->GetActiveMutationBreakpoint();

Assert(mutationBreakpoint);

if (mutationBreakpoint != nullptr)

{

if (i == 0)

{

// <Property Name> [Adding] : New Value

// <Property Name> [Changing] : Old Value

// <Property Name> [Deleting] : Old Value

WCHAR \* displayName = AnewArray(GetArenaFromContext(scriptContext), WCHAR, PENDING\_MUTATION\_VALUE\_MAX\_NAME);

swprintf\_s(displayName, PENDING\_MUTATION\_VALUE\_MAX\_NAME, L"%s [%s]", mutationBreakpoint->GetBreakPropertyName(), Js::MutationBreakpoint::GetBreakMutationTypeName(mutationType));

pResolvedObject->name = displayName;

if (mutationType == MutationTypeUpdate || mutationType == MutationTypeDelete)

{

// Old/Current value

PropertyId breakPId = mutationBreakpoint->GetBreakPropertyId();

pResolvedObject->propId = breakPId;

pResolvedObject->obj = JavascriptOperators::OP\_GetProperty(mutationBreakpoint->GetMutationObjectVar(), breakPId, scriptContext);

}

else

{

// New Value

pResolvedObject->obj = mutationBreakpoint->GetBreakNewValueVar();

pResolvedObject->propId = Constants::NoProperty;

}

}

else if ((i == 1) && (mutationType == MutationTypeUpdate))

{

pResolvedObject->name = L"[New Value]";

pResolvedObject->obj = mutationBreakpoint->GetBreakNewValueVar();

pResolvedObject->propId = Constants::NoProperty;

}

else if (((i == 1) && (mutationType != MutationTypeUpdate)) || (i == 2))

{

WCHAR \* displayName = AnewArray(GetArenaFromContext(scriptContext), WCHAR, PENDING\_MUTATION\_VALUE\_MAX\_NAME);

swprintf\_s(displayName, PENDING\_MUTATION\_VALUE\_MAX\_NAME, L"[Property container %s]", mutationBreakpoint->GetParentPropertyName());

pResolvedObject->name = displayName;

pResolvedObject->obj = mutationBreakpoint->GetMutationObjectVar();

pResolvedObject->propId = mutationBreakpoint->GetParentPropertyId();

}

else

{

Assert(false);

return FALSE;

}

pResolvedObject->scriptContext = scriptContext;

pResolvedObject->typeId = JavascriptOperators::GetTypeId(pResolvedObject->obj);

pResolvedObject->objectDisplay = pResolvedObject->CreateDisplay();

pResolvedObject->objectDisplay->SetDefaultTypeAttribute(DBGPROP\_ATTRIB\_VALUE\_READONLY | DBGPROP\_ATTRIB\_VALUE\_IS\_FAKE);

pResolvedObject->address = nullptr; // TODO: (SaAgarwa) Currently Pending mutation values are not editable, will do as part of another WI

return TRUE;

}

return FALSE;

}

#endif

}