Part III

Native ECMAScript Objects

1 Introduction

```
FILE:                          spec/library/intro.html
DRAFT STATUS:                  DRAFT 1 - ROUGH
REVIEWED AGAINST ES3:          NO
REVIEWED AGAINST PROPOSALS:    NO
REVIEWED AGAINST CODE:         NO
```

1 There are certain built-in objects available whenever an ECMAScript program begins execution. One, the global object, is in the scope chain of the executing program. Others are accessible as initial properties of the global object.

FIXME There may be multiple global objects.

2 Unless specified otherwise, the `[[Class]]` property of a built-in object is `"Class"` if that object is defined as a class, `"Function"` if that built-in object is not a class but has a `[[Call]]` property, or `"Object"` if that built-in object neither is a class nor has a `[[Call]]` property.

COMPATIBILITY NOTE The 3rd Edition of this Standard did not provide classes, and all built-in objects provided as classes in 4th Edition were previously provided as functions. The change from functions to classes is observable to programs that convert the built-in class objects to strings.

3 Many built-in objects behave like functions: they can be invoked with arguments. Some of them furthermore are constructors: they are classes intended for use with the `new` operator. For each built-in class, this specification describes the arguments required by that class's constructor and properties of the Class object. For each built-in class, this specification furthermore describes properties of the prototype object of that class and properties of specific object instances returned by a `new` expression that constucts instances of that class.

4 Built-in classes have four kinds of functions, collectively called methods: constructors, static methods, prototype methods, and intrinsic instance methods. Non-class built-in objects may additionally hold non-method functions.

COMPATIBILITY NOTE The 3rd Edition of this standard provided only constructors and prototype methods. The new methods are not visible to 3rd Edition code being executed by a 4th Edition implementation.

5 Unless otherwise specified in the description of a particular class, if a constructor, prototype method, or ordinary function described in this section is given fewer arguments than the function is specified to require, the function shall behave exactly as if it had been given sufficient additional arguments, each such argument being the `undefined` value.

6 Unless otherwise specified in the description of a particular class, if a constructor, prototype method, or ordinary function described in this section is given more arguments than the function is specified to allow, the behaviour of the function is undefined. In particular, an implementation is permitted (but not required) to throw a `TypeError` exception in this case.

NOTE Implementations that add additional capabilities to the set of built-in classes are encouraged to do so by adding new functions and methods rather than adding new parameters to existing functions and methods.

7 Every built-in function has the Function prototype object, which is the initial value of the expression `Function.prototype` (Function.prototype), as the value of its internal `[[Prototype]]` property.

8 Every built-in class has the Object prototype object, which is the initial value of the expression `Object.prototype` (Object.prototype), as the value of its internal `[[Prototype]]` property.

COMPATIBILITY NOTE In the 3rd Edition of this Standard every constructor function that is represented as a class in 4th Edition also had the Function prototype object as the value of its internal `[[Prototype]]` property. This change is observable to programs that attempt to call methods defined on the Function prototype object through a class object.

9 Every built-in prototype object has the Object prototype object, which is the initial value of the expression `Object.prototype` (Object.prototype), as the value of its internal `[[Prototype]]` property, except the Object prototype object itself.

10 None of the built-in functions described in this section shall implement the internal `[[Construct]]` method unless otherwise specified in the description of a particular function. None of the built-in functions described in this section shall initially have a `prototype` property unless otherwise specified in the description of a particular function. Every built-in Function object described in this section--whether as a constructor, an ordinary function, or a method--has a `length` property whose value is an integer. Unless otherwise specified, this value is equal to the largest number of named arguments shown in the section headings for the function description, including optional parameters.

NOTE For example, the Function object that is the initial value of the `slice` property of the String prototype object is described under the section heading `String.prototype.slice (start , end)` which shows the two named arguments start and end; therefore the value of the length property of that Function object is 2.

11 The built-in objects and functions are defined in terms of ECMAScript packages, namespaces, classes, types, methods, properties, and functions, with the help from a small number of implementation hooks.

NOTE Though the behavior and structure of built-in objects and functions is expressed in ECMAScript terms, implementations are not required to implement them in ECMAScript, only to preserve the behavior as it is defined in this Standard.

12 Implementation hooks manifest themselves as functions in the `magic` namespace, as in the definition of the intrinsic `toString` method on `Object` objects:

```
intrinsic function toString() : string
    "[object " + magic::getClassName(this) + "]";
```

13 All magic function definitions are collected in section library-magic.

14 The definitions of the built-in objects and functions also leave some room for the implementation to choose strategies for certain auxiliary and primitive operations. These variation points manifest themselves as functions in the `informative` namespace, as in the definition of the intrinsic global function `hashcode`:

```
intrinsic function hashcode(o): uint {
    switch type (o) {
    …
    case (x: String)    { return informative::stringHash(string(x)) }
    case (x: *)         { return informative::objectHash(x) }
    }
}
```

15 Informative methods and functions are defined non-operationally in the sections that make use of them.

16 The definitions of the built-in objects and functions also make use of internal helper functions and properties, written in ECMAScript. These helper functions and properties are not available to user programs and are included in this Standard for expository purposes, as they help to define the semantics of the functions that make use of them. Helper functions and properties manifest themselves as definitions in the `helper` namespace, as in the definition of the global `encodeURI` function:

```
intrinsic function encodeURI(uri: string): string
    helper::encode(uri, helper::uriReserved + helper::uriUnescaped + "#")
```

17 Helper functions and properties are defined where they are first used, but are sometimes referenced from multiple sections in this Standard.

FIXME We need a credible story for helper primitives like `ToString` and `ToNumeric`. In the current code, they are just treated like global functions; more appropriate would be if they were in a namespace like `helper` or `magic`.

18 Unless noted otherwise in the description of a particular class or function, the behavior of built-in objects is unaffected by definitions or assignments performed by the user program. This is accomplished first by defining all built-in objects, classes, functions, and properties inside a package whose name is private to the implementation, second by always preferring intrinsic methods and functions to prototype methods and unqualified functions, and finally by importing the public names of the package containing the built-ins into the global environment of the user program.

FIXME Does this provide us with the correct semantics? If we do it as described, a user program can create a new binding for "Object" that shadows our "Object". This is not a problem for the built-in; it may or may not be a benefit to the user program. It may or may not be backwards compatible (what happens if the user program contains `var isNaN` -- does this redundantly state that there is a binding for `isNaN` or does it create a new binding?)

```
package ...
{
    use default namespace public;
    use namespace intrinsic;

    // All global definitions, see section <XREF target="global-object">
}
```

2 Assumptions and notational conveniences

1 (This section will be removed eventually.)

2 The following assumptions are made throughout the description of the builtins. I believe they are correct for the language, but they need to be specified / cleaned up elsewhere; some of the descriptions here need to be merged into the foregoing sections.

2.1 Classes

1 Classes are reified as singleton class objects `C` which behave like ECMAScript objects in all respects. We do *not* assume here that these class objects are instances of yet other classes; they can be assumed just to exist. Class objects have some set of fixtures (always including the `prototype` property) and a `[[Prototype]]` chain, at a minimum.

2 The `Function` prototype object is on the `[[Prototype]]` chain of every class object, whether native or user defined. This was true for all constuctor functions in ES3; it does not seem reasonable to be incompatible for native objects in ES4, and it does not seem reasonable to have a special case for native objects in ES4 (though that would be possible).

3 *Consequence:* It will be assumed that the `Function` prototype object is on the prototype chain of every class object, and this will not be described explicitly for each object, unlike 3rd Edition.

2.2 Prototype chains

1 Every class object `C` has a constant `C.prototype` fixture property, with fixed type `Object`. Unless specified otherwise, `C.prototype` references an object `PC` that appears to be an instance of `C` except for the value of `PC.[[Prototype]]`, which is normally a reference to `B.prototype` where `B` is the base class of `C`. (Thus the prototype hierarchy mirrors the class hierarchy, and inheritance of prototype properties mirrors the inheritance of class properties.)

2 *Consequence:* It will be assumed that every class object has a `prototype` property and that that property will reference the prototype object for that class, which is always described separately. The fact that there is a `prototype` property will not be described explicitly for each object, unlike 3rd Edition.

3 Every `[[Prototype]]` property of an object `O` of class described by class object `C`, unless specified otherwise, is initialized from the value of `C.prototype`.

4 *Consequence:* The structure of the prototype chain is elided from the description of the native classes except where it diverges from the standard behavior.

2.3 Constant-initialized properties

1 Several properties on both class objects and prototype objects are initialized by references to constants, for example `length` properties on class objects and `constructor` properties on prototype objects. These properties are trivially described in the synopsis and normally do not get a separate section in the body of the class description.

2 As far as `constructor` is concerned, it is a standard feature of the prototype object and its initial value is always the class object, so it does not have to be described either. So it isn't.

3 The Global Object

```
FILE:                        spec/library/global.html
DRAFT STATUS:                DRAFT 1 - ROUGH
REVIEWED AGAINST ES3:        NO
REVIEWED AGAINST PROPOSALS: NO
REVIEWED AGAINST CODE:       NO
```

1 The global object does not have a `[[Construct]]` property; it is not possible to use the global object as a constructor with the new operator.

2 The global object does not have a `[[Call]]` property; it is not possible to invoke the global object as a function. The values of the `[[Prototype]]` and `[[Class]]` properties of the global object are implementation-dependent.

3.1 Synopsis

1 The global object contains the following properties, functions, types, and class definitions.

```
namespace __ES4__

class Object …
class Function …
class Array …
class String …
class Boolean …
class Number …
class Date …
class RegExp …
class Error …
class EvalError …
class RangeError …
class ReferenceError …
class SyntaxError …
class TypeError …
class URIError …

__ES4__ class string …
__ES4__ class boolean
__ES4__ class int …
__ES4__ class uint …
__ES4__ class double …
__ES4__ class decimal …
__ES4__ class Name …
__ES4__ class Namespace …
__ES4__ class ByteArray …
__ES4__ class Map …
__ES4__ class IdentityMap …

__ES4__ interface ObjectIdentity …

__ES4__ type EnumerableId = …
__ES4__ type Numeric = …

intrinsic interface Field …
intrinsic interface FieldValue …
intrinsic interface Type …
intrinsic interface NominalType …
intrinsic interface InterfaceType …
intrinsic interface ClassType …
intrinsic interface UnionType …
intrinsic interface RecordType …
intrinsic interface FunctionType …
intrinsic interface ArrayType …

intrinsic type FieldIterator = …
intrinsic type FieldValueIterator = …
intrinsic type TypeIterator = …
intrinsic type InterfaceIterator = …

const NaN: double = …
const Infinity: double = …
const undefined: undefined = …
const __ECMASCRIPT_VERSION__ = …
const Math: Object = …

__ES4__ const global: Object = …

intrinsic function eval(s: string) …
intrinsic function parseInt(s: string, r: (int,undefined)=undefined): Numeric …
intrinsic function parseFloat(s: string): Numeric …
intrinsic function isNaN(n: Numeric): boolean …
intrinsic function isFinite(n: Numeric): boolean …
intrinsic function decodeURI(s: string): string …
intrinsic function decodeURIComponent(s: string): string …
intrinsic function encodeURI(s: string): string …
```

```
intrinsic function encodeURIComponent(s: string): string …
intrinsic function hashcode(x): uint …

intrinsic function +(a,b) …
intrinsic function -(a,b) …
intrinsic function *(a,b) …
intrinsic function /(a,b) …
intrinsic function %(a,b) …
intrinsic function ^(a,b) …
intrinsic function &(a,b) …
intrinsic function |(a,b) …
intrinsic function <<(a,b) …
intrinsic function >>(a,b) …
intrinsic function >>>(a,b) …
intrinsic function ===(a,b) …
intrinsic function !==(a,b) …
intrinsic function ==(a,b) …
intrinsic function !=(a,b) …
intrinsic function <(a,b) …
intrinsic function <=(a,b) …
intrinsic function >(a,b) …
intrinsic function >=(a,b) …
intrinsic function ~(a) …

function eval(x) …
function parseInt(s, r=undefined) …
function parseFloat(s) …
function isNaN(x) …
function isFinite(x) …
function decodeURI(x) …
function decodeURIComponent(x) …
function encodeURI(x) …
function encodeURIComponent(x) …

__ES4__ function hashcode(x) …
```

3.2 Namespace for types

1 All new classes and type definitions in the global object are defined in the namespace `types`. This namespace is automatically opened by the implementation for code that is to be treated as 4th Edition code, but not for code that is to be treated as 3rd Edition code.

NOTE The risk of polluting the name space for 3rd Edition code with new names is deemed too great to always open the `types` name space.

FIXME The name and behavior of this namespace has yet to be fully resolved by the committee.

2 The means by which an implementation determines whether to treat code according to 3rd Edition or 4th Edition is outside the scope of this Standard.

NOTE This standard makes recommendations for how mime types should be used to tag script content in a web browser. See appendix-mime-types.

3.3 Value Properties of the Global Object

3.3.1 NaN

1 The value of `NaN` is **NaN** (section 8.5).

COMPATIBILITY NOTE This property was not marked ReadOnly in 3rd Edition.

3.3.2 Infinity

1 The value of `Infinity` is +∞ (section 8.5).

COMPATIBILITY NOTE This property was not marked ReadOnly in 3rd Edition.

3.3.3 undefined

1 The value of undefined is **undefined** (section 8.1).

COMPATIBILITY NOTE This property was not marked ReadOnly in 3rd Edition.

3.3.4 __ECMASCRIPT_VERSION__

1 The value of `__ECMASCRIPT_VERSION__` is the version of this Standard to which the implementation conforms. For this 4th Edition of the Standard, the value of `__ECMASCRIPT_VERSION__` is 4.

NOTE This property is new in 4th Edition.

3.4 Function Properties of the Global Object

3.4.1 eval

3.4.1.1 The `eval` operator

1 When the intrinsic and non-intrinsic `eval` functions are called directly by name (that is, by the explicit use of the name `eval` as an Identifier which is the MemberExpression in a CallExpression) they are treated like operators in the language. See eval-operator.

FIXME It's possible we want just the unqualified use of `eval` here.

3.4.1.2 intrinsic::eval (s)

Description

1 When the intrinsic `eval` function is called as a methods on the global objects in whose scope it is closed then it evaluates its argument as a program in the global scope that is the receiver object in the call.

2 When the intrinsic `eval` function is called as an ordinary function under other names than `eval` then it evaluates its argument as a program in a global scope that is the scope in which the `eval` function was closed.

Returns

3 The intrinsic `eval` function returns the value computed by the program that's evaluated.

Implementation

4 The definitions for the two cases described above can be summarized as follows, where the call to `eval` in the body is an instance of the former ("operator") case:

```
intrinsic function eval(s: string)
    eval(s);
```

FIXME That's not right, because `s` shadows any global `s` that should be visible to the evaluated program.

3.4.1.3 eval (s)

Description

1 The non-intrinsic `eval` function can be called as a method on the global object in whose scope it is closed, or it can be called as an ordinary function under another name, just like the intrinsic `eval` function.

2 If the argument to `eval` is a `String` object, then the program represented by that string is evaluated. Otherwise the argument is returned unchanged.

```
function eval(x) {
    if (!(x is String))
        return x;
    return intrinsic::eval(string(x));
}
```

NOTE The behavior of this function depends on the fact that the non-intrinsic `eval` function is closed in the same global object as the intrinsic `eval` function. Thus there's no need to capture and pass the `this` parameter.

3.4.1.4 Restrictions on the use of the `eval` property

1 If the value of the `eval` property is used in any way other than than the three listed previously, or if the `eval` property is assigned to, an `EvalError` exception may be thrown.

COMPATIBILITY NOTE The 3rd Edition of this Standard restricted the use of `eval` to the first case listed previously.

3.4.2 intrinsic::parseInt (s, r=...)

Description

1 The intrinsic `parseInt` function computes an integer value dictated by interpretation of the contents of the string argument *s* according to the specified radix *r* (which defaults to zero). Leading whitespace in *s* is ignored. If *r* is zero, it is assumed to be 10 except when the number begins with the character pairs

0x or 0X, in which case a radix of 16 is assumed. Any radix-16 number may also optionally begin with the character pairs 0x or 0X.

Returns

2 The intrinsic `parseInt` function returns a number.

Implementation

```
intrinsic function parseInt(s: string, r: int=0): Numeric {
    let i;

    for ( i=0 ; i < s.length && Unicode.isTrimmableSpace(s[i]) ; i++ )
        ;
    s = s.substring(i);

    let sign = 1;
    if (s.length >= 1 && s[0] == '-')
        sign = -1;
    if (s.length >= 1 && (s[0] == '-' || s[0] == '+'))
        s = s.substring(1);

    let maybe_hexadecimal = false;
    if (r == 0) {
        r = 10;
        maybe_hexadecimal = true;
    }
    else if (r == 16)
        maybe_hexadecimal = true;
    else if (r < 2 || r > 36)
        return NaN;

    if (maybe_hexadecimal && s.length >= 2 && s[0] == '0' && (s[1] == 'x' || s[1] == 'X')) {
        r = 16;
        s = s.substring(2);
    }

    for ( i=0 ; i < s.length && helper::isDigitForRadix(s[i], r) ; i++ )
        ;
    s = s.substring(0,i);

    if (s == "")
        return NaN;

    return sign * informative::numericValue(s, r);
}
```

3 The helper function `isDigitForRadix(c,r)` computes whether `c` is a valid digit for the radix `r`, see helper:isDigitForRadix.

4 The informative function `numericValue(s, r)` computes the numeric value of a radix-`r` string `s`. If `r` is 10 and `s` contains more than 20 significant digits, every significant digit after the 20th may be replaced by a 0 digit, at the option of the implementation; and if `r` is not 2, 4, 8, 10, 16, or 32, then the returned value may be an implementation-dependent approximation to the mathematical integer value that is represented by `s` in radix-`r` notation.

COMPATIBILITY NOTE In the 3rd Edition of this Standard, the `parseInt` function was allowed to, though not encouraged to, interpret a string with a leading `0` but no leading `0x` or `0X` as a base-8 number if the radix was not supplied in the call or was supplied as zero. This is no longer allowed; the function must interpret such a number as a base-10 number.

NOTE `parseInt` may interpret only a leading portion of the string as an integer value; it ignores any characters that cannot be interpreted as part of the notation of an integer, and no indication is given that any such characters were ignored.

3.4.2.1 isDigitForRadix

```
helper function isDigitForRadix(c, r) {
    c = c.toUpperCase();
    if (c >= '0' && c <= '9')
        return (c.charCodeAt(0) - '0'.charCodeAt(0)) < r;
    else if (c >= 'A' && c <= 'Z')
        return (c.charCodeAt(0) - 'A'.charCodeAt(0) + 10) < r;
    else
        return false;
}
```

3.4.3 parseInt (s, r=...)

Description

1 The `parseInt` function converts its first argument to `string` and its second argument to `int`, and then calls its intrinsic counterpart.

Returns

2 The `parseInt` function returns a number.

Implementation

```
function parseInt(s, r=0)
    intrinsic::parseInt(string(s), int(r));
```

3.4.4 intrinsic::parseFloat (s)

Description

1 The intrinsic `parseFloat` function computes a number value dictated by interpretation of the contents of the string argument *s* as a decimal literal.

Returns

2 The intrinsic `parseFloat` function returns a number.

Implementation

```
intrinsic function parseFloat(s: string) {
}
```

NOTE `parseFloat` may interpret only a leading portion of *s* as a number value; it ignores any characters that cannot be interpreted as part of the notation of an decimal literal, and no indication is given that any such characters were ignored.

3.4.5 parseFloat (s)

Description

1 The `parseFloat` function converts its argument to `string`, then calls its intrinsic counterpart.

Returns

2 The `parseFloat` function returns a number.

Implementation

```
function parseFloat(s)
    intrinsic::parseFloat(string(s));
```

3.4.6 intrinsic::isNaN (number)

Description

1 The intrinsic `isNaN` function tests whether a numeric value *number* is an IEEE not-a-number value.

Returns

2 The intrinsic `isNaN` function returns **true** if *number* is **NaN**, and otherwise returns **false**.

Implementation

```
intrinsic function isNaN(n: Numeric): boolean
    (!(n === n));
```

3.4.7 isNaN (number)

Description

1 The `isNaN` function converts its argument to a number, then calls its intrinsic counterpart.

Returns

2 The `isNaN` function returns **true** if *number* is **NaN**, and otherwise returns **false**.

Implementation

```
function isNaN(number)
    intrinsic::isNaN(ToNumeric(number));
```

3.4.8 intrinsic::isFinite (number)

Description

1 The intrinsic `isFinite` function tests whether a numeric value *number* is finite (neither not-a-number nor an infinity).

Returns

2 The intrinsic `isFinite` function returns **true** if *number* is finite, and otherwise returns **false**.

Implementation

```
intrinsic function isFinite(n: Numeric): boolean
    !isNaN(n) && n != -Infinity && n != Infinity;
```

3.4.9 isFinite (number)

Description

1 The `isFinite` function converts its argument to a number, then calls its intrinsic counterpart.

Returns

2 The `isFinite` function returns **true** if *number* is finite, and otherwise returns **false**.

Implementation

```
function isFinite(x)
    intrinsic::isFinite(ToNumeric(x));
```

3.4.10 URI Handling Function Properties

1 Uniform Resource Identifiers, or URIs, are strings that identify resources (e.g. web pages or files) and transport protocols by which to access them (e.g. HTTP or FTP) on the Internet. The ECMAScript language itself does not provide any support for using URIs except for functions that encode and decode URIs as described in sections decodeURI, decodeURIComponent, encodeURI, and encodeURIComponent.

NOTE Many implementations of ECMAScript provide additional functions and methods that manipulate web pages; these functions are beyond the scope of this standard.

2 A URI is composed of a sequence of components separated by component separators. The general form is:

Scheme `:` *First* `/` *Second* `;` *Third* `?` *Fourth*

3 where the italicised names represent components and the ":", "/", ";" and "?" are reserved characters used as separators. The `encodeURI` and `decodeURI` functions are intended to work with complete URIs; they assume that any reserved characters in the URI are intended to have special meaning and so are not encoded. The `encodeURIComponent` and `decodeURIComponent` functions are intended to work with the individual component parts of a URI; they assume that any reserved characters represent text and so must be encoded so that they are not interpreted as reserved characters when the component is part of a complete URI. The following lexical grammar specifies the form of encoded URIs.

uri :::
    uriCharacters$_{opt}$

uriCharacters :::
    uriCharacter uriCharacters$_{opt}$

uriCharacter :::
    uriReserved
    uriUnescaped
    uriEscaped

uriReserved ::: ***one of***
    `; / ? : @ & = + $ ,`

uriUnescaped :::
    uriAlpha
    DecimalDigit
    uriMark

uriEscaped :::
    % HexDigit HexDigit

uriAlpha ::: ***one of***
    `a b c d e f g h i j k l m n o p q r s t u v w x y z`
    `A B C D E F G H I J K L M N O P Q R S T U V W X Y Z`

uriMark ::: ***one of***

```
- _ . ! ~ * ' ( )
```

FIXME Upgrade to Unicode 5 in the following sections, and upgrade to handling the entire Unicode character set.

4 When a character to be included in a URI is not listed above or is not intended to have the special meaning sometimes given to the reserved characters, that character must be encoded. The character is first transformed into a sequence of octets using the UTF-8 transformation, with surrogate pairs first transformed from their UCS-2 to UCS-4 encodings. (Note that for code points in the range [0,127] this results in a single octet with the same value.) The resulting sequence of octets is then transformed into a string with each octet represented by an escape sequence of the form "`%xx`".

5 The encoding and escaping process is described by the helper function `encode` taking two string arguments `s` and `unescapedSet`.

```
helper function encode(s: string, unescapedSet: string): string {
    let R = "";
    let k = 0;

    while (k != s.length) {
        let C = s[k];

        if (unescapedSet.indexOf(C) != 1) {
            R = R + C;
            k = k + 1;
            continue;
        }

        let V = C.charCodeAt(0);
        if (V >= 0xDC00 && V <= 0xDFFF)
            throw new URIError("Invalid code");
        if (V >= 0xD800 && V <= 0xDBFF) {
            k = k + 1;
            if (k == s.length)
                throw new URIError("Truncated code");
            let V2 = s[k].charCodeAt(0);
            V = (V - 0xD800) * 0x400 + (V2 - 0xDC00) + 0x10000;
        }

        let octets = helper::toUTF8(V);
        for ( let j=0 ; j < octets.length ; j++ )
            R = R + "%" + helper::twoHexDigits(octets[j]);
        k = k + 1;
    }
    return R;
}

helper function twoHexDigits(B) {
    let s = "0123456789ABCDEF";
    return s[B >> 4] + s[B & 15];
}
```

6 The unescaping and decoding process is described by the helper function `decode` taking two string arguments `s` and `reservedSet`.

FIXME One feels regular expressions would be appropriate here...

```
helper function decode(s: string, reservedSet: string): string {
    let R = "";
    let k = 0;
    while (k != s.length) {
        if (s[k] != "%") {
            R = R + s[k];
            k = k + 1;
            continue;
        }

        let start = k;
        let B = helper::decodeHexEscape(s, k);
        k = k + 3;

        if ((B & 0x80) == 0) {
            let C = string.fromCharCode(B);
            if (reservedSet.indexOf(C) != -1)
                R = R + s.substring(start, k);
            else
                R = R + C;
            continue;
        }

        let n = 1;
        while (((B << n) & 0x80) == 1)
            ++n;
        if (n == 1 || n > 4)
            throw new URIError("Invalid encoded character");

        let octets = [B];
```

```
        for ( let j=1 ; j < n ; ++j ) {
            let B = helper::decodeHexEscape(s, k);
            if ((B & 0xC0) != 0x80)
                throw new URIError("Invalid encoded character");
            k = k + 3;
            octets.push(B);
        }
        let V = helper::fromUTF8(octets);
        if (V > 0x10FFFF)
            throw new URIError("Invalid Unicode code point");
        if (V > 0xFFFF) {
            L = ((V - 0x10000) & 0x3FF) + 0xD800;
            H = (((V - 0x10000) >> 10) & 0x3FF) + 0xD800;
            R = R + string.fromCharCode(H, L);
        }
        else {
            let C = string.fromCharCode(V);
            if (reservedSet.indexOf(C))
                R = R + s.substring(start, k);
            else
                R = R + C;
        }
    }
    return R;
}

helper function decodeHexEscape(s, k) {
    if (k + 2 >= s.length ||
        s[k] != "%" ||
        !helper::isDigitForRadix(s[k+1], 16) && !helper::isDigitForRadix(s[k+1], 16))
        throw new URIError("Invalid escape sequence");
    return parseInt(s.substring(k+1, k+3), 16);
}
```

7 The helper function `isDigitForRadix` was defined in section helper:isDigitForRadix.

NOTE The syntax of Uniform Resource Identifiers is given in RFC2396.

NOTE A formal description and implementation of UTF-8 is given in the Unicode Standard, Version 2.0, Appendix A. In UTF-8, characters are encoded using sequences of 1 to 6 octets. The only octet of a "sequence" of one has the higher-order bit set to 0, the remaining 7 bits being used to encode the character value. In a sequence of n octets, n>1, the initial octet has the n higher-order bits set to 1, followed by a bit set to 0. The remaining bits of that octet contain bits from the value of the character to be encoded. The following octets all have the higher-order bit set to 1 and the following bit set to 0, leaving 6 bits in each to contain bits from the character to be encoded. The possible UTF-8 encodings of ECMAScript characters are:

Code Point Value	Representation	1st Octet	2nd Octet	3rd Octet	4th Octet
0x0000 - 0x007F	00000000 0zzzzzzz	0zzzzzzz			
0x0080 - 0x07FF	00000yyy yyzzzzzz	110yyyyy	10zzzzzz		
0x0800 - 0xD7FF	xxxxyyyy yyzzzzzz	1110xxxx	10yyyyyy	10zzzzzz	
0xD800 - 0xDBFF followed by 0xDC00 - 0xDFFF	110110vv vvwwwwxx followed by 110111yy yyzzzzzz	11110uuu	10uuwwww	10xxyyyy	10zzzzzz
0xD800 - 0xDBFF not followed by 0xDC00 - 0xDFFF	causes URIError				
0xDC00 - 0xDFFF	causes URIError				
0xE000 - 0xFFFF	xxxxyyyy yyzzzzzz	1110xxxx	10yyyyyy	10zzzzzz	

8 Where

uuuuu = vvvv + 1

9 to account for the addition of 0x10000 as in section 3.7, Surrogates of the Unicode Standard version 2.0.

10 The range of code point values 0xD800-0xDFFF is used to encode surrogate pairs; the above transformation combines a UCS-2 surrogate pair into a UCS-4 representation and encodes the resulting 21-bit value in UTF-8. Decoding reconstructs the surrogate pair.

11 The helper functions `encode` and `decode`, defined above, use the helper functions `toUTF8` and `fromUTF8` to convert code points to UTF-8 sequences and to convert UTF-8 sequences to code points, respectively.

```
helper function toUTF8(v: uint) {
    if (v <= 0x7F)
        return [v];
    if (v <= 0x7FF)
        return [0xC0 | ((v >> 6) & 0x3F),
                0x80 | (v & 0x3F)];
    if (v <= 0xD7FF | v >= 0xE000 && v <= 0xFFFF)
        return [0xE0 | ((v >> 12) & 0x0F),
                0x80 | ((v >> 6) & 0x3F),
                0x80 | (v & 0x3F)];
    if (v >= 0x10000)
        return [0xF0 | ((v >> 18) & 0x07),
                0x80 | ((v >> 12) & 0x3F),
                0x80 | ((v >> 6) & 0x3F),
                0x80 | (v & 0x3F)];
    throw URIError("Unconvertable code");
}

helper function fromUTF8(octets) {
    let B = octets[0];
    let V;
    if ((B & 0x80) == 0)
        V = B;
    else if ((B & 0xE0) == 0xC0)
        V = B & 0x1F;
    else if ((B & 0xF0) == 0xE0)
        V = B & 0x0F;
    else if ((B & 0xF8) == 0xF0)
        V = B & 0x07;
    for ( let j=1 ; j < octets.length ; j++ )
        V = (V << 6) | (octets[j] & 0x3F);
    return V;
}
```

12 Several helper strings are defined based on the grammar shown previously:

```
helper const uriReserved = ";/?:@&=+$,";

helper const uriAlpha = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ";

helper const uriDigit = "0123456789";

helper const uriMark = "-_.!~*'()";

helper const uriUnescaped = helper::uriAlpha + helper::uriDigit + helper::uriMark;
```

3.4.10.1 intrinsic::decodeURI (encodedURI)

Description

1 The intrinsic `decodeURI` function computes a new version of a URI in which each escape sequence and UTF-8 encoding of the sort that might be introduced by the `encodeURI` function is replaced with the character that it represents. Escape sequences that could not have been introduced by `encodeURI` are not replaced.

Returns

2 The intrinsic `decodeURI` function returns a decoded string.

Implementation

```
intrinsic function decodeURI(encodedURI: string)
    helper::decode(encodedURI, helper::uriReserved + "#");
```

NOTE The character "#" is not decoded from escape sequences even though it is not a reserved URI character.

3.4.10.2 decodeURI (encodedURI)

Description

1 The `decodeURI` function converts its argument to `string`, then calls its intrinsic counterpart.

Returns

2 The `decodeURI` function returns a decoded string.

Implementation

```
function decodeURI(encodedURI)
    intrinsic::decodeURI(string(encodedURI));
```

3.4.10.3 intrinsic::decodeURIComponent (encodedURIComponent)

Description

1 The intrinsic `decodeURIComponent` function computes a new version of a URI in which each escape sequence and UTF-8 encoding of the sort that might be introduced by the `encodeURIComponent` function is replaced with the character that it represents.

Returns

2 The intrinsic `decodeURIComponent` function returns a decoded string.

Implementation

```
intrinsic function decodeURIComponent(encodedURIComponent)
    helper::decode(encodedURIComponent, "");
```

3.4.10.4 decodeURIComponent (encodedURIComponent)

Description

1 The `decodeURIComponent` function converts its argument to `string`, then calls its intrinsic counterpart.

Returns

2 The `decodeURIComponent` function returns a decoded string.

Implementation

```
function decodeURIComponent(encodedURIComponent)
    intrinsic::decodeURIComponent(string(encodedURIComponent));
```

3.4.10.5 intrinsic::encodeURI (uri)

Description

1 The intrinsic `encodeURI` function computes a new version of a URI in which each instance of certain characters is replaced by one, two or three escape sequences representing the UTF-8 encoding of the character.

Returns

2 The intrinsic `encodeURI` function returns a encoded string.

Implementation

```
intrinsic function encodeURI(uri: string): string
    helper::encode(uri, helper::uriReserved + helper::uriUnescaped + "#")
```

NOTE The character "#" is not encoded to an escape sequence even though it is not a reserved or unescaped URI character.

3.4.10.6 encodeURI (uri)

Description

1 The `encodeURI` function converts its argument to string, then calls its intrinsic counterpart.

Returns

2 The `encodeURI` function returns a encoded string.

Implementation

```
function encodeURI(uri)
    intrinsic::encodeURI(string(uri));
```

3.4.10.7 intrinsic::encodeURIComponent (uriComponent)

Description

1 The intrinsic `encodeURIComponent` function computes a new version of a URI in which each instance of certain characters is replaced by one, two or three escape sequences representing the UTF-8 encoding of the character.

Returns

2 The intrinsic `encodeURIComponent` function returns a encoded string.

Implementation

```
intrinsic function encodeURIComponent(uriComponent: string): string
    helper::encode(uri, helper::uriReserved);
```

3.4.10.8 encodeURIComponent (uriComponent)

Description

1 The `encodeURIComponent` function converts its argument to string, then calls its intrinsic counterpart.

Returns

2 The `encodeURIComponent` function returns a encoded string.

Implementation

```
function encodeURIComponent(uriComponent)
    intrinsic::encodeURIComponent(string(uriComponent));
```

3.4.11 intrinsic::hashcode (x)

Description

1 The intrinsic `hashcode` function computes a numeric value for its argument such that if two values `v1` and `v2` are equal by the operator `intrinsic::===` then `hashcode(v1)` is numerically equal to `hashcode(v2)`.

2 The hashcode of any value for which `isNaN` returns **true** is zero.

3 The hashcode computed for an object does not change over time.

Returns

4 The intrinsic `hashcode` function returns an unsigned integer.

Implementation

```
intrinsic function hashcode(o): uint {
    switch type (o) {
    case (x: null)      { return 0u }
    case (x: undefined) { return 0u }
    case (x: boolean)   { return uint(x) }
    case (x: int)       { return x < 0 ? -x : x }
    case (x: uint)      { return x }
    case (x: double)    { return isNaN(x) ? 0u : uint(x) }
    case (x: decimal)   { return isNaN(x) ? 0u : uint(x) }
    case (x: String)    { return informative::stringHash(string(x)) }
    case (x: *)         { return informative::objectHash(x) }
    }
}
```

5 The informative functions `stringHash` and `objectHash` compute hash values for strings and arbitrary objects, respectively. They can take into account their arguments' immutable structure only.

6 The implementation should strive to compute different hashcodes for objects that are not the same by `intrinsic::===`, as the utility of this function depends on that property. (The user program should be able to expect that the hashcodes of objects that are not the same are different with high probability.)

NOTE A typical implementation of `stringHash` will make use of the string's character sequence and its length.

NOTE A typical implementation of `objectHash` may make use of the object's address in memory if the object, or it may maintain a separate table mapping objects to hash codes.

IMPLEMENTATION NOTE The intrinsic `hashcode` function should not return pointer values cast to integers, even in implementations that do not use a moving garbage collector. Exposing memory locations of objects may make security vulnerabilities in the host environment significantly worse. Implementations -- in particular those which read network input -- should return numbers unrelated to memory addresses if possible, or at least use memory addresses subject to some cryptographically strong one-way transformation, or sequence numbers, cookies, or similar.

3.4.12 Operator functions

FIXME These are defined as implementing the primitive functionality of each operator, bypassing any user overloading. They can be referenced by prefixing them with a namespace, eg `intrinsic::===`.

3.5 Class and Interface Properties of the Global Object

1 The class properties of the global object are defined in later sections of this Standard:

- The `Object` class is defined in section class Object
- The `Function` class is defined in section class Function
- The `Name` class is defined in section class Name
- The `Namespace` class is defined in section class Namespace

- The `Array` class is defined in section class Array
- The `ByteArray` class is defined in section class ByteArray
- The `String` and `string` classes are defined in sections class String and class string, respectively.
- The `Boolean` and `boolean` classes are defined in sections class Boolean and class boolean, respectively.
- The `Number`, `int`, `uint`, `double`, and `decimal` classes are defined in sections class Number, class int, class uint, class double, and class decimal, respectively.
- The `Date` class is defined in section class Date
- The `RegExp` class is defined in section class RegExp
- The `Map` class is defined in section class Map
- The `Error` class and its subclasses `EvalError`, `RangeError`, `ReferenceError`, `SyntaxError`, `TypeError`, and `URIError` are defined in sections class Error, class EvalError, class RangeError, class ReferenceError, class SyntaxError, class TypeError, and class URIError, respectively.

3.6 Type Properties on the Global Object

3.6.1 EnumerableId

1 The type `EnumerableId` is a union type that collects all nominal types that are treated as property names by the iteration protocol and the built-in objects:

```
__ES4__ type EnumerableId = (int,uint,Name,string);
```

3.6.2 Numeric

1 The type `Numeric` is a union type that collects all nominal types that are treated as numbers by the implementation:

```
__ES4__ type Numeric = (int,uint,double,decimal,Number);
```

3.7 Meta-Object Interface and Type Properties of the Global Object

1 The interface types `Field`, `FieldValue`, `Type`, `NominalType`, `InterfaceType`, `ClassType`, `UnionType`, `RecordType`, `FunctionType`, and `ArrayType`, as well as the structural types `FieldIterator`, `FieldValueIterator`, `TypeIterator`, and `InterfaceIterator`, are defined in section meta-objects.

3.8 Other Properties of the Global Object

3.8.1 Math

1 See section math-object.

3.8.2 global

1 The intrinsic `global` property holds a reference to the global object that contains that property.

NOTE There may be multiple global objects in a program, but these objects may share values or immutable state: for example, their `isNaN` properties may hold the same function object. However, each global object has separate mutable state, and a separate value for the intrinsic `global` property.

NOTE This property is new in 4th Edition.

4 The class `Object`

```
FILE:                        spec/library/Object.html
DRAFT STATUS:                DRAFT 1 - ROUGH
REVIEWED AGAINST ES3:        NO
REVIEWED AGAINST PROPOSALS:  NO
REVIEWED AGAINST CODE:       NO
```

1 The class `Object` is a dynamic non-final class that does not subclass any other objects: it is the root of the class hierarchy.

2 All values in ECMAScript except **undefined** and **null** are instances of the class `Object` or one of its subclasses.

> **NOTE** Host objects may not be instances of `Object` or its subclasses, but must to some extent behave as if they are (see Host objects).

4.1 Synopsis

1 The class `Object` provides this interface:

```
dynamic class Object
{
    function Object(value=undefined) …
    meta static function invoke(value=undefined) …

    static const length = 1

    intrinsic function toString() : string …
    intrinsic function toLocaleString() : string …
    intrinsic function toJSONString() : string …
    intrinsic function valueOf() : Object! …
    intrinsic function hasOwnProperty(V: EnumerableId): boolean …
    intrinsic function isPrototypeOf(V): boolean …
    intrinsic function propertyIsEnumerable(prop: EnumerableId, …
}
```

2 The `Object` prototype object provides these direct properties:

```
    toString:             function () … ,
    toLocaleString:       function () … ,
    toJSONString:         function () … ,
    valueOf:              function () … ,
    hasOwnProperty:       function (V) … ,
    isPrototypeOf:        function (V) … ,
    propertyIsEnumerable: function (name, flag=undefined) … ,
```

3 The `Object` prototype object is itself an instance of the class `Object`, with the exception that the value of its `[[Prototype]]` property is **null**.

4.2 Methods on the `Object` class object

4.2.1 new Object (value=…)

Description

1 When the `Object` constructor is called with an argument *value* (defaulting to **undefined**) as part of a `new` expression, it transforms the *value* to an object in a way that depends on the type of *value*.

Returns

2 The `Object` constructor returns an object (an instance of `Object` or one of its subclasses, or a host object).

> **NOTE** The `Object` constructor is the only constructor function defined on a class in the language whose result may be a value of a different class than the one in which the constructor is defined.

Implementation

3 The `Object` constructor can't be expressed as a regular ECMAScript constructor. Instead it is presented below as a function `makeObject` that the ECMAScript implementation will invoke when it evaluates `new Object`.

4 The function `makeObject` is only invoked on native ECMAScript values. If `new Object` is evaluated on a host object, then actions are taken and a result is returned in an implementation dependent manner that may depend on the host object.

```
function makeObject(value=undefined) {
    switch type (value) {
    case (s:string) {
        return new String(s);
    }
    case (b:boolean) {
        return new Boolean(b);
    }
    case (n:(int,uint,double,decimal)) {
        return new Number(n);
    }
    case (o:Object) {
        return o;
    }
    case (x:(null,undefined)) {
        return magic::createObject();
    }
    }
}
```

4.2.2 Object (value=...)

Description

1 When the `Object` class object is called as a function with zero or one arguments it performs a type conversion.

Returns

2 It returns the converted value.

Implementation

```
meta static function invoke(value=undefined) {
    if (value === null || value === undefined)
        return new Object();
    return ToObject(value);
}
```

4.3 Methods on `Object` instances

4.3.1 intrinsic::toString ( )

Description

1 The intrinsic `toString` method converts the `this` object to a string.

Returns

2 The intrinsic `toString` method returns the concatenation of `"["`, `"Object"`, the class name of the object, and `"]"`.

Implementation

```
intrinsic function toString() : string
    "[object " + magic::getClassName(this) + "]";
```

3 The function `magic::getClassName` extracts the `[[Class]]` property from the object. See magic:getClassName.

4.3.2 intrinsic::toLocaleString ( )

Description

1 The intrinsic `toLocaleString` method calls the public `toString` method on the `this` object.

NOTE This method is provided to give all objects a generic `toLocaleString` interface, even though not all may use it. Currently, `Array`, `Number`, and `Date` provide their own locale-sensitive `toLocaleString` methods.

NOTE The first parameter to this function is likely to be used in a future version of this standard; it is recommended that implementations do not use this parameter position for anything else.

Returns

2 The intrinsic `toLocaleString` method returns a string.

Implementation

```
intrinsic function toLocaleString() : string
    this.toString();
```

4.3.3 intrinsic::toJSONString ( )

FIXME Waiting for proposal to be cleaned up and the RI method to be implemented.

4.3.4 intrinsic::valueOf ( )

Description

1 The intrinsic `valueOf` method returns its `this` value.

2 If the object is the result of calling the Object constructor with a host object (Host objects), it is implementation-defined whether `valueOf` returns its `this` value or another value such as the host object originally passed to the constructor.

Returns

3 The intrinsic `toLocaleString` method returns an object value.

Implementation

```
intrinsic function valueOf() : Object!
    this;
```

4.3.5 intrinsic::hasOwnProperty (name)

Description

1 The intrinsic `hasOwnProperty` method determines whether the `this` object contains a property with a certain *name*, without considering the prototype chain.

NOTE Unlike `[[HasProperty]]` (HasProperty-defn), this method does not consider objects in the prototype chain.

Returns

2 The intrinsic `hasOwnProperty` method returns `true` if the object contains the property, otherwise it returns `false`.

Implementation

```
intrinsic function hasOwnProperty(V: EnumerableId): boolean
    magic::hasOwnProperty(this, V);
```

3 The function `magic::hasOwnProperty` tests whether the object contains the named property on its local property list (the prototype chain is not considered). See magic:hasOwnProperty.

4.3.6 intrinsic::isPrototypeOf (obj)

Description

1 The intrinsic `isPrototypeOf` method determines whether its `this` object is a prototype object of the argument *obj*.

Returns

2 The intrinsic `isPrototypeOf` method returns `true` if the `this` object is on the prototype chain of *obj*, otherwise it returns `false`.

Implementation

```
intrinsic function isPrototypeOf(V): boolean {
    if (!(V is Object))
        return false;

    while (true) {
        V = magic::getPrototype(V);
        if (V === null || V === undefined)
            return false;
        if (V === this)
            return true;
    }
}
```

3 The function `magic::getPrototype` extracts the `[[Prototype]]` property from the object. See magic:getPrototype.

4.3.7 intrinsic::propertyIsEnumerable (name, flag=...)

Description

1 The intrinsic `propertyIsEnumerable` method retrieves, and optionally sets, the enumerability flag for a property with a certain *name* on the `this` object, without considering the prototype chain.

NOTE This method does not consider objects in the prototype chain.

Returns

2 The intrinsic `propertyIsEnumerable` method returns `false` if the property does not exist on the `this` object; otherwise it returns the value of the enumerability flag for the property before any change was made.

Implementation

```
intrinsic function propertyIsEnumerable(prop: EnumerableId,
                                        e:(boolean,undefined) = undefined): boolean
{
    if (!magic::hasOwnProperty(this,prop))
        return false;

    let oldval = !magic::getPropertyIsDontEnum(this, prop);
    if (!magic::getPropertyIsDontDelete(this, prop))
        if (e is boolean)
            magic::setPropertyIsDontEnum(this, prop, !e);
    return oldval;
}
```

3 The function `magic::hasOwnProperty` tests whether the object contains the named property on its local property list. See magic:hasOwnProperty.

4 The function `magic::getPropertyIsDontEnum` gets the DontEnum flag of the property. See magic:getPropertyIsDontEnum.

5 The function `magic::getPropertyIsDontDelete` gets the DontDelete flag of the property. See magic:getPropertyIsDontDelete.

6 The function `magic::setPropertyIsDontEnum` sets the DontEnum flag of the property. See magic:setPropertyIsDontEnum.

4.4 Methods on the `Object` prototype object

Description

1 The methods on the `Object` prototype object all call the corresponding intrinsic methods of the `Object` class.

Returns

2 The prototype methods return what their corresponding intrinsic methods return.

Implementation

```
prototype function toString()
    this.intrinsic::toString();

prototype function toLocaleString()
    this.intrinsic::toLocaleString();

prototype function toJSONString()
    this.intrinsic::toJSONString();

prototype function valueOf()
    this.intrinsic::valueOf();

prototype function hasOwnProperty(V)
    this.intrinsic::hasOwnProperty(V is EnumerableId ? V : string(V));

prototype function isPrototypeOf(V)
    this.intrinsic::isPrototypeOf(V);

prototype function propertyIsEnumerable(prop, e)
    this.intrinsic::propertyIsEnumerable(prop is EnumerableId ? prop : string(prop),
                                         e is (boolean,undefined) ? e : boolean(e));
```

5 The class `Function`

```
FILE:                         spec/library/Function.html
DRAFT STATUS:                 DRAFT 1 - ROUGH
REVIEWED AGAINST ES3:         NO
REVIEWED AGAINST PROPOSALS:   NO
REVIEWED AGAINST CODE:        NO
```

1 The class `Function` is a dynamic non-final subclass of `Object` (see class Object).

2 All objects defined by `function` definitions or expressions in ECMAScript are instances of the class `Function`.

3 Not all objects that can be called as functions are instances of subclasses of the `Function` class, however. Any object that has a `meta invoke` method can be called as a function.

NOTE Host functions may also not be instances of `Function` or its subclasses, but must to some extent behave as if they are (see Host objects).

5.1 Synopsis

1 The class `Function` provides the following interface:

```
dynamic class Function extends Object
{
    function Function(...args) …
    meta static function invoke(...args) …

    static function apply(fn : Function!, thisArg=undefined, argArray=undefined) …
    static function call(fn, thisObj=undefined, ...args:Array):* …

    static const length = 1

    meta final function invoke( … ) …

    override intrinsic function toString() : string …

    intrinsic function apply(thisArg=undefined, argArray=undefined) …
    intrinsic function call(thisObj=undefined, ...args) …
    intrinsic function HasInstance(V) …

    const length = …
    var   prototype = …
}
```

2 The `Function` prototype object provides these direct properties:

```
    toString:    function () … ,
    apply:       function(thisArg, argArray) … ,
    call:        function(thisArg, ...args) … ,
```

5.2 Methods on the `Function` class object

5.2.1 new Function (p1, p2, … , pn, body)

Description

1 When the `Function` constructor is called with some arguments as part of a `new` expression, it creates a new `Function` instance whose parameter list is given by the concatenation of the p_i arguments and whose executable code is given by the *body* argument.

2 There may be no p_i arguments, and *body* is optional too, defaulting to the empty string.

3 If the list of parameters is not parsable as a $FormalParameterList_{opt}$, or if the body is not parsable as a *FunctionBody*, then a **SyntaxError** exception is thrown.

FIXME Cross-reference to grammar here for those production names.

Returns

4 The `Function` constructor returns a new `Function` instance.

Implementation

```
function Function(...args)
    helper::createFunction(args);

helper function createFunction(args) {
    let parameters = "";
```

```
    let body = "";
    if (args.length > 0) {
        body = args[args.length-1];
        args.length = args.length-1;
        parameters = args.join(",");
    }
    body = string(body);
    magic::initializeFunction(this, intrinsic::global, parameters, body);
}
```

5 The helper function `createFunction` is also used by the `invoke` method (see Function: meta static invoke).

6 The magic function `initializeFunction` initializes the function object `this` from the list of parameters and the body, as specified in section translation:FunctionExpression. The global object is passed in as the Scope parameter.

7 A `prototype` property is automatically created for every function, to provide for the possibility that the function will be used as a constructor.

NOTE It is permissible but not necessary to have one argument for each formal parameter to be specified. For example, all three of the following expressions produce the same result:

```
new Function("a", "b", "c", "return a+b+c")

new Function("a, b, c", "return a+b+c")

new Function("a,b", "c", "return a+b+c")
```

FIXME Type annotations? The RI barfs (looks like an incomplete or incorrect set of namespaces is provided during construction).

FIXME Return type annotations? No way to specify this using the current shape of the constructor.

FIXME Default values? The RI says yes.

FIXME Rest arguments? The RI says yes.

FIXME One possibility is to extend the syntax, s.t. the *pi* concatenated can form a syntactically valid parameter list bracketed by `(` and `)`; this creates the possibility that a return type annotation can follow the `)`.

5.2.2 Function (p1, p2, … , pn, body)

Description

1 When the `Function` class object is called as a function it creates and initialises a new `Function` object. Thus the function call `Function(…)` is equivalent to the object creation expression `new Function(…)` with the same arguments.

Returns

2 The `Function` class object called as a function returns a new `Function` instance.

Implementation

```
meta static function invoke(...args)
    helper::createFunction(args)
```

3 The helper function `createFunction` was defined along with the `Function` constructor (see Function: constructor).

5.2.3 apply (fn, thisArg=…, argArray=…)

Description

1 The static `apply` method takes arguments *fn*, *thisArg*, and *argArray*, and performs a function call using the `[[Call]]` property of *fn*, passing *thisArg* as the value for `this` and the members of *argArray* as the individual argument values.

2 If *fn* does not have a `[[Call]]` property, a **TypeError** exception is thrown.

Returns

3 The `apply` method returns the value returned by *fn*.

Implementation

```
static function apply(fn : Function!, thisArg=undefined, argArray=undefined) {
    if (thisArg === undefined || thisArg === null)
        thisArg = global;
    if (argArray === undefined || argArray === null)
```

```
        argArray = [];
    else if (!(argArray is Array))
        throw new TypeError("argument array to 'apply' must be Array");
    return magic::apply(fn, thisArg, argArray);
}
```

4 The magic `apply` function performs the actual invocation (see magic::apply).

5.2.4 call (fn, thisArg=..., ...args)

Description

1 The static `call` method takes arguments *fn* and *thisArg* and optionally some *args*, and performs a function call using the `[[Call]]` property of *fn*, passing *thisArg* as the value for `this` and the members of *args* as the individual argument values.

2 If *fn* does not have a `[[Call]]` property, a **TypeError** exception is thrown.

Returns

3 The `call` method returns the value returned by *fn*.

Implementation

```
static function call(fn, thisObj=undefined, ...args:Array):*
    Function.apply(fn, thisObj, args);
```

5.3 Methods on Function instances

5.3.1 meta::invoke (...)

Description

1 The meta method `invoke` is specialized to the individual function object. When called, it evaluates the executable code for the function.

2 The meta method `invoke` is typically called by the ECMAScript implementation as part of the function invocation and object construction protocols. When a function or method is invoked, the `invoke` method of the function or method object provides the code to run. When a function is used to construct a new object, the `invoke` method provides the code for the constructor function.

3 The signature of the meta method `invoke` is determined when the `Function` instance is created, and is determined by the text that defines the function being created.

> **NOTE** The meta method `invoke` is `final`; therefore subclasses can add properties and methods but can't override the function calling behavior.

> **FIXME** While it is necessary that the `invoke` method is completely magic in `Function` instances, it's not clear it needs to be magic for instances of subclasses of `Function`, because these can be treated like other objects that have `invoke` methods (and which already work just fine). Therefore it should not be `final`.

Returns

4 The meta method `invoke` returns the result of evaluating the executable code for the function represented by this `Function` object.

5.3.2 intrinsic::toString ( )

Description

1 The intrinsic `toString` method converts the executable code of the function to a string representation. This representation has the syntax of a *FunctionDeclaration*. Note in particular that the use and placement of white space, line terminators, and semicolons within the representation string is implementation-dependent.

> **FIXME** It doesn't make a lot of sense for (function () {}).toString() to return something that looks like a *FunctionDeclaration*, since the function has no name, so we might at least specify what is produced in that case.

Returns

2 The intrinsic `toString` method returns a string.

Implementation

```
intrinsic function toString(): string
    informative::source;
```

3 The informative property `source` holds a string representation of this function object.

5.3.3 intrinsic::apply (thisObj=..., args=...)

Description

1 The intrinsic `apply` method calls the static `apply` method with the current value of `this` as the first argument.

Returns

2 The intrinsic `apply` method returns the result of the static `apply` method.

Implementation

```
intrinsic function apply(thisArg=undefined, argArray=undefined)
    Function.apply(this, thisArg, argArray);
```

5.3.4 intrinsic::call (thisObj=..., ...args)

Description

1 The intrinsic `call` method calls the static `apply` method with the current value of `this` as the first argument.

Returns

2 The intrinsic `call` method returns the result of the static `call` method.

Implementation

```
intrinsic function call(thisObj=undefined, ...args)
    Function.apply(this, thisObj, args);
```

5.3.5 [[HasInstance]] (V)

FIXME Is this what we want?

Description

1 The `[[HasInstance]]` method of a `Function` object called with a value *V* determines if *V* is an instance of the `Function` object.

Returns

2 A boolean value.

Implementation

```
intrinsic function HasInstance(V) {
    if (!(V is Object))
        return false;

    let O : Object = this.prototype;
    if (!(O is Object))
        throw new TypeError("[[HasInstance]]: prototype is not object");

    while (true) {
        V = magic::getPrototype(V);
        if (V === null)
            return false;
        if (O == V)
            return true;
    }
}
```

3 The magic `getPrototype` function extracts the `[[Prototype]]` property from the object (see magic:getPrototype).

5.4 Properties of `Function` instances

1 In addition to the required internal properties, every function instance has a `[[Call]]` property, a `[[Construct]]` property and a `[[Scope]]` property (see sections 8.6.2 and 13.2).

5.4.1 length

1 The value of the constant `length` property is the number of non-rest arguments accepted by the function.

2 The value of the `length` property is an integer that indicates the "typical" number of arguments expected by the function. However, the language permits the function to be invoked with some other number of arguments. The behaviour of a function when invoked on a number of arguments other than the number specified by its length property depends on the function.

5.4.2 prototype

1 The initial value of the `prototype` property is a fresh `Object` instance.

2 The value of the `prototype` property is used to initialise the internal `[[Prototype]]` property of a newly created object before the `Function` instance is invoked as a constructor for that newly created object.

5.5 Invoking the `Function` prototype object

1 When the `Function` prototype object is invoked it accepts any arguments and returns **undefined**:

```
meta prototype function invoke(...args)
    undefined;
```

5.6 Methods on the `Function` prototype object

1 The methods on the `Function` prototype object call their intrinsic counterparts:

```
prototype function toString()
    this.source;

prototype function apply(thisArg, argArray)
    Function.apply(this, thisArg, argArray);

prototype function call(thisObj, ...args)
    Function.apply(this, thisObj, args);
```

2 The Function prototype object does not have a `valueOf` property of its own; however, it inherits the `valueOf` property from the `Object` prototype object.

6 The class `Array`

```
FILE:                       spec/library/Array.html
DRAFT STATUS:               DRAFT 1 - ROUGH
REVIEWED AGAINST ES3:       NO
REVIEWED AGAINST PROPOSALS: NO
REVIEWED AGAINST CODE:      NO
```

1 The class `Object` is a dynamic non-final subclass of `Object` (see class Object).

2 Array objects give special treatment to a certain class of property names. A property name that can be interpreted as an unsigned integer less than 2^{32}-1 is an *array index*.

3 A property name *P* of some type *T* from among `int`, `double`, `decimal`, or `string` is an array index if and only if *T(uint(P))* is equal to *P* and *uint(P)* is not equal to 2^{32}-1.

FIXME What about `Name` objects and `String` objects more generally? For the latter, maybe a general `ToString` conversion applies, but for the former?

4 Every `Array` object has a `length` property whose value is always a nonnegative integer less than 2^{32}. The value of the `length` property is numerically greater than the name of every property whose name is an array index; whenever a property of an Array object is created or changed, other properties are adjusted as necessary to maintain this invariant. Specifically, whenever a property is added whose name is an array index, the `length` property is changed, if necessary, to be one more than the numeric value of that array index; and whenever the `length` property is changed, every property whose name is an array index whose value is not smaller than the new length is automatically deleted. This constraint applies only to properties of the Array object itself and is unaffected by `length` or array index properties that may be inherited from its prototype.

5 The set of *array elements* held by any object (not just `Array` objects) are those properties of the object that are named by array indices numerically less than the object's `length` property. (If the object has no `length` property then its value is assumed to be zero, and the object has no array elements.)

6.1 Synopsis

1 The Array class provides the following interface:

```
dynamic class Array extends Object
{
    function Array(...args) …
    meta static function invoke(...items) …

    static function concat(object/*: Object!*/, ...items): Array …
    static function every(object/*:Object!*/, checker/*:function*/, thisObj:Object=null)
        : boolean …
    static function filter(object/*:Object!*/, checker/*function*/, thisObj:Object=null)
        : Array …
    static function forEach(object/*:Object!*/, eacher/*function*/, thisObj:Object=null)
        : void …
    static function indexOf(object/*:Object!*/, value, from:Numeric=0): Numeric …
    static function join(object/*: Object!*/, separator: string=","): string …
    static function lastIndexOf(object/*:Object!*/, value, from:Numeric=NaN)
        : Numeric …
    static function map(object/*:Object!*/, mapper/*:function*/, thisObj:Object=null)
        : Array …
    static function pop(object/*:Object!*/) …
    static function push(object/*: Object!*/, ...args): uint …
    static function reverse(object/*: Object!*/)/*: Object!*/ …
    static function shift(object/*: Object!*/) …
    static function slice(object/*: Object!*/, start: Numeric=0, end: Numeric=Infinity) …
    static function some(object/*:Object!*/, checker/*:function*/, thisObj:Object=null)
        : boolean …
    static function sort(object/*: Object!*/, comparefn) …
    static function splice(object/*: Object!*/, start: Numeric, deleteCount:
Numeric, ...items)
        : Array …
    static function unshift(object/*: Object!*/, ...items) : uint …

    static const length = 1

    intrinsic function concat(...items): Array …
    intrinsic function every(checker:Checker, thisObj:Object=null): boolean …
    intrinsic function filter(checker:Checker, thisObj:Object=null): Array …
    intrinsic function forEach(eacher:Eacher, thisObj:Object=null): void …
    intrinsic function indexOf(value, from:Numeric=0): Numeric …
    intrinsic function join(separator: string=","): string …
    intrinsic function lastIndexOf(value, from:Numeric=NaN): Numeric …
    intrinsic function map(mapper:Mapper, thisObj:Object=null): Array …
    intrinsic function pop() …
    intrinsic function push(...args): uint …
```

```
    intrinsic function reverse()/*: Object!*/ …
    intrinsic function shift() …
    intrinsic function slice(start: Numeric=0, end: Numeric=Infinity): Array …
    intrinsic function some(checker:Checker, thisObj:Object=null): boolean …
    intrinsic function sort(comparefn:Comparator):Array …
    intrinsic function splice(start: Numeric, deleteCount: Numeric, ...items)
        : Array …
    intrinsic function unshift(...items): uint …

    function get length(): uint …
    function set length(len: uint): void …
}
```

2 The `Array` prototype object provides these direct properties:

```
toString:       function () … ,
toLocaleString: function () … ,
concat:         function (...items) … ,
every:          function (checker, thisObj=null) … ,
filter:         function (checker, thisObj=null) … ,
forEach:        function (eacher, thisObj=null) … ,
indexOf:        function (value, from=0) … ,
join:           function (separator=",") … ,
lastIndexOf:    function (value, from=Infinity) … ,
map:            function (mapper, thisObj=null) … ,
pop:            function () … ,
push:           function (...items) … ,
reverse:        function () … ,
shift:          function () … ,
slice:          function (start=0, end=Infinity) … ,
some:           function (checker, thisObj=null) … ,
sort:           function (comparefn=undefined) … ,
splice:         function (start, deleteCount, ...items) … ,
unshift:        function (...items) … ,
length:         …
```

6.2 Methods on the `Array` class object

1 The `Array` class provides a number of static methods for manipulating array elements: `concat`, `every`, `filter`, `forEach`, `indexOf`, `join`, `lastIndexOf`, `map`, `pop`, `push`, `reverse`, `shift`, `slice`, `some`, `sort`, `splice`, and `unshift`. These static methods are intentionally *generic*; they do not require that their *object* argument be an `Array` object. Therefore they can be applied to other kinds of objects as well. Whether the generic Array methods can be applied successfully to a host object is implementation-dependent.

COMPATIBILITY NOTE The static generic methods on the Array class are all new in 4th edition.

6.2.1 new Array (…items)

Description

1 When the `Array` constructor is called with some set of arguments *items* as part of a `new Array` expression, it initializes the `Array` object from its argument values.

2 If there is exactly one argument of any number type, then its value is taken to be the initial value of the `length` property. The value must be a nonnegative integer less than 2^{32}.

3 If there are zero or more than one arguments, the arguments are taken to be the initial values of array elements, and there will be as many elements as there are arguments.

Implementation

```
function Array(...items) {
    if (items.length === 1) {
        let item = items[0];
        if (item is Numeric) {
            if (uint(item) === item)
                this.length = uint(item);
            else
                throw new RangeError("Invalid array length");
        }
        else {
            this.length = 1;
            this[0] = item;
        }
    }
    else {
        this.length = items.length;
        for ( let i=0, limit=items.length ; i < limit ; i++ )
            this[i] = items[i];
    }
}
```

6.2.2 Array (...items)

Description

1 When `Array` class is called as a function rather than as a constructor, it creates and initialises a new `Array` object. Thus the function call `Array(…)` is equivalent to the object creation expression new `Array(…)` with the same arguments.

Returns

2 The `Array` class called as function returns a new `Array` object.

Implementation

```
meta static function invoke(...items) {
    if (items.length == 1)
        return new Array(items[0]);
    else
        return items;
}
```

6.2.3 concat (object, ...items)

Description

1 The static `concat` method collects the array elements from *object* followed by the array elements from the additional *items*, in order, into a new Array object. All the *items* must be objects.

Returns

2 The static `concat` method returns a new `Array` object.

Implementation

```
static function concat(object/*: Object!*/, ...items): Array
    helper::concat(object, items);

helper static function concat(object/*: Object!*/, items: Array): Array {
    let out = new Array;

    let function emit(x) {
        if (x is Array) {
            for (let i=0, limit=x.length ; i < limit ; i++)
                out[out.length] = x[i];
        }
        else
            out[out.length] = x;
    }

    emit( object );
    for (let i=0, limit=items.length ; i < limit ; i++)
        emit( items[i] );

    return out;
}
```

3 The helper `concat` method is also used by the intrinsic and prototype variants of `concat`.

6.2.4 every (object, checker, thisObj=…)

Description

1 The static `every` method calls *checker* on every array element of *object* in increasing numerical index order, stopping as soon as any call returns **false**.

2 *Checker* is called with three arguments: the property value, the property index, and *object* itself. The *thisObj* is used as the `this` object in the call.

Returns

3 The static `every` method returns **true** if all the calls to *checker* returned true values, otherwise it returns **false**.

Implementation

```
static function every(object/*:Object!*/, checker/*:function*/, thisObj:Object=null): boolean
 {

    if (typeof checker != "function")
        throw new TypeError("Function object required to 'every'");

    for (let i=0, limit=object.length ; i < limit ; i++) {
        if (i in object)
            if (!checker.call(thisObj, object[i], i, object))
                return false;
```

```
    }
    return true;
}
```

6.2.5 filter (object, checker, thisObj=…)

Description

1 The static `filter` method calls *checker* on every array element of *object* in increasing numerical index order, collecting all the array elements for which checker returns a true value.

2 *Checker* is called with three arguments: the property value, the property index, and *object* itself. The *thisObj* is used as the `this` object in the call.

Returns

3 The static `filter` method returns a new `Array` object containing the elements that were collected, in the order they were collected.

Implementation

```
static function filter(object/*:Object!*/, checker/*function*/, thisObj:Object=null): Array {

    if (typeof checker != "function")
        throw new TypeError("Function object required to 'filter'");

    let result = [];
    for (let i = 0, limit=object.length ; i < limit ; i++) {
        if (i in object) {
            let item = object[i];
            if (checker.call(thisObj, item, i, object))
                result[result.length] = item;
        }
    }
    return result;
}
```

6.2.6 forEach (object, eacher, thisObj=…)

Description

1 The static `forEach` method calls *eacher* on every array element of *object* in increasing numerical index order, discarding any return value of *eacher*.

2 *Eacher* is called with three arguments: the property value, the property index, and *object* itself. The *thisObj* is used as the `this` object in the call.

Returns

3 The static `forEach` method does not return a value.

Implementation

```
static function forEach(object/*:Object!*/, eacher/*function*/, thisObj:Object=null): void {

    if (typeof eacher != "function")
        throw new TypeError("Function object required to 'forEach'");

    for (let i=0, limit = object.length ; i < limit ; i++)
        if (i in object)
            eacher.call(thisObj, object[i], i, object);
}
```

6.2.7 indexOf (object, value, from=…)

Description

1 The static `indexOf` method compares *value* with every array element of *object* in increasing numerical index order, starting at the index *from*, stopping when an array element is equal to *value* by the `===` operator.

2 *From* is rounded toward zero before use. If *from* is negative, it is treated as *`object`*`.length+`*`from`*.

Returns

3 The static `indexOf` method returns the array index the first time *value* is equal to an element, or -1 if no such element is found.

Implementation

```
static function indexOf(object/*:Object!*/, value, from:Numeric=0): Numeric {
    let len = object.length;
```

```
    from = from < 0 ? Math.ceil(from) : Math.floor(from);
    if (from < 0)
        from = from + len;

    while (from < len) {
        if (from in object)
            if (value === object[from])
                return from;
        from = from + 1;
    }
    return -1;
}
```

6.2.8 join (object, separator=…)

Description

1 The static `join` method concatenates the string representations of the array elements of *object* in increasing numerical index order, separating the individual strings by occurrences of *separator*.

Returns

2 The static `join` method returns the complete concatenated string.

Implementation

```
static function join(object/*: Object!*/, separator: string=","): string {
    let out = "";

    for (let i=0, limit=uint(object.length) ; i < limit ; i++) {
        if (i > 0)
            out += separator;
        let x = object[i];
        if (x !== undefined && x !== null)
            out += string(x);
    }

    return out;
}
```

6.2.9 lastIndexOf (object, value, from=…)

Description

1 The static `lastIndexOf` method compares *value* with every array element of *object* in decreasing numerical index order, starting at the index *from*, stopping when an array element is equal to *value* by the `===` operator.

2 *From* is rounded toward zero before use. If *from* is negative, it is treated as *`object`*`.length+`*`from`*.

Returns

3 The static `lastIndexOf` method returns the array index the first time *value* is equal to an element, or -1 if no such element is found.

Implementation

```
static function lastIndexOf(object/*:Object!*/, value, from:Numeric=NaN): Numeric {
    let len = object.length;

    if (isNaN(from))
        from = len - 1;
    else {
        from = from < 0 ? Math.ceil(from) : Math.floor(from);
        if (from < 0)
            from = from + len;
        else if (from >= len)
            from = len - 1;
    }

    while (from > -1) {
        if (from in object)
            if (value === object[from])
                return from;
        from = from - 1;
    }
    return -1;
}
```

6.2.10 map (object, mapper, thisObj=…)

Description

1 The static `map` method calls *mapper* on each array element of *object* in increasing numerical index order, collecting the return values from *mapper* in a new `Array` object.

2 *Mapper* is called with three arguments: the property value, the property index, and *object* itself. The *thisObj* is used as the `this` object in the call.

Returns

3 The static `map` method returns a new `Array` object where the array element at index *i* is the value returned from the call to *mapper* on *object[i]*.

Implementation

```
static function map(object/*:Object!*/, mapper/*:function*/, thisObj:Object=null): Array {

    if (typeof mapper != "function")
        throw new TypeError("Function object required to 'map'");

    let result = [];
    for (let i = 0, limit = object.length; i < limit ; i++)
        if (i in object)
            result[i] = mapper.call(thisObj, object[i], i, object);
    return result;
}
```

6.2.11 pop (object)

Description

1 The static `pop` method extracts the last array element from *object* and removes it by decreasing the value of the `length` property of *object* by 1.

Returns

2 The static `pop` method returns the removed element.

Implementation

```
static function pop(object/*:Object!*/) {
    let len = uint(object.length);

    if (len != 0) {
        len = len - 1;
        let x = object[len];
        delete object[len]
        object.length = len;
        return x;
    }
    else {
        object.length = len;
        return undefined;
    }
}
```

6.2.12 push (object, ...items)

Description

1 The static `push` method appends the values in *items* to the end of the array elements of *object*, in the order in which they appear, in the process updating the `length` property of *object*.

Returns

2 The static `push` method returns the new value of the `length` property of *object*.

Implementation

```
static function push(object/*: Object!*/, ...args): uint
    Array.helper::push(object, args);

helper static function push(object/*:Object!*/, args: Array): uint {
    let len = uint(object.length);

    for (let i=0, limit=args.length ; i < limit ; i++)
        object[len++] = args[i];

    object.length = len;
    return len;
}
```

3 The helper `push` method is also used by the intrinsic and prototype variants of `push`.

6.2.13 reverse (object)

Description

1 The static `reverse` method rearranges the array elements of *object* so as to reverse their order. The `length` property of *object* remains unchanged.

Returns

2 The static `reverse` method returns *object*.

Implementation

```
static function reverse(object/*: Object!*/)/*: Object!*/ {
    let len = uint(object.length);
    let middle = Math.floor(len / 2);

    for ( let k=0 ; k < middle ; ++k ) {
        let j = len - k - 1;
        if (j in object) {
            if (k in object)
                [object[k], object[j]] = [object[j], object[k]];
            else {
                object[k] = object[j];
                delete object[j];
            }
        }
        else if (k in object) {
            object[j] = object[k];
            delete object[k];
        }
        else {
            delete object[j];
            delete object[k];
        }
    }

    return object;
}
```

6.2.14 shift (object)

Description

1 The static `shift` method removes the element called `0` in *object*, moves the element at index *i+1* to index *i*, and decrements the `length` property of *object* by 1.

Returns

2 The static `shift` method returns the element that was removed.

Implementation

```
static function shift(object/*: Object!*/) {
    let len = uint(object.length);
    if (len == 0) {
        object.length = 0;
        return undefined;
    }

    let x = object[0];

    for (let i = 1; i < len; i++)
        object[i-1] = object[i];
    delete object[len - 1];
    object.length = len - 1;
    return x;
}
```

6.2.15 slice (object, start=..., end=...)

Description

1 The static `slice` method extracts the subrange of array elements from *object* between *start* (inclusive) and *end* (exclusive) into a new Array.

2 If *start* is negative, it is treated as *`object`*`.length+`*`start`*. If *end* is negative, it is treated as *`object`*`.length+`*`end`*. In either case the values of *start* and *end* are bounded between 0 and *`object`*`.length`.

Returns

3 The static `slice` method returns a new `Array` object containing the extracted array elements.

Implementation

```
static function slice(object/*: Object!*/, start: Numeric=0, end: Numeric=Infinity) {
    let len = uint(object.length);
```

```
    let a = helper::clamp(start, len);
    let b = helper::clamp(end, len);
    if (b < a)
        b = a;

    let out = new Array;
    for (let i = a; i < b; i++)
        out.push(object[i]);

    return out;
}

helper static function clamp(intValue: double, len: uint): uint {
    if (intValue < 0) {
        if (intValue + len < 0)
            return 0;
        else
            return uint(intValue + len);
    }
    else if (intValue > len)
        return len;
    else
        return uint(intValue);
}
```

6.2.16 some (object, checker, thisObj=…)

Description

1 The static `some` method calls *checker* on every array element in *object* in increasing numerical index order, stopping as soon as *checker* returns a true value.

2 *Checker* is called with three arguments: the property value, the property index, and the object itself. The *thisObj* is used as the `this` object in the call.

Returns

3 The static `some` method returns **true** when *checker* returns a true value, otherwise returns **false** if all the calls to *checker* return false values.

Implementation

```
static function some(object/*:Object!*/, checker/*:function*/, thisObj:Object=null): boolean
{

    if (typeof checker != "function")
        throw new TypeError("Function object required to 'some'");

    for (let i=0, limit=object.length; i < limit ; i++) {
        if (i in object)
            if (checker.call(thisObj, object[i], i, object))
                return true;
    }
    return false;
}
```

6.2.17 sort (object, comparefn=…)

Description

1 The static `sort` method sorts the array elements of *object*, it rearranges the elements of *object* according to some criterion.

2 The sort is not necessarily stable (that is, elements that compare equal do not necessarily remain in their original order). If *comparefn* is not **undefined**, it should be a function that accepts two arguments *x* and *y* and returns a negative value if $x < y$, zero if $x = y$, or a positive value if $x > y$.

3 If *comparefn* is not **undefined** and is not a consistent comparison function for the array elements of *object* (see below), the behaviour of `sort` is implementation-defined. Let *len* be `uint(object.length)`. If there exist integers *i* and *j* and an object *P* such that all of the conditions below are satisfied then the behaviour of `sort` is implementation-defined:

1. $0 \le i < len$
2. $0 \le j < len$
3. *object* does not have a property with name `ToString(i)`
4. *P* is obtained by following one or more `[[Prototype]]` properties starting at this
5. *P* has a property with name `ToString(j)`

FIXME Probably use `uint(x)` rather than `ToUint32(x)` throughout.

FIXME The use of `ToString` is not suitable for ES4 (though it is correct). See comments at the top of the Array section.

4 Otherwise the following steps are taken.

1. Let *M* be the result of calling the `[[Get]]` method of *object* with argument `"length"`.
2. Let *L* be the result of `ToUint32(`*`M`*`)`.
3. Perform an implementation-dependent sequence of calls to the `[[Get]]`, `[[Put]]`, and `[[Delete]]` methods of *object* and to *SortCompare* (described below), where the first argument for each call to `[[Get]]`, `[[Put]]`, or `[[Delete]]` is a nonnegative integer less than *L* and where the arguments for calls to *SortCompare* are results of previous calls to the `[[Get]]` method.

5 Following the execution of the preceding algorithm, *object* must have the following two properties.

1. There must be some mathematical permutation π of the nonnegative integers less than *L*, such that for every nonnegative integer *j* less than *L*, if property *old[j]* existed, then *new[π(j)]* is exactly the same value as *old[j]*, but if property *old[j]* did not exist, then *new[π(j)]* does not exist.
2. Then for all nonnegative integers *j* and *k*, each less than *L*, if *SortCompare(j,k) < 0* (see *SortCompare* below), then *π(j) < π(k)*.

6 Here the notation *old[j]* is used to refer to the hypothetical result of calling the `[[Get]]` method of this object with argument *j* before this function is executed, and the notation *new[j]* to refer to the hypothetical result of calling the `[[Get]]` method of this object with argument *j* after this function has been executed.

7 A function *comparefn* is a consistent comparison function for a set of values *S* if all of the requirements below are met for all values *a*, *b*, and *c* (possibly the same value) in the set *S*: The notation *a <CF b* means *comparefn(a,b) < 0*; *a =CF b* means *comparefn(a,b) = 0* (of either sign); and *a >CF b* means *comparefn(a,b) > 0*.

1. Calling *comparefn(a,b)* always returns the same value *v* when given a specific pair of values *a* and *b* as its two arguments. Furthermore, *v* has type *Number*, and *v* is not **NaN**. Note that this implies that exactly one of *a <CF b*, *a =CF b*, and *a >CF b* will be true for a given pair of *a* and *b*.
2. *a =CF a* (reflexivity)
3. If *a =CF b*, then *b =CF a* (symmetry)
4. If *a =CF b* and *b =CF c*, then *a =CF c* (transitivity of *=CF*)
5. If *a <CF b* and *b <CF c*, then *a <CF c* (transitivity of *<CF*)
6. If *a >CF b* and *b >CF c*, then *a >CF c* (transitivity of *>CF*)

NOTE The above conditions are necessary and sufficient to ensure that *comparefn* divides the set *S* into equivalence classes and that these equivalence classes are totally ordered.

Returns

8 The static `sort` method returns *object*.

Implementation

10 The interface to the `sort` method can be described as follows: static function sort(object/*: Object!*/, comparefn) …

11 When the *SortCompare* operator is called with two arguments *j* and *k*, the following steps are taken:

```
helper function sortCompare(j:uint, k:uint, comparefn:Comparator): Numeric {
    if (!(j in this) && !(k in this))
        return 0;
    if (!(j in this))
        return 1;
    if (!(k in this))
        return -1;

    let x = this[j];
    let y = this[k];

    if (x === undefined && y === undefined)
        return 0;
    if (x === undefined)
        return 1;
    if (y === undefined)
        return -1;

    if (comparefn === undefined) {
        x = x.toString();
        y = y.toString();
```

```
        if (x < y) return -1;
        if (x > y) return 1;
        return 0;
    }
    return comparefn(x, y);
}
```

NOTE Because non-existent property values always compare greater than **undefined** property values, and **undefined** always compares greater than any other value, **undefined** property values always sort to the end of the result, followed by non-existent property values.

6.2.18 splice (object, start, deleteCount, ...items)

Description

1 The static `splice` method replaces the *deleteCount* array elements of *object* starting at array index *start* with values from the *items*.

Returns

2 The static `splice` method returns a new Array object containing the array elements that were removed from *objects*, in order.

Implementation

```
static function splice(object/*: Object!*/, start: Numeric, deleteCount: Numeric, ...items):
Array
    Array.helper::splice(object, start, deleteCount, items);

helper static function splice(object/*: Object!*/, start: Numeric, deleteCount: Numeric,
items: Array) {
    let out = new Array();

    let numitems = uint(items.length);
    if (numitems == 0)
        return undefined;

    let len = object.length;
    let start = helper::clamp(double(items[0]), len);
    let d_deleteCount = numitems > 1 ? double(items[1]) : (len - start);
    let deleteCount = (d_deleteCount < 0) ? 0 : uint(d_deleteCount);
    if (deleteCount > len - start)
        deleteCount = len - start;

    let end = start + deleteCount;

    for (let i:uint = 0; i < deleteCount; i++)
        out.push(object[i + start]);

    let insertCount = (numitems > 2) ? (numitems - 2) : 0;
    let l_shiftAmount = insertCount - deleteCount;
    let shiftAmount;

    if (l_shiftAmount < 0) {
        shiftAmount = uint(-l_shiftAmount);

        for (let i = end; i < len; i++)
            object[i - shiftAmount] = object[i];

        for (let i = len - shiftAmount; i < len; i++)
            delete object[i];
    }
    else {
        shiftAmount = uint(l_shiftAmount);

        for (let i = len; i > end; ) {
            --i;
            object[i + shiftAmount] = object[i];
        }
    }

    for (let i:uint = 0; i < insertCount; i++)
        object[start+i] = items[i + 2];

    object.length = len + l_shiftAmount;
    return out;
}
```

3 The helper `clamp` function was defined earlier (see Array.slice).

6.2.19 unshift (object, ...items)

Description

1 The static `unshift` method inserts the values in *items* as new array elements at the start of *object*, such that their order within the array elements of *object* is the same as the order in which they appear in

items. Existing array elements in *object* are shifted upward in the index range, and the `length` property of *object* is updated.

Returns

2 The static `unshift` method returns the new value of the `length` property of *object*.

Implementation

```
static function unshift(object/*: Object!*/, ...items) : uint
    Array.helper::unshift(this, object, items);

helper static function unshift(object/*: Object!*/, items: Array) : uint {
    let len = uint(object.length);
    let numitems = items.length;

    for ( let k=len-1 ; k >= 0 ; --k ) {
        let d = k + numitems;
        if (k in object)
            object[d] = object[k];
        else
            delete object[d];
    }

    for (let i=0; i < numitems; i++)
        object[i] = items[i];

    object.length = len+numitems;

    return len+numitems;
}
```

6.3 Method Properties of Array Instances

6.3.1 Intrinsic methods

Description

1 The intrinsic methods on Array instances delegate to their static counterparts. Unlike their static and prototype counterparts, these methods are bound by their instance and they are not generic.

Returns

2 The intrinsic methods on Array instances return what their static counterparts return.

Implementation

```
override intrinsic function toString():string
    join();

override intrinsic function toLocaleString():string {
    let out = "";
    for (let i = 0, limit = this.length; i < limit ; i++) {
        if (i > 0)
            out += ",";
        let x = this[i];
        if (x !== null && x !== undefined)
            out += x.toLocaleString();
    }
    return out;
}

intrinsic function concat(...items): Array
    Array.helper::concat(this, items);

intrinsic function every(checker:Checker, thisObj:Object=null): boolean
    Array.every(this, checker, thisObj);

intrinsic function filter(checker:Checker, thisObj:Object=null): Array
    Array.filter(this, checker, thisObj);

intrinsic function forEach(eacher:Eacher, thisObj:Object=null): void {
    Array.forEach(this, eacher, thisObj);
}

intrinsic function indexOf(value, from:Numeric=0): Numeric
    Array.indexOf(this, value, from);

intrinsic function join(separator: string=","): string
    Array.join(this, separator);

intrinsic function lastIndexOf(value, from:Numeric=NaN): Numeric
    Array.lastIndexOf(this, value, from);

intrinsic function map(mapper:Mapper, thisObj:Object=null): Array
    Array.map(this, mapper, thisObj);

intrinsic function pop()
    Array.pop(this);
```

```
intrinsic function push(...args): uint
    Array.helper::push(this, args);

intrinsic function reverse()/*: Object!*/
    Array.reverse(this);

intrinsic function shift()
    Array.shift(this);

intrinsic function slice(start: Numeric=0, end: Numeric=Infinity): Array
    Array.slice(this, start, end);

intrinsic function some(checker:Checker, thisObj:Object=null): boolean
    Array.some(this, checker, thisObj);

intrinsic function sort(comparefn:Comparator):Array
    Array.sort(this, comparefn);

intrinsic function splice(start: Numeric, deleteCount: Numeric, ...items): Array
    Array.helper::splice(this, start, deleteCount, items);

intrinsic function unshift(...items): uint
    Array.helper::unshift(this, items);
```

6.3.2 [[Put]] (P, V)

1 Array objects use a variation of the `[[Put]]` method used for other native ECMAScript objects (section 8.6.2.2).

2 Assume A is an Array object and P is a string.

FIXME P may be not-a-string in ES4.

3 When the `[[Put]]` method of A is called with property P and value V, the following steps are taken:

1. Call the `[[CanPut]]` method of A with name P.
2. If Result(1) is false, return.
3. If A doesn't have a property with name P, go to step 7.
4. If P is "length", go to step 12.
5. Set the value of property P of A to V.
6. Go to step 8.
7. Create a property with name P, set its value to V and give it empty attributes.
8. If P is not an array index, return.
9. If ToUint32(P) is less than the value of the length property of A, then return.
10. Change (or set) the value of the length property of A to ToUint32(P)+1.
11. Return.
12. Compute ToUint32(V).
13. If Result(12) is not equal to ToNumber(V), throw a RangeError exception.
14. For every integer k that is less than the value of the length property of A but not less than Result (12), if A itself has a property (not an inherited property) named ToString(k), then delete that property.
15. Set the value of property P of A to Result(12).
16. Return.

6.4 Value properties of Array instances

1 Array instances inherit properties from the Array prototype object and also have the following properties.

6.4.1 length

1 The `length` property of this Array object is always numerically greater than the name of every property whose name is an array index.

6.5 Method properties on the `Array` prototype object

=

6.5.1 toString ( )

Description

1 The prototype `toString` method converts the array to a `string`. It has the same effect as if the intrinsic `join` method were invoked for this object with no argument.

Returns

2 The prototype `toString` method returns a `string`.

Implementation

```
prototype function toString(this:Array)
    this.join();
```

6.5.2 toLocaleString ( )

Description

1 The elements of this Array are converted to strings using their public `toLocaleString` methods, and these strings are then concatenated, separated by occurrences of a separator string that has been derived in an implementation-defined locale-specific way. The result of calling this function is intended to be analogous to the result of `toString`, except that the result of this function is intended to be locale-specific.

Returns

2 The prototype `toLocaleString` method returns a `string`.

Implementation

```
prototype function toLocaleString(this:Array)
    this.toLocaleString();
```

NOTE The first parameter to this function is likely to be used in a future version of this standard; it is recommended that implementations do not use this parameter position for anything else.

6.5.3 Generic methods

1 These methods delegate to their static counterparts, and like their counterparts, they are generic: they can be transferred to other objects for use as methods. Whether these methods can be applied successfully to a host object is implementation-dependent.

```
prototype function concat(...items)
    Array.helper::concat(this, items);

prototype function every(checker, thisObj=null)
    Array.every(this, checker, thisObj);

prototype function filter(checker, thisObj=null)
    Array.filter(this, checker, thisObj);

prototype function forEach(eacher, thisObj=null) {
    Array.forEach(this, eacher, thisObj);
}

prototype function indexOf(value, from=0)
    Array.indexOf(this, value, ToNumeric(from));

prototype function join(separator=undefined)
    Array.join(this, separator === undefined ? "," : string(separator));

prototype function lastIndexOf(value, from=NaN)
    Array.lastIndexOf(this, value, ToNumeric(from));

prototype function map(mapper, thisObj=null)
    Array.map(this, mapper, thisObj);

prototype function pop()
    Array.pop(this);

prototype function push(...args)
    Array.helper::push(this, args);

prototype function reverse()
    Array.reverse(this);

prototype function shift()
    Array.shift(this);

prototype function slice(start, end)
    Array.slice(this,
                start === undefined ? 0 : ToNumeric(start),
                end === undefined ? Infinity : ToNumeric(end));

prototype function some(checker, thisObj=null)
    Array.some(this, checker, thisObj);
```

```
prototype function sort(comparefn)
    Array.sort(this, comparefn);

prototype function splice(start, deleteCount, ...items)
    Array.helper::splice(this, ToNumeric(start), ToNumeric(deleteCount), items);

prototype function unshift(...items)
    Array.helper::unshift(this, items);
```

COMPATIBILITY NOTE In the 3rd Edition of this Standard some of the functions on the Array prototype object had `length` properties that did not reflect those functions' signatures. In the 4th Edition of this Standard, all functions on the Array prototype object have `length` properties that follow the general rule stated in section function-semantics.

7 The class `Date`

```
FILE:                       spec/library/Date.html
DRAFT STATUS:               DRAFT 1 - ROUGH
REVIEWED AGAINST ES3:       NO
REVIEWED AGAINST PROPOSALS: NO
REVIEWED AGAINST CODE:      NO
```

1 The `Date` object serves two purposes: as a record of an instant in time, and as a simple timer.

2 Time is measured in ECMAScript in milliseconds since 01 January, 1970 UTC (the "epoch"), and a `Date` object contains a number indicating a particular instant in time to within a millisecond relative to the epoch. The number may also be NaN, indicating that the Date object does not represent a specific instant of time.

3 A `Date` object also contains a record of its time of creation to nanosecond precision, and can be queried for the elapsed time since its creation to within a nanosecond.

7.1 Synopsis

1 The `Date` class provides this interface:

```
dynamic class Date extends Object
{
    function Date(year=NOARG, month=NOARG, date=NOARG, hours=NOARG, minutes=NOARG,
seconds=NOARG, ms=NOARG) …
    meta static function invoke(...args)   // args are ignored. …

    static intrinsic function parse(s:string, reference:double=0.0) : double …
    static intrinsic function UTC(year: double, …
    static function now() : double …

    static var parse = function parse(string, reference:double=0.0) …
    static var UTC = …

    override intrinsic function toString() : string …
    intrinsic function toDateString() : string …
    intrinsic function toTimeString():string …
    override intrinsic function toLocaleString() : string …
    intrinsic function toLocaleDateString() : string …
    intrinsic function toLocaleTimeString() : string …
    intrinsic function toUTCString() : string …
    intrinsic function toISOString() : string …
    intrinsic function nanoAge() : double …
    intrinsic function getTime() : double …
    intrinsic function getYear() : double …
    intrinsic function getFullYear() : double …
    intrinsic function getUTCFullYear() : double …
    intrinsic function getMonth() : double …
    intrinsic function getUTCMonth() : double …
    intrinsic function getDate() : double …
    intrinsic function getUTCDate() : double …
    intrinsic function getDay() : double …
    intrinsic function getUTCDay() : double …
    intrinsic function getHours() : double …
    intrinsic function getUTCHours() : double …
    intrinsic function getMinutes() : double …
    intrinsic function getUTCMinutes() : double …
    intrinsic function getSeconds() : double …
    intrinsic function getUTCSeconds() : double …
    intrinsic function getMilliseconds() : double …
    intrinsic function getUTCMilliseconds() : double …
    intrinsic function getTimezoneOffset() : double …

    intrinsic function setTime(t:double) : double …
    intrinsic function setYear(this:Date, year:double) …
    intrinsic function setFullYear(year:double, …
    intrinsic function setUTCFullYear(year:double, …
    intrinsic function setMonth(month:double, date:double = getDate()):double …
    intrinsic function setUTCMonth(month:double, date:double = getUTCDate()):double …
    intrinsic function setDate(date: double): double …
    intrinsic function setUTCDate(date: double): double …
    intrinsic function setHours(hour: double, …
    intrinsic function setUTCHours(hour: double, …
    intrinsic function setMinutes(min:double, …
    intrinsic function setUTCMinutes(min:double, …
    intrinsic function setSeconds(sec:double, ms:double = getMilliseconds())
        : double …
    intrinsic function setUTCSeconds(sec:double, ms:double = getUTCMilliseconds())
        : double …
    intrinsic function setMilliseconds(ms:double) : double …
    intrinsic function setUTCMilliseconds(ms:double) : double …

    function get time(this:Date) : double …
    function get year(this:Date) : double …
    function get fullYear(this:Date) : double …
```

```
    function get UTCFullYear(this:Date) : double …
    function get month(this:Date) : double …
    function get UTCMonth(this:Date) : double …
    function get date(this:Date) : double …
    function get UTCDate(this:Date) : double …
    function get day(this:Date) : double …
    function get UTCDay(this:Date) : double …
    function get hours(this:Date) : double …
    function get UTCHours(this:Date) : double …
    function get minutes(this:Date) : double …
    function get UTCMinutes(this:Date) : double …
    function get seconds(this:Date) : double …
    function get UTCSeconds(this:Date) : double …
    function get milliseconds(this:Date) : double …
    function get UTCMilliseconds(this:Date) : double …

    function set time(this:Date, t : double) : double …
    function set year(this:Date, t: double) : double …
    function set fullYear(this:Date, t : double) : double …
    function set UTCFullYear(this:Date, t : double) : double …
    function set month(this:Date, t : double) : double …
    function set UTCMonth(this:Date, t : double) : double …
    function set date(this:Date, t : double) : double …
    function set UTCDate(this:Date, t : double) : double …
    function set hours(this:Date, t : double) : double …
    function set UTCHours(this:Date, t : double) : double …
    function set minutes(this:Date, t : double) : double …
    function set UTCMinutes(this:Date, t : double) : double …
    function set seconds(this:Date, t : double) : double …
    function set UTCSeconds(this:Date, t : double) : double …
    function set milliseconds(this:Date, t : double) : double …
    function set UTCMilliseconds(this:Date, t : double) : double …

    private var timeval: double = …
}
```

2 The `Date` prototype object is itself a `Date` object whose time value is NaN. It provides the following direct properties:

```
    toString:           function () … ,
    toDateString:       function () … ,
    toTimeString:       function () … ,
    toLocaleString:     function () … ,
    toLocaleDateString: function () … ,
    toLocaleTimeString: function () … ,
    toUTCString:        function () … ,
    toISOString:        function () … ,
    valueOf:            function () … ,
    getTime:            function () … ,
    getFullYear:        function () … ,
    getUTCFullYear:     function () … ,
    getMonth:           function () … ,
    getUTCMonth:        function () … ,
    getDate:            function () … ,
    getUTCDate:         function () … ,
    getDay:             function () … ,
    getUTCDay:          function () … ,
    getHours:           function () … ,
    getUTCHours:        function () … ,
    getMinutes:         function () … ,
    getUTCMinutes:      function () … ,
    getSeconds:         function () … ,
    getUTCSeconds:      function () … ,
    getMilliseconds:    function () … ,
    getUTCMilliseconds: function () … ,
    getTimezoneOffset:  function () … ,
    setTime:            function (time) … ,
    setMilliseconds:    function (ms) … ,
    setUTCMilliseconds: function (ms) … ,
    setSeconds:         function (sec, ms=undefined) … ,
    setUTCSeconds:      function (sec, ms=undefined) … ,
    setMinutes:         function (min, sec=undefined, ms=undefined) … ,
    setUTCMinutes:      function (min, sec=undefined, ms=undefined) … ,
    setHours:           function (hour, min=undefined, sec=undefined, ms=undefined) … ,
    setUTCHours:        function (hour, min=undefined, sec=undefined, ms=undefined) … ,
    setDate:            function (date) … ,
    setUTCDate:         function (date) … ,
    setMonth:           function (month, date=undefined) … ,
    setUTCMonth:        function (month, date=undefined) … ,
    setFullYear:        function (year, month=undefined, date=undefined) … ,
    setUTCFullYear:     function (year, month=undefined, date=undefined) … ,
```

7.2 Overview of Date Objects and Definitions of Helper Functions

1 A `Date` object contains a private property `timeval` that indicates a particular instant in time to within a millisecond. The number may also be **NaN**, indicating that the `Date` object does not represent a specific instant of time.

2 The following sections define a number of helper functions for operating on time values. Note that, in every case, if any argument to such a function is **NaN**, the result will be **NaN**.

3 For the sake of succinctness, the `helper` and `informative` namespaces are open in all the definitions that follow.

7.2.1 Time Range

1 Time is measured in ECMAScript in milliseconds since 01 January, 1970 UTC. Leap seconds are ignored. It is assumed that there are exactly 86,400,000 milliseconds per day. ECMAScript `double` values can represent all integers from -9,007,199,254,740,991 to 9,007,199,254,740,991; this range suffices to measure times to millisecond precision for any instant that is within approximately 285,616 years, either forward or backward, from 01 January, 1970 UTC.

2 The actual range of times supported by ECMAScript Date objects is slightly smaller: exactly -100,000,000 days to 100,000,000 days measured relative to midnight at the beginning of 01 January, 1970 UTC. This gives a range of 8,640,000,000,000,000 milliseconds to either side of 01 January, 1970 UTC.

3 The exact moment of midnight at the beginning of 01 January, 1970 UTC is represented by the value +0.

7.2.2 Constants

1 The following simple constants are used by the helper functions defined below.

```
helper const hoursPerDay = 24;

helper const minutesPerHour = 60;

helper const secondsPerMinute = 60;

helper const daysPerYear = 365.2425;

helper const msPerSecond = 1000;

helper const msPerMinute = msPerSecond * secondsPerMinute;

helper const msPerHour = msPerMinute * minutesPerHour;

helper const msPerDay = msPerHour * hoursPerDay;

helper const msPerYear = msPerDay * daysPerYear;
```

2 The table `monthOffsets` contains the day offset within a non-leap year of the first day of each month:

```
helper const monthOffsets = [0, 31, 59, 90, 120, 151, 181, 212, 243, 273, 304, 334];
```

7.2.3 Day Number and Time within Day

1 A given time value *t* belongs to day number `Day(t)`:

```
helper function Day(t : double) : double
    Math.floor(t / msPerDay);
```

2 The remainder is called the time within the day, `TimeWithinDay(t)`:

```
helper function TimeWithinDay(t : double) : double
    t % msPerDay;
```

7.2.4 Year Number

1 ECMAScript uses an extrapolated Gregorian system to map a day number to a year number and to determine the month and date within that year. In this system, leap years are precisely those which are (divisible by 4) and ((not divisible by 100) or (divisible by 400)). The number of days in year number *y* is therefore defined by `DaysInYear(y)`:

```
helper function DaysInYear(y : double) : double {
    if (y % 4 !== 0 || y % 100 === 0 && y % 400 !== 0)
        return 365;
    else
        return 366;
}
```

2 All non-leap years have 365 days with the usual number of days per month and leap years have an extra day in February. The day number of the first day of year *y* is given by `DayFromYear(y)`:

```
helper function DayFromYear(y : double) : double
    365 * (y-1970) + Math.floor((y-1969)/4) - Math.floor((y-1901)/100) + Math.floor((y-1601)
/400);
```

3 The time value of the start of a year *y* is `TimefromYear(y)`:

```
helper function TimeFromYear(y : double) : double
    msPerDay * DayFromYear(y);
```

4 A time value *t* determines a year by `YearFromTime(t)`, which yields the largest integer *y* (closest to positive infinity) such that `TimeFromYear(y) ≤ t`.

5 The function `YearFromTime` is not defined precisely by this Standard.

```
informative static function YearFromTime(t: double): double …
```

FIXME Is there any good reason not to define how `YearFromTime` should be computed? The RI uses a non-iterative algorithm which I believe comes from SpiderMonkey. I have seen iterative algorithms elsewhere.

6 The leap-year function `InLeapYear` is 1 for a time within a leap year and otherwise is zero:

```
helper function InLeapYear(t : double) : double
    (DaysInYear(YearFromTime(t)) == 365) ? 0 : 1;
```

7.2.5 Month Number

1 Months are identified by an integer in the range 0 to 11, inclusive. The mapping from a time value *t* to a month number is defined by `MonthFromTime(t)` :

```
helper function MonthFromTime(t : double) : double {
    let dwy = DayWithinYear(t),
        ily = InLeapYear(t);
    for ( let i=monthOffsets.length-1; i >= 0; i-- ) {
        let firstDayOfMonth = monthOffsets[i];
        if (i >= 2)
            firstDayOfMonth += ily;
        if (dwy >= firstDayOfMonth)
            return i;
    }
}

helper function DayWithinYear(t : double) : double
    Day(t) - DayFromYear(YearFromTime(t));
```

2 A month value of 0 specifies January; 1 specifies February; 2 specifies March; 3 specifies April; 4 specifies May; 5 specifies June; 6 specifies July; 7 specifies August; 8 specifies September; 9 specifies October; 10 specifies November; and 11 specifies December.

NOTE `MonthFromTime(0)=0`, corresponding to Thursday, 01 January, 1970.

7.2.6 Date Number

1 A date number is identified by an integer in the range 1 through 31, inclusive. The mapping from a time value *t* to a month number is defined by `DateFromTime(t)`:

```
helper function DateFromTime(t : double) : double {
    let dwy = DayWithinYear(t),
        mft = MonthFromTime(t),
        ily = InLeapYear(t);
    return (dwy+1) - (monthOffsets[mft]) - (mft >= 2 ? ily : 0);
}
```

7.2.7 Week Day

1 The weekday for a particular time value *t* is defined as `WeekDay(t)`:

```
helper function WeekDay(t : double) : double {
    let v = (Day(t) + 4) % 7;
    if (v < 0)
        return v + 7;
    return v;
}
```

2 A weekday value of 0 specifies Sunday; 1 specifies Monday; 2 specifies Tuesday; 3 specifies Wednesday; 4 specifies Thursday; 5 specifies Friday; and 6 specifies Saturday.

NOTE `WeekDay(0)=4`, corresponding to Thursday, 01 January, 1970.

7.2.8 Local Time Zone Adjustment

1 An implementation of ECMAScript is expected to determine the local time zone adjustment. The local time zone adjustment is a value `LocalTZA` measured in milliseconds which when added to UTC represents the local standard time. Daylight saving time is not reflected by `LocalTZA`.

```
informative function LocalTZA(): double …
```

2 The value `LocalTZA` does not vary with time but depends only on the geographic location.

FIXME This is bogus because it assumes time zone boundaries are fixed for all eternity. Yet time zone (standard time) is political; changing political conditions can lead to adoption of a different standard time (analogous to the changes in daylight savings time adjustment). So the above assertion needs to go, and probably be replaced by language similar to that we want to adopt for DaylightSavingsTA, which encourages "best effort for the given time".

7.2.9 Daylight Saving Time Adjustment

1 An implementation of ECMAScript is expected to determine the daylight saving time algorithm. The algorithm to determine the daylight saving time adjustment for a time *t*, implemented by `DaylightSavingTA(t)`, measured in milliseconds, must depend only on four things:

1. The time since the beginning of the year: `t - TimeFromYear(YearFromTime(t))`
2. Whether t is in a leap year: `InLeapYear(t)`
3. The week day of the beginning of the year: `WeekDay(TimeFromYear(YearFromTime(t))`
4. The geographic location.

2 The implementation of ECMAScript should not try to determine whether the exact time *t* was subject to daylight saving time, but just whether daylight saving time would have been in effect if the current daylight saving time algorithm had been used at the time. This avoids complications such as taking into account the years that the locale observed daylight saving time year round.

3 If the host environment provides functionality for determining daylight saving time, the implementation of ECMAScript is free to map the year in question to an equivalent year (same leapyear-ness and same starting week day for the year) for which the host environment provides daylight saving time information. The only restriction is that all equivalent years should produce the same result.

FIXME We've already agreed that the above is bogus; the implementation needs to make a "best effort" to find the correct adjustment for the time *t*, in the year of *t*. More to come here. Also see note above for `LocalTZA`.

7.2.10 Local Time

1 Conversion from UTC to local time is defined by

```
helper function LocalTime(t : double) : double
    t + LocalTZA() + DaylightSavingsTA(t);
```

2 Conversion from local time to UTC is defined by

```
helper function UTCTime(t : double) : double
    t - LocalTZA() - DaylightSavingsTA(t - LocalTZA());
```

3 Note that `UTCTime(LocalTime(t))` is not necessarily always equal to *t* because the former expands as `t+DaylightSavingsTA(t)-DaylightSavingsTA(t-LocalTZA())`.

7.2.11 Hours, Minutes, Seconds, and Milliseconds

1 The following functions are useful in decomposing time values:

```
helper function HourFromTime(t : double) : double {
    let v = Math.floor(t / msPerHour) % hoursPerDay;
    if (v < 0)
        return v + hoursPerDay;
    return v;
}
```

```
helper function MinFromTime(t : double) : double {
    let v = Math.floor(t / msPerMinute) % minutesPerHour;
    if (v < 0)
        return v + minutesPerHour;
    return v;
}

helper function SecFromTime(t : double) : double {
    let v = Math.floor(t / msPerSecond) % secondsPerMinute;
    if (v < 0)
        return v + secondsPerMinute;
    return v;
}

helper function msFromTime(t : double) : double
    t % msPerSecond;
```

7.2.12 MakeTime (hour, min, sec, ms)

1 The operator MakeTime calculates a number of milliseconds from its four arguments, which must be ECMAScript number values. This operator functions as follows:

```
helper function MakeTime(hour:double, min:double, sec:double, ms:double ):double {
    if (!isFinite(hour) || !isFinite(min) || !isFinite(sec) || !isFinite(ms))
        return NaN;

    return (ToInteger(hour) * msPerHour +
            ToInteger(min) * msPerMinute +
            ToInteger(sec) * msPerSecond +
            ToInteger(ms));
}
```

7.2.13 MakeDay (year, month, date)

1 The helper function `MakeDay` calculates a number of days from its three arguments, which must be ECMAScript `double` values:

```
helper function MakeDay(year : double, month : double, date : double) : double {
    if (!isFinite(year) || !isFinite(month) || !isFinite(date))
        return NaN;

    year = ToInteger(year);
    month = ToInteger(month);
    date = ToInteger(date);

    /* INFORMATIVE, the spec is non-operational. */
    year += Math.floor(month / 12);

    month = month % 12;
    if (month < 0)
        month += 12;

    let leap = (DaysInYear(year) == 366);

    let yearday = Math.floor(TimeFromYear(year) / msPerDay);
    let monthday = DayFromMonth(month, leap);

    return yearday + monthday + date - 1;
}
```

7.2.14 MakeDate (day, time)

1 The helper function `MakeDate` calculates a number of milliseconds from its two arguments, which must be ECMAScript `double` values:

```
helper function MakeDate(day : double, time : double) : double {
    if (!isFinite(day) || !isFinite(time))
        return NaN;

    return day * msPerDay + time;
}
```

7.2.15 TimeClip (time)

1 The helper function `TimeClip` calculates a number of milliseconds from its argument, which must be an ECMAScript `double` value:

```
helper function TimeClip(t : double) : double
    (!isFinite(t) || Math.abs(t) > 8.64e15) ? NaN : adjustZero(ToInteger(t));
```

informative function adjustZero(t: double): double …

NOTE The informative function `adjustZero(t)` can either return *t* unchanged or it can add (+0) to it. The point of this freedom is that an implementation is permitted a choice of internal representations of time values, for example as a 64-bit signed integer or as a 64-bit floating-point value. Depending on the implementation, this internal representation may or may not distinguish -0 and +0.

7.3 Date strings

1 Dates can be converted to string representations for purposes of human consumption and data transmission in a number of ways, many of them locale-dependent.

2 Some of the string representations of dates are required to be lossless, which is to say that converting a time value to a string and then parsing that string as a Date will always yield the same time value. Other string representations are implementation-dependent and it is not guaranteed that they can be parsed to yield the same time value (or that they can be parsed at all).

3 This Standard defines numerous methods on `Date` instances to generate strings from time values: `toString`, `toDateString`, `toTimeString`, `toLocaleString`, `toLocaleDateString`, `toLocaleTimeString`, `toUTCString`, and `toISOString`.

4 The `toString` and `toUTCString` methods convert time values to a string losslessly except for fractional seconds, which may not be represented in the string. The format of these strings is implementation-dependent.

5 The `toISOString` method converts time values to a string losslessly, and the string conforms to the ISO date grammar defined below.

6 This Standard defines the static `parse` method on the `Date` class to parse strings and compute time values represented by those strings. The `parse` method is only required to parse all strings that conform to the ISO date grammar defined below, as well as all strings produced by the `toString` and `toUTCString` methods on `Date` instances.

7 The grammar for ISO date strings is defined by the following regular expression:

```
helper const isoTimestamp =
    /^
     # Date, optional
     (?: (?P<year> - [0-9]+ | [0-9]{4} [0-9]* )
      (?: - (?P<month> [0-9]{2} )
       (?: - (?P<day> [0-9]{2} ) )? )? )?
     T
     # Time, optional
     (?: (?P<hour> [0-9]{2} )
      (?: : (?P<minutes> [0-9]{2} )
       (?: : (?P<seconds> [0-9]{2} )
        (?: \. (?P<fraction> [0-9]+ ) )? )? )? )?
     # Timezone, optional
     (?: (?P<zulu> Z )
      | (?P<offs>
         (?P<tzdir> \+ | - )
         (?P<tzhr> [0-9]{2} )
         (?: : (?P<tzmin> [0-9]{2} ) )? ) )?
     $/x;
```

FIXME Replace the regexp by a proper grammar, eventually.

7.4 Methods on the Date class

7.4.1 new Date (year=…, month=…, date=…, hours=…, minutes=…, seconds=…, ms=…)

Description

1 When the `Date` constructor is called as part of a `new Date` expression it initialises the newly created object by setting its private `timeval` property.

2 The `Date` constructor can be called with zero, one, or two to seven arguments, and sets `timeval` in different ways depending on how it is called.

Implementation

```
function Date(year=NOARG, month=NOARG, date=NOARG, hours=NOARG, minutes=NOARG, seconds=NOARG,
 ms=NOARG) {
    informative::setupNanoAge();

    switch (NOARG) {
```

```
    case year:
        timeval = Date.now();
        return;

    case month: {
        let v = ToPrimitive(year);
        if (v is string)
            return parse(v);

        timeval = TimeClip(double(v));
        return;
    }

    default:
        ms = double(ms);

    case ms:
        seconds = double(seconds);

    case seconds:
        minutes = double(minutes);

    case minutes:
        hours = double(hours);

    case hours:
        date = double(date);

    case date:
        year = double(year);
        month = double(month);

        let intYear : int = ToInteger(year);
        if (!isNaN(year) && 0 <= intYear && intYear <= 99)
            intYear += 1900;
        timeval = TimeClip(UTCTime(MakeDate(MakeDay(intYear, month, date),
                                            MakeTime(hours, minutes, seconds, ms))));
    }
}
```

NOTE The default value NOARG is an unforgeable private value and is used to detect the difference between an unsupplied parameter and a parameter value of **undefined**.

7.4.2 Date (...args)

Description

1 When the Date class is called as a function rather than as a constructor, it converts the current time (as returned by the static method `now` on `Date`) to a string.

2 All arguments are ignored. A string is created as if by the expression `(new Date()).toString()`.

NOTE The function call `Date(…)` is not equivalent to the object creation expression `new Date(…)` with the same arguments.

Returns

3 The `Date` class called as a function returns a `string` object.

Implementation

```
meta static function invoke(...args)   // args are ignored.
    (new Date()).public::toString();
```

7.4.3 intrinsic::parse (s, reference=…)

Description

1 The static intrinsic `parse` method applies the `string` function to its argument *s* and interprets the resulting string as a date. The string may be interpreted as a local time, a UTC time, or a time in some other time zone, depending on the contents of the string.

2 The value *reference* (defaulting to zero) is a time value that will provide default values for any fields missing from the string.

3 If *x* is any Date object whose milliseconds amount is zero within a particular implementation of ECMAScript, then all of the following expressions should produce the same numeric value in that implementation, if all the properties referenced have their initial values:

```
x.valueOf()
Date.parse(x.toString())
Date.parse(x.toUTCString())
```

4 However, the expression `Date.parse(x.toLocaleString())` is not required to produce the same number value as the preceding three expressions and, in general, the value produced by Date.parse is implementation-dependent when given any string value that could not be produced in that implementation by the toString or toUTCString method.

Returns

5 The static `parse` method returns a number, the UTC time value corresponding to the date represented by the string.

Implementation

6 The static `parse` method parses a string that conforms to the ISO grammar as an ISO date string. Otherwise, the parsing is implementation-dependent.

```
static intrinsic function parse(s:string, reference:double=0.0) : double {

    function fractionToMilliseconds(frac: string): double
        Math.floor(1000 * (parseInt(frac) / Math.pow(10, frac.length)));

    let isoRes = isoTimestamp.exec(s);
    let defaults = new Date(reference);
    if (isoRes) {
        let year = isoRes.year !== undefined ? parseInt(isoRes.year) : defaults.UTCYear;
        let month = isoRes.month !== undefined ? parseInt(isoRes.month)-1 :
defaults.UTCMonth;
        let day = isoRes.day !== undefined ? parseInt(isoRes.day) : defaults.UTCDay;
        let hour = isoRes.hour !== undefined ? parseInt(isoRes.hour) : defaults.UTCHour;
        let mins = isoRes.minutes !== undefined ? parseInt(isoRes.minutes) :
defaults.UTCMinutes;
        let secs = isoRes.seconds !== undefined ? parseInt(isoRes.seconds) :
defaults.UTCSeconds;
        let millisecs = isoRes.fraction !== undefined ?
            fractionToMilliseconds(isoRes.fraction) :
            defaults.UTCMilliseconds;
        let tzo = defaults.timezoneOffset;
        if (isoRes.zulu !== undefined)
            tzo = 0;
        else if (isoRes.offs !== undefined) {
            tzo = parseInt(isoRes.tzhr) * 60;
            if (isoRes.tzmin !== undefined)
                tzo += parseInt(isoRes.tzmin);
            if (isoRes.tzdir === "-")
                tzo = -tzo;
        }
        return new Date.UTC(year, month, day, hour, mins, secs, millisecs) - tzo;
    }
    else
        return informative::fromDateString(s, reference);
}
```

7.4.4 parse(s, reference=…)

Description

1 The static `parse` method applies the `string` function to its argument *s* and the `double` function to its argument *reference* (which defaults to zero), and then calls the intrinsic `parse` method on the resulting values.

Returns

2 The static `parse` method returns a number, the UTC time value corresponding to the date represented by the string.

Implementation

```
static var parse = function parse(string, reference:double=0.0) {
    return Date.parse(ToString(string), reference);
}
```

7.4.5 intrinsic::UTC (year, month, date=…, hours=…, minutes=…, seconds=…, ms=…)

Description

1 When the static intrinsic `UTC` method is called with two to seven arguments, it computes the date from *year*, *month* and (optionally) *date*, *hours*, *minutes*, *seconds* and *ms*.

> **NOTE** The `UTC` method differs from the `Date` constructor in two ways: it returns a time value as a number, rather than creating a `Date` object, and it interprets the arguments in UTC rather than as local time.

Returns

2 The static intrinsic `UTC` method returns a time value.

Implementation

```
static intrinsic function UTC(year: double,
                              month: double,
                              date: double=1,
                              hours: double=0,
                              minutes: double=0,
                              seconds: double=0,
                              ms: double=0) : double
{
    let intYear = ToInteger(year);
    if (!isNaN(year) && 0 <= intYear && intYear <= 99)
        intYear += 1900;
    return TimeClip(MakeDate(MakeDay(intYear, month, date),
                             MakeTime(hours, minutes, seconds, ms)));
}
```

7.4.6 UTC (year, month, date=..., hours=..., minutes=..., seconds=..., ms=...)

Description

1 When the static intrinsic `UTC` method is called with fewer than two arguments, the behaviour is implementation dependent. When the `UTC` method is called with two to seven arguments, it computes the date from *year*, *month* and (optionally) *date*, *hours*, *minutes*, *seconds* and *ms* by converting all arguments to `double` values and calling the static intrinsic `UTC` method.

Returns

2 The static `UTC` method returns a time value.

Implementation

```
static var UTC =
    function UTC(year, month, date=NOARG, hours=NOARG, minutes=NOARG, seconds=NOARG,
ms=NOARG) {
        switch (NOARG) {
        case date:    date = 1;
        case hours:   hours = 0;
        case minutes: minutes = 0;
        case seconds: seconds = 0;
        case ms:      ms = 0;
        }
        return Date.UTC(double(year),
                        double(month),
                        double(date),
                        double(hours),
                        double(minutes),
                        double(seconds),
                        double(ms));
    };
```

NOTE The default value NOARG is an unforgeable private value and is used to detect the difference between an unsupplied parameter and a parameter value of **undefined**.

7.4.7 now

Description

1 The static `now` method produces the time value at the time of the call.

Returns

2 The static `now` method returns a `double` representing a time value.

Implementation

3 The static `now` method is implementation-dependent.

7.5 Methods on Date instances

7.5.1 intrinsic::toString ( )

Description

1 The intrinsic `toString` method converts the `Date` value to a string. The contents of the string are intended to represent the value in the current time zone in a convenient, human-readable form.

NOTE It is intended that for any `Date` value *d*, the result of `Date.parse(d.toString())` is equal to *d*. (See Date.parse.)

Returns

2 A `string` value.

Implementation

3 The intrinsic `toString` method is implementation-dependent.

7.5.2 intrinsic::toDateString ( )

Description

1 The intrinsic `toLocaleString` method converts the "date" portion of the `Date` value to a string. The contents of the string are intended to represent the value in the current time zone in a convenient, human-readable form.

Returns

2 A `string` value.

Implementation

3 The intrinsic `toDateString` method is implementation-dependent.

7.5.3 intrinsic::toTimeString ( )

Description

1 The intrinsic `toTimeString` method converts the "time" portion of the `Date` value to a string. The contents of the string are intended to represent the value in the current time zone in a convenient, human-readable form.

Returns

2 A `string` value.

Implementation

3 The intrinsic `toTimeString` method is implementation-dependent.

7.5.4 intrinsic::toLocaleString ( )

Description

1 The intrinsic `toLocaleString` method converts the `Date` value to a string. The contents of the string are intended to represent the value in the current time zone in a convenient, human-readable form that corresponds to the conventions of the host environment's current locale.

> **NOTE** The first parameter to this function is likely to be used in a future version of this standard; it is recommended that implementations do not use this parameter position for anything else.

Returns

2 A `string` value.

Implementation

3 The intrinsic `toLocaleString` method is implementation-dependent.

7.5.5 intrinsic::toLocaleDateString ( )

Description

1 The intrinsic `toLocaleDateString` method converts the "date" portion of the `Date` value to a string. The contents of the string are intended to represent the value in the current time zone in a convenient, human-readable form that corresponds to the conventions of the host environment's current locale.

> **NOTE** The first parameter to this function is likely to be used in a future version of this standard; it is recommended that implementations do not use this parameter position for anything else.

Returns

2 A `string` value.

Implementation

3 The intrinsic `toLocaleDateString` method is implementation-dependent.

7.5.6 intrinsic::toLocaleTimeString ( )

Description

1 The intrinsic `toLocaleTimeString` method converts the "time" portion of the `Date` value to a string. The contents of the string are intended to represent the value in the current time zone in a convenient, human-readable form that corresponds to the conventions of the host environment's current locale.

> **NOTE** The first parameter to this function is likely to be used in a future version of this standard; it is recommended that implementations do not use this parameter position for anything else.

Returns

2 A `string` value.

Implementation

3 The intrinsic `toLocaleTimeString` method is implementation-dependent.

7.5.7 intrinsic::toUTCString ( )

Description

1 The intrinsic `toUTCString` method converts the `Date` value to a string. The contents of the string are intended to represent the value in UTC in a convenient, human-readable form.

Returns

2 A `string` value.

Implementation

3 The intrinsic `toUTCString` method is implementation-dependent.

7.5.8 intrinsic::toISOString ( )

Description

1 The intrinsic `toISOString` method converts the `Date` value to a string. The string conforms to the ISO time and date grammar presented in section ISO date grammar. All fields are present in the string and the shortest possible nonempty string of digits follows the period in the time part. The time zone is always UTC, denoted by a suffix `Z`.

Returns

2 A `string` value.

Implementation

```
intrinsic function toISOString() : string {
    return (formatYears(UTCFullYear) + "-" +
            zeroFill(UTCMonth+1, 2) + "-" +
            zeroFill(UTCDate, 2) +
            "T" +
            zeroFill(UTCHours, 2) + ":" +
            zeroFill(UTCMinutes, 2) + ":" +
            zeroFill(UTCSeconds, 2) + "." +
            removeTrailingZeroes(int(UTCMilliseconds)) +
            "Z");
}

helper function formatYears(n: double): string {
    if (n >= 0 && n <= 9999)
        return zeroFill(int(n), 4);
    else
        return n.toString();
}
```

3 The helper functions `removeTrailingZeroes` and `zeroFill` are described in section Minor date helpers.

7.5.9 intrinsic::nanoAge()

Description

1 The intrinsic `nanoAge` method computes an approximation of the number of nanoseconds of real time that have elapsed since this `Date` object was created.

> **NOTE** The approximation is of unspecified quality, and may vary in both accuracy and precision from platform to platform. The approximation will necessarily lose precision as its object ages, since it is expressed as a double: after approximately 104 days of real time, its object will have been alive for over 2^{53} nanoseconds, so the result of this call will carry more than 2 nanoseconds rounding error after 104 days, and more than 4 nanoseconds rounding error after 208 days. Code wishing to

measure greater periods of real time may either construct fresh Date objects after 104 days, or accept the gradual loss of precision.

Returns

2 A `double` object.

Implementation

3 The static `nanoAge` method is implementation-dependent.

7.5.10 intrinsic::valueOf ( )

Description

1 The intrinsic `valueOf` method returns the time value of the `Date` object.

Returns

2 A `double` object.

Implementation

```
override intrinsic function valueOf() : Object
    getTime();
```

7.5.11 intrinsic::getTime ( )

Description

1 The intrinsic `getTime` method retrieves the full time value of the Date object.

Returns

2 This time value.

Implementation

```
intrinsic function getTime() : double
    timeval;
```

7.5.12 intrinisic::getFullYear ( )

Description

1 The intrinsic `getFullYear` method retrieves the year number of the Date object, in the local time zone.

Returns

2 A year number (year number).

Implementation

```
intrinsic function getFullYear() : double
    let (t = timeval)
        isNaN(t) ? t : YearFromTime(LocalTime(t));
```

7.5.13 intrinisic::getUTCFullYear ( )

Description

1 The intrinsic `getUTCFullYear` method retrieves the year number of the Date object, in UTC.

FIXME Is the phrasing "in UTC" appropriate? (Ditto for all following functions.)

Returns

2 A year number (year number).

Implementation

```
intrinsic function getUTCFullYear() : double
    let (t = timeval)
        isNaN(t) ? t : YearFromTime(t);
```

7.5.14 intrinisic::getMonth ( )

Description

1 The intrinsic `getMonth` method retrieves the month number of the Date object, in the local time zone.

Returns

2 A month number (month number).

Implementation

```
intrinsic function getMonth() : double
    let (t = timeval)
        isNaN(t) ? t : MonthFromTime(LocalTime(t));
```

7.5.15 intrinisic::getUTCMonth ( )

Description

1 The intrinsic `getUTCMonth` method retrieves the month number of the Date object, in UTC.

Returns

2 A month number (month number).

Implementation

```
intrinsic function getUTCMonth() : double
    let (t = timeval)
        isNaN(t) ? t : MonthFromTime(t);
```

7.5.16 intrinisic::getDate ( )

Description

1 The intrinsic `getDate` method retrieves the date number of the Date object, in the local time zone.

Returns

2 A date number (date number).

Implementation

```
intrinsic function getDate() : double
    let (t = timeval)
        isNaN(t) ? t : DateFromTime(LocalTime(t));
```

7.5.17 intrinisic::getUTCDate ( )

Description

1 The intrinsic `getUTCDate` method retrieves the date number of the Date object, in UTC.

Returns

2 A date number (date number).

Implementation

```
intrinsic function getUTCDate() : double
    let (t = timeval)
        isNaN(t) ? t : DateFromTime(t);
```

7.5.18 intrinisic::getDay ( )

Description

1 The intrinsic `getDay` method retrieves the day number of the Date object, in the local time zone.

Returns

2 A day number (day number).

Implementation

```
intrinsic function getDay() : double
    let (t = timeval)
        isNaN(t) ? t : WeekDay(LocalTime(t));
```

7.5.19 intrinisic::getUTCDay ( )

Description

1 The intrinsic `getUTCDay` method retrieves the day number of the Date object, in UTC.

Returns

2 A day number (day number).

Implementation

```
intrinsic function getUTCDay() : double
    let (t = timeval)
        isNaN(t) ? t : WeekDay(t);
```

7.5.20 intrinisic::getHours ( )

Description

1 The intrinsic `getHours` method retrieves the hours value of the Date object, in the local time zone.

Returns

2 An hours value (hours, minutes, seconds, and milliseconds).

Implementation

```
intrinsic function getHours() : double
    let (t = timeval)
        isNaN(t) ? t : HourFromTime(LocalTime(t));
```

7.5.21 intrinisic::getUTCHours ( )

Description

1 The intrinsic `getUTCHours` method retrieves the hours value of the Date object, in UTC.

Returns

2 An hours value (hours, minutes, seconds, and milliseconds).

Implementation

```
intrinsic function getUTCHours() : double
    let (t = timeval)
        isNaN(t) ? t : HourFromTime(t);
```

7.5.22 intrinisic::getMinutes ( )

Description

1 The intrinsic `getMinutes` method retrieves the minutes value of the Date object, in the local time zone.

Returns

2 A minutes value (hours, minutes, seconds, and milliseconds).

Implementation

```
intrinsic function getMinutes() : double
    let (t = timeval)
        isNaN(t) ? t : MinFromTime(LocalTime(t));
```

7.5.23 intrinisic::getUTCMinutes ( )

Description

1 The intrinsic `getUTCMinutes` method retrieves the minutes value of the Date object, in UTC.

Returns

2 A minutes value (hours, minutes, seconds, and milliseconds).

Implementation

```
intrinsic function getUTCMinutes() : double
    let (t = timeval)
        isNaN(t) ? t : MinFromTime(t);
```

7.5.24 intrinisic::getSeconds ( )

Description

1 The intrinsic `getSeconds` method retrieves the seconds value of the Date object, in the local time zone.

Returns

2 A seconds value (hours, minutes, seconds, and milliseconds).

Implementation

```
intrinsic function getSeconds() : double
    let (t = timeval)
        isNaN(t) ? t : SecFromTime(LocalTime(t));
```

7.5.25 intrinisic::getUTCSeconds ( )

Description

1 The intrinsic `getUTCSeconds` method retrieves the seconds value of the Date object, in UTC.

Returns

2 A seconds value (hours, minutes, seconds, and milliseconds).

Implementation

```
intrinsic function getUTCSeconds() : double
    let (t = timeval)
        isNaN(t) ? t : SecFromTime(t);
```

7.5.26 intrinisic::getMilliseconds ( )

Description

1 The intrinsic `getMilliseconds` method retrieves the milliseconds value of the Date object, in the local time zone.

Returns

2 A milliseconds value (hours, minutes, seconds, and milliseconds).

Implementation

```
intrinsic function getMilliseconds() : double
    let (t = timeval)
        isNaN(t) ? t : msFromTime(LocalTime(t));
```

7.5.27 intrinisic::getUTCMilliseconds ( )

Description

1 The intrinsic `getUTCMilliseconds` method retrieves the milliseconds value of the Date object, in UTC.

Returns

2 A milliseconds value (hours, minutes, seconds, and milliseconds).

Implementation

```
intrinsic function getUTCMilliseconds() : double
    let (t = timeval)
        isNaN(t) ? t : msFromTime(t);
```

7.5.28 intrinisic::getTimezoneOffset ( )

Description

1 Computes the difference between local time and UTC time.

Returns

2 A possibly non-integer number of minutes.

Implementation

```
intrinsic function getTimezoneOffset() : double
    let (t = timeval)
        isNaN(t) ? t : (t - LocalTime(t)) / msPerMinute;
```

7.5.29 intrinisic::setTime (time)

Description

1 The intrinsic `setTime` method sets the time value of the Date object.

Returns

2 The new time value.

Implementation

```
intrinsic function setTime(t:double) : double
    timeval = TimeClip(t);
```

7.5.30 intrinisic::setMilliseconds (ms)

Description

1 The intrinsic `setMilliseconds` method sets the milliseconds value of the Date object, taking *ms* to be a value in the local time zone.

Returns

2 The new time value.

Implementation

```
intrinsic function setMilliseconds(ms:double) : double
    timeval = let (t = LocalTime(timeval))
                  UTCTime(MakeDate(Day(t), MakeTime(HourFromTime(t),
                                                    MinFromTime(t),
                                                    SecFromTime(t),
                                                    ms)));
```

7.5.31 intrinisic::setUTCMilliseconds (ms)

Description

1 The intrinsic `setUTCMilliseconds` method sets the milliseconds value of the Date object, taking *ms* to be a value in UTC.

Returns

2 The new time value.

Implementation

```
intrinsic function setUTCMilliseconds(ms:double) : double
    timeval = let (t = timeval)
                  MakeDate(Day(t), MakeTime(HourFromTime(t),
                                            MinFromTime(t),
                                            SecFromTime(t),
                                            ms));
```

7.5.32 intrinisic::setSeconds (sec, ms=...)

Description

1 The intrinsic `setSeconds` method sets the seconds value (and optionally the milliseconds value) of the Date object, taking *sec* and *ms* to be values in the local time zone.

Returns

2 The new time value.

Implementation

```
intrinsic function setSeconds(sec:double, ms:double = getMilliseconds()) : double
    timeval = let (t = LocalTime(timeval))
                  UTCTime(MakeDate(Day(t), MakeTime(HourFromTime(t),
                                                    MinFromTime(t),
                                                    sec,
                                                    ms)));
```

FIXME Default arguments: is this the way we want it?

For this and the following methods the signature has the following impliciation: if a program subclasses Date and overrides the intrinsic `getMilliseconds()` method, the new method *will* be invoked if setSeconds is called with one argument.

There are various ways to avoid this, though I don't think it's really a problem that there is this dependence, except that it binds implementations in how they represent and handle dates.

3rd Edition has imprecise language here, it says that if ms is not provided by the caller then its value will be as if *ms* were specified with the value *getMilliseconds()*. Whether that implies that that method is called (and that the user could override it) or not is not at all clear.

7.5.33 intrinisic::setUTCSeconds (sec, ms=...)

Description

1 The intrinsic `setUTCSeconds` method sets the seconds value (and optionally the milliseconds value) of the Date object, taking *sec* and *ms* to be values in UTC.

Returns

2 The new time value.

Implementation

```
intrinsic function setUTCSeconds(sec:double, ms:double = getUTCMilliseconds()) : double
    timeval = let (t = timeval)
                MakeDate(Day(t), MakeTime(HourFromTime(t),
                                          MinFromTime(t),
                                          sec,
                                          ms));
```

7.5.34 intrinisic::setMinutes (min, sec=..., ms=...)

Description

1 The intrinsic `setMinutes` method sets the minutes value (and optionally the seconds and milliseconds values) of the Date object, taking *min*, *sec* and *ms* to be values in the local time zone.

Returns

2 The new time value.

Implementation

```
intrinsic function setMinutes(min:double,
                              sec:double = getSeconds(),
                              ms:double = getMilliseconds()) : double
    timeval = let (t = LocalTime(timeval))
                UTCTime(MakeDate(Day(t), MakeTime(HourFromTime(t),
                                                  min,
                                                  sec,
                                                  ms)));
```

7.5.35 intrinisic::setUTCMinutes (min, sec=..., ms=...)

Description

1 The intrinsic `setUTCMinutes` method sets the minutes value (and optionally the seconds and milliseconds values) of the Date object, taking *min*, *sec* and *ms* to be values in UTC.

Returns

2 The new time value.

Implementation

```
intrinsic function setUTCMinutes(min:double,
                                 sec:double = getUTCSeconds(),
                                 ms:double = getUTCMilliseconds()) : double
    timeval = let (t = timeval)
                MakeDate(Day(t), MakeTime(HourFromTime(t),
                                          min,
                                          sec,
                                          ms));
```

7.5.36 intrinisic::setHours (hour, min=minutes, sec=..., ms=...)

Description

1 The intrinsic `setHours` method sets the hours value (and optionally the minutes, seconds, and milliseconds values) of the Date object, taking *hour*, *min*, *sec* and *ms* to be values in the local time zone.

Returns

2 The new time value.

Implementation

```
intrinsic function setHours(hour: double,
                            min: double = getMinutes(),
                            sec: double = getSeconds(),
                            ms: double = getMilliseconds()) : double
    timeval = let (t = LocalTime(timeval))
                UTCTime(MakeDate(Day(t), MakeTime(hour,
                                                  min,
                                                  sec,
                                                  ms)));
```

7.5.37 intrinisic::setUTCHours (hour, min=..., sec=..., ms=...)

Description

1 The intrinsic `setUTCHours` method sets the hours value (and optionally the minutes, seconds, and milliseconds values) of the Date object, taking *hour*, *min*, *sec* and *ms* to be values in UTC.

Returns

2 The new time value.

Implementation

```
intrinsic function setUTCHours(hour: double,
                               min: double = getUTCMinutes(),
                               sec: double = getUTCSeconds(),
                               ms: double = getUTCMilliseconds()) : double
    timeval = let (t = timeval)
                  MakeDate(Day(t), MakeTime(hour,
                                            min,
                                            sec,
                                            ms));
```

7.5.38 intrinisic::setDate (date)

Description

1 The intrinsic `setDate` method sets the date value of the Date object, taking *date* to be a value in the local time zone.

Returns

2 The new time value.

Implementation

```
intrinsic function setDate(date: double): double
    timeval = let (t = LocalTime(timeval))
                  UTCTime(MakeDate(MakeDay(YearFromTime(t), MonthFromTime(t), date),
                                   TimeWithinDay(t)));
```

7.5.39 intrinisic::setUTCDate (date)

Description

1 The intrinsic `setUTCDate` method sets the date value of the Date object, taking *date* to be a value in UTC.

Returns

2 The new time value.

Implementation

```
intrinsic function setUTCDate(date: double): double
    timeval = let (t = timeval)
                  MakeDate(MakeDay(YearFromTime(t), MonthFromTime(t), date),
                           TimeWithinDay(t));
```

7.5.40 intrinisic::setMonth (month, date=…)

Description

1 The intrinsic `setMonth` method sets the month value (and optionally the date value) of the Date object, taking *month* and *date* to be values in the local time zone.

Returns

2 The new time value.

Implementation

```
intrinsic function setMonth(month:double, date:double = getDate()):double
    timeval = let (t = LocalTime(timeval))
                  UTCTime(MakeDate(MakeDay(YearFromTime(t), month, date),
                                   TimeWithinDay(t)));
```

7.5.41 intrinisic::setUTCMonth (month, date=…)

Description

1 The intrinsic `setUTCMonth` method sets the month value (and optionally the date value) of the Date object, taking *month* and *date* to be values in UTC.

Returns

2 The new time value.

Implementation

```
intrinsic function setUTCMonth(month:double, date:double = getUTCDate()):double
    timeval = let (t = timeval)
```

```
                MakeDate(MakeDay(YearFromTime(t), month, date),
                         TimeWithinDay(t));
```

7.5.42 intrinisic::setFullYear (year, month=..., date=...)

Description

1 The intrinsic `setFullYear` method sets the year value (and optionally the month and date values) of the Date object, taking *year*, *month*, and *date* to be values in the local time zone.

Returns

2 The new time value.

Implementation

```
intrinsic function setFullYear(year:double,
                               month:double = getMonth(),
                               date:double = getDate()) : double
    timeval = let (t = LocalTime(timeval))
                  UTCTime(MakeDate(MakeDay(year, month, date),
                                   TimeWithinDay(t)));
```

7.5.43 intrinisic::setUTCFullYear (year, month=..., date=...)

Description

1 The intrinsic `setFullYear` method sets the year value (and optionally the month and date values) of the Date object, taking *year*, *month*, and *date* to be values in UTC.

Returns

2 The new time value.

Implementation

```
intrinsic function setUTCFullYear(year:double,
                                  month:double = getUTCMonth(),
                                  date:double = getUTCDate()) : double
    timeval = let (t = timeval)
                  MakeDate(MakeDay(year, month, date),
                           TimeWithinDay(t));
```

7.6 Getters on Date instances

Description

1 The Date object provides a number of getters that call the object's corresponding accessor methods.

Returns

2 The getters all return what their corresponding accessor methods return.

Implementation

```
function get time(this:Date) : double
    getTime();

function get year(this:Date) : double
    getYear();

function get fullYear(this:Date) : double
    getFullYear();

function get UTCFullYear(this:Date) : double
    getUTCFullYear();

function get month(this:Date) : double
    getMonth();

function get UTCMonth(this:Date) : double
    getUTCMonth();

function get date(this:Date) : double
    getDate();

function get UTCDate(this:Date) : double
    getUTCDate();

function get day(this:Date) : double
    getDay();

function get UTCDay(this:Date) : double
    getUTCDay();

function get hours(this:Date) : double
    getHours();
```

```
function get UTCHours(this:Date) : double
    getUTCHours();

function get minutes(this:Date) : double
    getMinutes();

function get UTCMinutes(this:Date) : double
    getUTCMinutes();

function get seconds(this:Date) : double
    getSeconds();

function get UTCSeconds(this:Date) : double
    getUTCSeconds();

function get milliseconds(this:Date) : double
    getMilliseconds();

function get UTCMilliseconds(this:Date) : double
    getUTCMilliseconds();
```

7.7 Setters on Date instances

Description

1 The Date object provides a number of setters that call the object's corresponding updater methods. Since the setters only accept a single argument, the updaters will be called with default arguments for all arguments beyond the first.

Returns

2 The setters all return what their corresponding updater methods return.

Implementation

```
function set time(this:Date, t : double) : double
    setTime(t);

function set year(this:Date, t: double) : double
    setYear(t);

function set fullYear(this:Date, t : double) : double
    setFullYear(t);

function set UTCFullYear(this:Date, t : double) : double
    setUTCFullYear(t);

function set month(this:Date, t : double) : double
    setMonth(t);

function set UTCMonth(this:Date, t : double) : double
    setUTCMonth(t);

function set date(this:Date, t : double) : double
    setDate(t);

function set UTCDate(this:Date, t : double) : double
    setUTCDate(t);

function set hours(this:Date, t : double) : double
    setHours(t);

function set UTCHours(this:Date, t : double) : double
    setUTCHours(t);

function set minutes(this:Date, t : double) : double
    setMinutes(t);

function set UTCMinutes(this:Date, t : double) : double
    setUTCMinutes(t);

function set seconds(this:Date, t : double) : double
    setSeconds(t);

function set UTCSeconds(this:Date, t : double) : double
    setUTCSeconds(t);

function set milliseconds(this:Date, t : double) : double
    setMilliseconds(t);

function set UTCMilliseconds(this:Date, t : double) : double
    setUTCMilliseconds(t);
```

7.8 Method properties on the `Date` prototype object

Description

1 The `Date` prototype methods are not generic; their `this` object must be a `Date`. The methods forward the call to the corresponding intrinsic method in all cases.

Returns

2 The `Date` prototype methods return the values returned by the intrinsic methods they call.

Implementation

```
prototype function toString(this:Date)
    this.toString();

prototype function toDateString(this:Date)
    this.toDateString();

prototype function toTimeString(this:Date)
    this.toTimeString();

prototype function toLocaleString(this:Date)
    this.toLocaleString();

prototype function toLocaleDateString(this:Date)
    this.toLocaleDateString();

prototype function toLocaleTimeString(this:Date)
    this.toLocaleTimeString();

prototype function toUTCString(this:Date)
    this.toUTCString();

prototype function toISOString(this:Date)
    this.toISOString();

prototype function valueOf(this:Date)
    this.valueOf();

prototype function getTime(this:Date)
    this.intrinsic::getTime();

prototype function getFullYear(this:Date)
    this.intrinsic::getFullYear();

prototype function getUTCFullYear(this:Date)
    this.intrinsic::getUTCFullYear();

prototype function getMonth(this:Date)
    this.intrinsic::getMonth();

prototype function getUTCMonth(this:Date)
    this.intrinsic::getUTCMonth();

prototype function getDate(this:Date)
    this.intrinsic::getDate();

prototype function getUTCDate(this:Date)
    this.intrinsic::getUTCDate();

prototype function getDay(this:Date)
    this.intrinsic::getDay();

prototype function getUTCDay(this:Date)
    this.intrinsic::getUTCDay();

prototype function getHours(this:Date)
    this.intrinsic::getHours();

prototype function getUTCHours(this:Date)
    this.intrinsic::getUTCHours();

prototype function getMinutes(this:Date)
    this.intrinsic::getMinutes();

prototype function getUTCMinutes(this:Date)
    this.intrinsic::getUTCMinutes();

prototype function getSeconds(this:Date)
    this.intrinsic::getSeconds();

prototype function getUTCSeconds(this:Date)
    this.intrinsic::getUTCSeconds();

prototype function getMilliseconds(this:Date)
    this.intrinsic::getMilliseconds();

prototype function getUTCMilliseconds(this:Date)
    this.intrinsic::getUTCMilliseconds();

prototype function getTimezoneOffset(this:Date)
    this.intrinsic::getTimezoneOffset();

prototype function setTime(this:Date, t)
    this.intrinsic::setTime(double(t));
```

```
prototype function setMilliseconds(this:Date, ms)
    this.intrinsic::setMilliseconds(double(ms))

prototype function setUTCMilliseconds(this:Date, ms)
    this.intrinsic::setUTCMilliseconds(double(ms));

prototype function setSeconds(this:Date, sec, ms = getMilliseconds())
    this.intrinsic::setSeconds(double(sec), double(ms));

prototype function setUTCSeconds(this:Date, sec, ms = getUTCMilliseconds())
    this.intrinsic::setUTCSeconds(double(sec), double(ms));

prototype function setMinutes(this:Date, min, sec = getSeconds(), ms = getMilliseconds())
    this.intrinsic::setMinutes(double(min), double(sec), double(ms));

prototype function setUTCMinutes(this:Date, min, sec = getUTCSeconds(), ms =
getUTCMilliseconds())
    this.intrinsic::setUTCMinutes(double(min), double(sec), double(ms));

prototype function setHours(this:Date, hour, min=getMinutes(), sec=getSeconds(),
ms=getMilliseconds())
    this.intrinsic::setHours(double(hour), double(min), double(sec), double(ms));

prototype function setUTCHours(this:Date,
                               hour,
                               min=getUTCMinutes(),
                               sec=getUTCSeconds(),
                               ms=getUTCMilliseconds())
    this.intrinsic::setUTCHours(double(hour), double(min), double(sec), double(ms));

prototype function setDate(this:Date, date)
    this.intrinsic::setDate(double(date));

prototype function setUTCDate(this:Date, date)
    this.intrinsic::setUTCDate(double(date));

prototype function setMonth(this:Date, month, date=getDate())
    this.intrinsic::setMonth(double(month), double(date));

prototype function setUTCMonth(this:Date, month, date=getUTCDate())
    this.intrinsic::setUTCMonth(double(month), double(date));

prototype function setFullYear(this:Date, year, month=getMonth(), date=getDate())
    this.intrinsic::setFullYear(double(year), double(month), double(date));

prototype function setUTCFullYear(this:Date, year, month=getUTCMonth(), date=getUTCDate())
    this.intrinsic::setUTCFullYear(double(year), double(month), double(date));
```

8 Error classes

```
FILE:                         spec/library/Error.html
DRAFT STATUS:                 DRAFT 1 - ROUGH
REVIEWED AGAINST ES3:         NO
REVIEWED AGAINST PROPOSALS:   NO
REVIEWED AGAINST CODE:        NO
```

1 ECMAScript provides a hierarchy of standard native error classes rooted at the class `Error` (see class Error).

2 The ECMAScript implementation throws a new instance of one of the native error classes when it detects certain run-time errors. The conditions under which run-time errors are detected are explained throughout this Standard. The description of each of the native error objects contains a summary of the conditions under which an instance of that particular error class is thrown.

3 The class `Error` serves as the base class for all the classes describing standard errors thrown by the ECMAScript implementation: `EvalError`, `RangeError`, `ReferenceError`, `SyntaxError`, `TypeError`, and `URIError`. (See class EvalError, class RangeError, class ReferenceError, class SyntaxError, class TypeError, class URIError.)

4 The class `Error` as well as all its native subclasses are non-final and dynamic and may be subclassed by user-defined exception classes.

5 All the built-in subclasses of `Error` share the same structure.

9 The class `Error`

1 The class `Error` is a dynamic non-final subclass of `Object`. Instances of `Error` are not thrown by the implementation; rather, `Error` is intended to serve as a base class for other error classes whose instances represent specific classes of run-time errors.

9.1 Synopsis

1 The class `Error` provides the following interface:

```
dynamic class Error extends Object
{
    function Error(message) …
    meta static function invoke(message) …

    static const length = 1

    override intrinsic function toString() …
}
```

2 The `Error` prototype object provides these direct properties:

```
    toString: function () … ,
    name:     "Error" ,
    message:  … ,
```

9.2 Methods on the `Error` class

9.2.1 new Error (message)

Description

1 When the `Error` constructor is called as part of a `new Error` expression it initialises the newly created object: If *message* is not **undefined**, the dynamic `message` property of the newly constructed `Error` object is set to `string(`*message*`)`.

Implementation

```
function Error(message) {
    if (message !== undefined)
        this.public::message = string(message);
}
```

9.2.2 Error (message)

Description

1 When the `Error` class object is called as a function, it creates and initialises a new `Error` object by invoking the `Error` constructor.

Returns

2 The `Error` class object called as a function returns a new `Error` object.

Implementation

```
meta static function invoke(message)
    new Error(message);
```

9.3 Methods on `Error` instances

9.3.1 intrinsic::toString( )

Description

1 The intrinsic `toString` method converts the `Error` object to an implementation-defined string.

Returns

2 A `string` object.

Implementation

3 The intrinsic `toString` method is implementation-dependent.

9.4 Methods on the `Error` prototype object

9.4.1 toString ( )

Description

1 The prototype `toString` method calls the intrinsic `toString` method.

Returns

2 The prototype `toString` method returns a `string` object.

Implementation

```
prototype function toString()
    this.intrinsic::toString();
```

9.5 Value properties on the `Error` prototype object

9.5.1 message

1 The initial value of the `message` prototype property is an implementation-defined string.

10 The class `EvalError`

1 The implementation throws a new `EvalError` instance when it detects that the global function `eval` was used in a way that is incompatible with its definition. See XX.XX.

FIXME Clean up the section references when we reach final draft.

10.1 Synopsis

1 The `EvalError` class provides this interface:

```
dynamic class EvalError extends Error
{
    function EvalError(message) …
    meta static function invoke(message) …

    static const length = 1
}
```

2 The `EvalError` prototype object provides these direct properties:

```
    name:    "EvalError" ,
    message: … ,
```

10.2 Methods on the `EvalError` class

10.2.1 new EvalError (message)

Description

1 When the `EvalError` constructor is called as part of a `new EvalError` expression it initialises the newly created object by delegating to the `Error` constructor.

Implementation

```
function EvalError(message)
    : super(message)
{
}
```

10.2.2 EvalError (message)

Description

1 When the `EvalError` class object is called as a function, it creates and initialises a new `EvalError` object by invoking the `EvalError` constructor.

Returns

2 The `EvalError` class object called as a function returns a new `EvalError` object.

Implementation

```
meta static function invoke(message)
    new EvalError(message);
```

10.3 Value properties on the `EvalError` prototype object

10.3.1 message

1 The initial value of `message` prototype property is an implementation-defined string.

11 The class `RangeError`

1 The implementation throws a new `RangeError` instance when it detects that a numeric value has exceeded the allowable range. See 15.4.2.2, 15.4.5.1, 15.7.4.5, 15.7.4.6, and 15.7.4.7.

FIXME Clean up the section references when we reach final draft.

11.1 Synopsis

1 The `RangeError` class provides this interface:

```
dynamic class RangeError extends Error
{
    function RangeError(message) …
    meta static function invoke(message) …

    static const length = 1
}
```

2 The `RangeError` prototype object provides these direct properties:

```
    name:    "RangeError" ,
    message: … ,
```

11.2 Methods on the `RangeError` class

11.2.1 new RangeError (message)

Description

1 When the `RangeError` constructor is called as part of a `new RangeError` expression it initialises the newly created object by delegating to the `Error` constructor.

Implementation

```
function RangeError(message)
    : super(message)
{
}
```

11.2.2 RangeError (message)

Description

1 When the `RangeError` class object is called as a function, it creates and initialises a new `RangeError` object by invoking the `RangeError` constructor.

Returns

2 The `RangeError` class object called as a function returns a new `RangeError` object.

Implementation

```
meta static function invoke(message)
    new RangeError(message);
```

11.3 Value properties on the `RangeError` prototype object

11.3.1 message

1 The initial value of `message` prototype property is an implementation-defined string.

12 The class `ReferenceError`

1 The implementation throws a new `ReferenceError` instance when it detects an invalid reference value. See 8.7.1, and 8.7.2.

FIXME Clean up the section references when we reach final draft.

12.1 Synopsis

```
dynamic class ReferenceError extends Error
{
    function ReferenceError(message) …
    meta static function invoke(message) …

    static const length = 1
}
```

1 The `ReferenceError` prototype object provides these direct properties:

```
    name:    "ReferenceError" ,
    message: … ,
```

12.2 Methods on the `ReferenceError` class

12.2.1 new ReferenceError (message)

Description

1 When the `ReferenceError` constructor is called as part of a `new ReferenceError` expression it initialises the newly created object by delegating to the `Error` constructor.

Implementation

```
function ReferenceError(message)
    : super(message)
{
}
```

12.2.2 ReferenceError (message)

Description

1 When the `ReferenceError` class object is called as a function, it creates and initialises a new `ReferenceError` object by invoking the `ReferenceError` constructor.

Returns

2 The `ReferenceError` class object called as a function returns a new `ReferenceError` object.

Implementation

```
meta static function invoke(message)
    new ReferenceError(message);
```

12.3 Value properties on the `ReferenceError` prototype object

12.3.1 message

1 The initial value of `message` prototype property is an implementation-defined string.

13 The class `SyntaxError`

1 The implementation throws a new `SyntaxError` instance when a parsing error has occurred. See 15.1.2.1, 15.3.2.1, 15.10.2.5, 15.10.2.9, 15.10.2.15, 15.10.2.19, and 15.10.4.1.

FIXME Clean up the section references when we reach final draft.

13.1 Synopsis

```
dynamic class SyntaxError extends Error
{
    function SyntaxError(message) …
    meta static function invoke(message) …

    static const length = 1
}
```

1 The `SyntaxError` prototype object provides these direct properties:

```
    name:    "SyntaxError" ,
    message: … ,
```

13.2 Methods on the `SyntaxError` class

13.2.1 new SyntaxError (message)

Description

1 When the `SyntaxError` constructor is called as part of a `new SyntaxError` expression it initialises the newly created object by delegating to the `Error` constructor.

Implementation

```
function SyntaxError(message)
    : super(message)
{
}
```

13.2.2 SyntaxError (message)

Description

1 When the `SyntaxError` class object is called as a function, it creates and initialises a new `SyntaxError` object by invoking the `SyntaxError` constructor.

Returns

2 The `SyntaxError` class object called as a function returns a new `SyntaxError` object.

Implementation

```
meta static function invoke(message)
    new SyntaxError(message);
```

13.3 Value properties on the `SyntaxError` prototype object

13.3.1 message

1 The initial value of `message` prototype property is an implementation-defined string.

14 The class `TypeError`

1 The implementation throws a new `TypeError` instance when it has detected that the actual type of an operand is different than the expected type. See 8.6.2, 8.6.2.6, 9.9, 11.2.2, 11.2.3, 11.8.6, 11.8.7, 15.3.4.2, 15.3.4.3, 15.3.4.4, 15.3.5.3, 15.4.4.2, 15.4.4.3, 15.5.4.2, 15.5.4.3, 15.6.4, 15.6.4.2, 15.6.4.3, 15.7.4, 15.7.4.2, 15.7.4.4, 15.9.5, 15.9.5.9, 15.9.5.27, 15.10.4.1, and 15.10.6.

FIXME Clean up the section references when we reach final draft.

14.1 Synopsis

```
dynamic class TypeError extends Error
{
    function TypeError(message) …
    meta static function invoke(message) …

    static const length = 1
}
```

1 The `TypeError` prototype object provides these direct properties:

```
    name:    "TypeError" ,
    message: … ,
```

14.2 Methods on the `TypeError` class

14.2.1 new TypeError (message)

Description

1 When the `TypeError` constructor is called as part of a `new TypeError` expression it initialises the newly created object by delegating to the `Error` constructor.

Implementation

```
function TypeError(message)
    : super(message)
{
}
```

14.2.2 TypeError (message)

Description

1 When the `TypeError` class object is called as a function, it creates and initialises a new `TypeError` object by invoking the `TypeError` constructor.

Returns

2 The `TypeError` class object called as a function returns a new `TypeError` object.

Implementation

```
meta static function invoke(message)
    new TypeError(message);
```

14.3 Value properties on the `TypeError` prototype object

14.3.1 message

1 The initial value of `message` prototype property is an implementation-defined string.

15 The class URIError

1 The implementation throws a new URIError when one of the global URI handling functions was used in a way that is incompatible with its definition. See 15.1.3.

FIXME Clean up the section references when we reach final draft.

15.1 Synopsis

```
dynamic class URIError extends Error
{
    function URIError(message) …
    meta static function invoke(message) …

    static const length = 1
}
```

1 The URIError prototype object provides these direct properties:

```
    name:    "URIError" ,
    message: … ,
```

15.2 Methods on the URIError class

15.2.1 new URIError (message)

Description

1 When the URIError constructor is called as part of a new URIError expression it initialises the newly created object by delegating to the Error constructor.

Implementation

```
function URIError(message)
    : super(message)
{
}
```

15.2.2 URIError (message)

Description

1 When the URIError class object is called as a function, it creates and initialises a new URIError object by invoking the URIError constructor.

Returns

2 The URIError class object called as a function returns a new URIError object.

Implementation

```
meta static function invoke(message)
    new URIError(message);
```

15.3 Value properties on the URIError prototype object

15.3.1 message

1 The initial value of message prototype property is an implementation-defined string.