Specification Level Nominal Typing of Objects in ECMASCript 5 and Harmony

There are at least two forms of nominal typing used within the ES5 specification. The first is the set of nominal "ECMAScript types" defined in section 8 that classify the primitive values forms used within the specification. A subset of these, the "ECMAScript language types" correspond to the actual kinds of values (Undefined, Null, Boolean, Number, String, Object) that are directly visible to ECMAScript programmers.

The second use of nominal typing in the ES5 specification is its internal subclassification of various forms of the Object ECMAScript type. The goal of this paper is to analyze to use of this second form of nominal Object typing in order to better understand if and how it could be better reified within ECMAScript Harmony and whether support for this typing model needs to be incorporated into Harmony Proxy objects.

Informal Object Typing in ES5

Various parts of the ES5 specification actually use several different semi-formal and informal techniques to identify and classify different kinds of ECMAScript objects. The techniques include:

- Identification of the specific string values with the [[Class]] internal property of objects.
- Informal reference to specific kinds of objects (usually defined in section 15) using phrases such as "String objects" which, for example, refers to objects that have the characteristics of "String Instances" as defined in 15.5.5.
- Identification by the presence of specific internal state properties such as [[PrimitiveValue]].
- Identification by the presence or absence of specific internal methods such as [Call]].
- Identification by specific over-rides of common internal methods such as [[DefineOwnProperty]].

The use of each of these techniques occur somewhere in the ES5 specification but not, necessarily in any systematic manner. The differences seem to have arisen over time as editorial style variations by different specification authors and also reflect an evolving emphasis on specification preciseness. The various techniques do not always yield identical classification of objects. For example, the Math and JSON objects have different values for their [[Class]] internal property but otherwise present identical set of internal properties and internal methods.

Among them, the object classifications are used for various purposes within the specification including:

- Explicit data parameterization of built-in functions based upon object "type". For example, Object.prototype.toString.
- Explicit behavioral parameterization of built-in functions based upon object "type". For example, Array.prototype.concat.
- Explicit behavioral parameterization of language features based upon the object "type" of operands. For example, the delete operator's use of the [[Delete]] internal method.
- Association of private runtime state with specific object instances where the state is used by various specification algorithms. For example, the [[Prototype]] internal property.
- Association of abstracted private state with specific object instances where the actual runtime state space is an implementation detail. For example, the [[Scope]] and [[Code]] internal properties of function objects.
- Association of abstracted private state with specific object categories where the state is only used for specification purposes and need not actually exists at runtime. For example, the [[FormalParameters]] internal property of function objects.
- Specifying a dependency between private runtime state of specific object instances and specific built-in methods that use that state. Date methods that depend upon the [[PrimitiveValue]] property.
- Specifying a dependency between object type specific internal behaviors and specific built-in methods that use those behaviors. For example, String.prototype.split method's use of [[Match]] internal method.

Impact of Specification Nominal Typing on Proxy Functionality

The typing techniques described above are all used exclusively as specification devices. They do not require that any specific implementation techniques be used to by an implication to achieve the specified results. For example, an implementation might use a tag value to indicate specific type or it might describe based upon implementation language typing such as a C++ class type. In addition, modern ECMAScript implementations typically use other typing concepts to distinguish objects for optimization purposes such as polymorphic inline caching.

Historically, these various typing specification techniques and the implementation techniques used to conform to the specification had little if any impact upon interoperability among ECMAScript implementations. In particular, existence of these typing schemes and their implementations were generally not directly observable by ECMAScript code. However, the introduction of Proxies in Harmony potentially exposes these details. Proxies are intended to allow ECMAScript code to define new behavioral variants of a type that previous could have only been achieved by modify an implementation of an ECMAScript. These Proxy-based objects must correctly interact with the internal mechanisms of an implementation

including mechanisms that correspond to the internal object typing used within the ECMAScript specification. This requires careful identification of all such mechanisms used in the specification and the definition of public ECMAScript APIs that may be used by Proxy Handlers to provide integrate Proxy-based objects with these implementations.

While in the past, there was little reason to be concerned with the proliferation of these specification typing mechanisms the need to reify them for use by Proxy handlers is a strong inceptive to minimize the number of actual mechanisms that need to be exposed via the Proxy mechanism. For that reason, we need to carefully examine all existing internal typing use cases from that perspective.

ES5 Specification Use Cases for the [[Class]] Internal Property

The ES5 specification defines each objects as having an internal string-valued property named [[Class]]. There is a distinct class value assigned to instances of each of the major kinds of objects defined in section 15 ("Object", "Function", "Array", "String", "Boolean", "Number", "Math", "Date", "RegExp", "Error", and "JSON"). In addition, arguments objects have a [[Class]] value of "Arguments". Host objects may define other [[Class]] values except that they many not assign any of the values already used within the ES5 specification. However, except for its use by Object.prototype.toString such host object [[Class]] values have no significance in the context of the ES5 specification other than being distinct from the specified [[Class]] values.

ECMAScript implementations typically do not have an actual string-valued internal property slot corresponding to [[Class]] for each object. Instead, they use internal mechanisms of map objects to corresponding [[Class]] string values in the once situation where the string value is actually required. It is an implementation detail whether or not an open ended set of [[Class]] string values may be associated with host objects.

Data Parameterization of Object.prototype.toString Using [[Class]] Value

The essential steps in the definition of this built-in method are:

3. Let *O* be the result of calling ToObject passing the **this** value as the argument.
4. Let *class* be the value of the [[Class]] internal property of *O*.
5. Return the String value that is the result of concatenating the three Strings `"[object "`, *class*, and `"]"`.

From this, it may appear that for this usage [[Class]] must be an actual runtime value that is associated with each object. However, if we remember that this is a specification and not an actual runtime algorithm we can recast the above specification into a form that doesn't actually use [[Class]] at all:

3. Let *O* be the result of calling ToObject passing the **this** value as the argument.

4. If *O* is the built-in Object prototype object (15.2.4) or an object created by the standard built-in Object constructor (15.2.2.1) or by the [[Construct]] internal method as defined in (13.2.2) then let *class* be **"Object"**,
5. Else if *O* is the built-in String prototype object (15.5.4) or an object created by the standard built-in String constructor (15.5.2.1) then let *class* be **"String"**,
6. Else if *O* is the built-in Number prototype object (15.7.4) or an object created by the standard built-in Number constructor (15.7.2.1) then let *class* be **"Number"**,
7. Else if *O* is the built-in Boolean prototype object (15.6.4) or an object created by the standard built-in Boolean constructor (15.6.2.1) then let *class* be **"Boolean"**,
8. Else if *O* is the built-in Function prototype object (15.3.4) or an object created by algorithm 13.2 or algorithm 13.2.3, or by the `Function.prototype.bind` built-in method (15.3.4.5) or is any built-in function, method, or constructor object defined in clause 15 then let *class* be **"Function"**,
9. Else if *O* is the built-in Array prototype object (15.4.4) or an object created by the standard built-in Array constructors (15.4.2.1) or (15.4.2.2) then let *class* be **"Array"**,
10. Else if *O* is the built-in RegExp prototype object (15.10.6) or an object created by the standard built-in RegExp constructor (15.10.4.1) then let *class* be **"RegExp"**,
11. Else if *O* is the built-in Date prototype object (15.9.5) or an object created by the standard built-in Date constructors (15.9.3.1), (15.9.3.2), or (15.9.3.3) then let *class* be **"Date"**,
12. Else if *O* is the built-in Math object (15.8) then let *class* be **"Math"**,
13. Else if *O* is the built-in JSON object (15.12) then let *class* be **"JSON"**,
14. Else if *O* is the built-in Error prototype object (15.11.4) or an object created by the standard built-in Error constructors (15.11.1.1) or (15.11.2.1) then let *class* be **"Error"**,
15. Else if *O* is a built-in NativeError object created by an implementation (15.11.7) or an object created by a standard built-in NativeError constructor (15.11.7.4) then let *class* be **"Error"**,
16. Else if *O* is an arguments object created by as specified by algorithm (10.6) then let *class* be **"Arguments"**,
17. Else if a *O* is an implementation defined native built-object or an object created by an implementation defined native built-in constructor, then let *class* be an implementation defined string value,
18. Else if *O* is a host object, then let *class* be an implementation defined string value. However, in this case the value must not be one of **"Arguments"**, **"Array"**, **"Boolean"**, **"Date"**, **"Error"**, **"Function"**, **"JSON"**, **"Math"**, **"Number"**, **"Object"**, **"RegExp"**, and **"String"**.
19. Return the String value that is the result of concatenating the three Strings **"[object "**, *class*, and **"]"**.

Whether or not this is a better specification form is probably subject to debate. However, this transformation does demonstrate that the [[Class]] internal property is not essential to the semantics of Object.prototype.toString.

Behavioral Parameterization of JSON.stringify Using [[Class]] Values.

The specification for the built-in function JSONstringify (15.12.3) uses a test for specific [[Class]] values of objects to be processed to control the specific processing behavior applied to the objects.

This is generally a mess (largely of my own doing). Some are nominal type checks for Number, String, or Boolean wrappers objects. These could be replaces with

"instance of" checks. Other uses could be replaced with toJSON methods. It isn't clear why the replacer array needs to be an actual array object.

Private State Access of Boolean objects determined by [[Class]] value.

The specification for the built-in methods Boolean.prototype.toString (15.6.4.2) and Boolean.prototype.valueOf (15.6.4.3) use a test for [[Class]] equal to "Boolean" to determine whether the [[PrimitiveValue]] internal property of the this object can be accessed. Other similar built-in methods are specified without explicit reference to [[Class]] and that could also be done here.

Public State Access of Function objects determined by [[Class]] value

The specification for Function.prototype.bind (15.3.34.5) uses a test for [[Class]] equal to "Function" to determine if the target function must have a length property (as defined in 15.3.5.1) and then uses that value of that property directly without access via [[Get]]. This class test is independent of a isCallable test that determines whether the target object has [[Call]] internal method. (Callable objects need not be instances of Function).

This [[Class]] test could be replaces with an instance of test. Alternatively, the specification could be generalized to check for the existence of a length property on callable objects regardless or whether they are actual function instances.

Nominal Classification of Array objects based upon [[Class]] values

The built-in function Array.isArray (15.4.3.2) is specified to test for [[Class]] equal to "Array" to determine its result. Based upon the known history of this function, this test probably can not be replaced by a simple instance of test. In particular, it is known that this function was intended to classify "array objects" as such even when the object originated in a different global context than the testing code. In these situations the array object is not necessarily an instance of the Array constructor object of the current global context. The "Array" [[Class]] value is assumed to be independent of such global context dependencies. *It isn't clear without further analysis which if any of the uses of nominal classification within ES5 should be context independent in this manner.*

It isn't clear what the actual use cases are for this sort of context independent typing. It appears to have first been proposed in the context of JSON serializes that need to a determination of which objects to serialize using array literal notation. However, Array.isArray is not actually used by the ES5 JSON.stringify algorithm which instead directly tests for [[Class]]. It isn't clear why

The only unique characteristic of Array instances (15.4.5) are the invariants imposed by their specialized version of [[DefineOwnProperty]]. Specify Array.isArray based upon the presence of the specific internal method definition

would, in the context of the ES5 specification, be equivalent to testing for the "Array" [[Class]] value.

The built-in method Array.prototype.concat (15.4.4.4) tests each of its arguments for [[Class]] equal to "Array" to determine whether the argument should be concatenated as a single object or whether it should be considered a transparent container of objects that are individually concatenated. Because it is a class test it is context independent.

This test is problematic in that prevents used defined array-like objects from have this same transparent container behavior. This applied even to objects that have the Array prototype on their prototype chain. This transparent container treatment has no fundamental dependency upon the actual unique semantics of built-in arrays. It should be replaced with a check for some user-definable flag property that enables/disables transparent container treatment.

The built-in function JSON.parse (15.12.2) uses a [[Class]] equals "Array" test as it recursive walks the tree it had previous created. The test is used to determine whether all properties or only array index properties should be recursively walked.

This is purely a specification device. The objects that are being tested were created by a previous step of the algorithm and each object was explicitly created as either an array or as regular object based upon the original encoding as a *JSONValue*. The [[Class]] test in the algorithm can be restated using language such as "if *val* was created in correspondence to a *JSONArray*".

The built-in function JSON.stringify (15.12.3) uses a [[Class]] equals "Array" test on its *reviver* argument to determine if that argument is a collection of white-listed property names. Actually, it classifies *reviver* objects into three categories: functions via isCallable, arrays via [[Class]], and everything else. Object's in the everything else category are simply ignored. It isn't clear why generic array-like objects are not supported as white-list provider. If the array and everything else cases where merged into a generic "array-like" category then only the isCallable test would be necessary and the [[Class]] test could be eliminated. Alternatively, the same or similar "transparent container" marker property as proposed for concat could be used to identify while-list providers.

JSON.stringify also uses a [[Class]] test to determine whether an array literal or object literal syntax should be used to represent the object. This prevents user defined array-like objects from being directly serialized using array literal notation. This usage could be eliminated by a user-definable property value.

As noted above, many of these [[Class]]-based classification tests could be generalized by using an explicit object property to control the conditional behavior. Other this creates a tension between backwards compatibility and future usability. The simplest why to do this generalization would be to add additional properties to

Array.prototype (for example, isTransparentContainer, useJSONArray, etc.) and update the specification to remove the corresponding [[Class]] tests and only test for those properties. That would be the cleanest specification and would be easiest for future ECMAScript programmers. However, that would mean that existing code that create non-Array objects that inherit from Array.prototype would change its behavior in those situations. A compromise would be to recognize such new properties but to not add them to Array.prototype. In addition, the existing [[Class]] test would probably be replaced by an equivalent Array instance test. This would maintain backwards compatibility but would require future "subclasses" of array to explicit opt-in to the array-like behavior by defining the necessary properties on their own prototype or instances if it was desired. The compatibility and convenience risks of each approach need to be further considered.