Ecma/TC39/2008/057

1 Introduction

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.

2 ECMAScript execution environments may provide multiple global objects, each of which may be accessible from the others. Whether each of these global objects has separate intial values for the initial properties described in this section, or whether the values are shared, is implementation-defined.

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.

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.

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.

IMPLEMENTATION 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, and every built-in class object with a `meta::invoke` method, 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 object that does not have a `meta::invoke` method 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.

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.

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
    private::toString();
```

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

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

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.

NOTE In effect, the `intrinsic` namespace is open for all built-in code, and this namespace takes precedence over the `public` namespace.

19 In some cases the built-in functions construct new error objects that are then thrown as exceptions. For purposes of documentation an informative string is passed to the constructors of the error objects. These strings are never to be considered normative.

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 callable 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.

2.4 Special cases

1 This is a list of all the special cases I'm aware of in the sections following.

- Object.prototype.`[[Prototype]]` is null
- Math.`[[Prototype]]` does not have a constructor
- Math is an instance of a class that is not constructable through the meta-objects system
- double.prototype === Number.prototype and double.`[[Prototype]]` === Number.`[[Prototype]]`
- decimal.prototype === Number.prototype and decimal.`[[Prototype]]` === Number.`[[Prototype]]`
- string.prototype === String.prototype and string.`[[Prototype]]` === String.`[[Prototype]]`
- boolean.prototype === Boolean.prototype and boolean.`[[Prototype]]` === Boolean.`[[Prototype]]`

3 The Global Object

1 The global object is an instance of an implementation-dependent class. In particular, the name of this class and the contents of the class's prototype object are implementation-dependent.

2 The class describing the global object does not have an accessible constructor function; it is not possible to use the global object as a constructor with the `new` operator.

3 The class describing the global object does not have a `meta::invoke` method; it is not possible to call the global object as a function.

3.1 Synopsis

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

```
namespace __ES4__

__ES4__ namespace intrinsic = …
__ES4__ namespace iterator = …
__ES4__ namespace reflect = …
__ES4__ namespace meta = …

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 double …
__ES4__ class decimal …
__ES4__ class Name …
__ES4__ class Namespace …
__ES4__ class Map.<K,V> …
__ES4__ class Vector.<T> …
```

```
__ES4__ type EnumerableId = …
__ES4__ type AnyNumber = …
__ES4__ type AnyString = …
__ES4__ type AnyBoolean = …
__ES4__ type Callable = …

intrinsic const function eval(s: string) …
intrinsic const function parseInt(s: string, r: double=0): AnyNumber …
intrinsic const function parseFloat(s: string): AnyNumber …
intrinsic const function isNaN(n: AnyNumber): boolean …
intrinsic const function isFinite(n: AnyNumber): boolean …
intrinsic const function isIntrinsic(n: AnyNumber): boolean …
intrinsic const function isInt(n: AnyNumber): boolean …
intrinsic const function isUint(n: AnyNumber): boolean …
intrinsic const function toInt(n: AnyNumber): double …
intrinsic const function toUint(n: AnyNumber): double …
intrinsic const function decodeURI(s: string): string …
intrinsic const function decodeURIComponent(s: string): string …
intrinsic const function encodeURI(s: string): string …
intrinsic const function encodeURIComponent(s: string): string …
intrinsic const function hashcode(x): double …

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 isInt(n: AnyNumber): boolean …
__ES4__ function isUint(n: AnyNumber): boolean …
__ES4__ function toInt(n: AnyNumber): double …
__ES4__ function toUint(n: AnyNumber): double …

const NaN: double = …
const Infinity: double = …
const undefined: undefined = …
const Math: helper::MathType = …
const __ECMASCRIPT_VERSION__: double = …
__ES4__ const global = …
```

3.2 Namespace Properties on the Global Object

COMPATIBILITY NOTE The namespace properties are all new in the 4th Edition of this Standard.

3.2.1 __ES4__

1 The namespace `__ES4__` is used to tag all names introduced in the global object in the 4th Edition of this Standard, except for two: `__ES4__` and `__ECMASCRIPT_VERSION__`.

2 The namespace `__ES4__` 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.

COMPATIBILITY NOTE The risk of polluting the name space for 3rd Edition code with new names is deemed too great to always open the `__ES4__` name space.

3 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.)

3.2.2 intrinsic

1 The namespace `intrinsic` is used to tag pre-defined types, properties, and methods.

2 The namespace `intrinsic` is reserved by the language. Except in the case where a method tagged `intrinsic` overrides an intrinsic method inherited from a pre-defined class, it is an error for user code to introduce new bindings in the `intrinsic` namespace.

3 The bindings in the `intrinsic` namespace are always constant fixtures.

NOTE A *fixture* is a binding that is not deletable and which takes precedence over dynamic names during lexical name lookup.

3.2.3 reflect

1 The namespace `reflect` is used to tag pre-defined interfaces in the reflection subsystem.

2 The namespace `reflect` is reserved by the language. It is an error for user code to introduce new bindings in the `reflect` namespace.

3.2.4 meta

1 The namespace `meta` is used to tag methods that participate in the language's protocols for invocation and property access.

2 The namespace `meta` is reserved by the language. Except in the case where a class definition uses it to tag (possibly static) methods called `invoke`, `get`, `set`, `has`, or `delete`, it is an error for user code to introduce new bindings in the `meta` namespace.

3.2.5 iterator

1 The namespace `iterator` is used for the iteration protocol, which is defined elsewhere. (See iterators.)

NOTE Unlike the namespaces `__ES4__`, `intrinsic`, `reflect`, and `meta`, the namespace `iterator` is not reserved by the system.

3.3 Value Properties on the Global Object

3.3.1 __ECMASCRIPT_VERSION__

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

COMPATIBILITY NOTE This property is new in the 4th Edition of this Standard. It is one of two properties introduced in the 4th Edition of this Standard that is not in the `__ES4__` namespace. (The other is `__ES4__`.)

3.3.2 NaN

1 The value of the constant property `NaN` is **NaN** (see nan-value).

Implementation

```
const NaN : double = …
```

COMPATIBILITY NOTE `NaN` was not constant in the 3rd Edition of this Standard.

3.3.3 Infinity

1 The value of `Infinity` is $+\infty$ (see infinity-value).

Implementation

```
const Infinity : double = …
```

COMPATIBILITY NOTE `Infinity` was not constant in the 3rd Edition of this Standard.

3.3.4 undefined

1 The value of `undefined` is **undefined** (see undefined-value).

Implementation

```
const undefined : undefined = …
```

COMPATIBILITY NOTE `undefined` was not constant in the 3rd Edition of this Standard.

3.3.5 Math

1 The value of `Math` is the Math object (see math-object).

Implementation

```
const Math : helper::MathType = …
```

COMPATIBILITY NOTE `Math` was not constant in the 3rd Edition of this Standard.

2 The helper type `MathType` (see MathType) is a structural record type that includes a property for every intrinsic method and public constant property defined on the Math object.

NOTE The type of `Math` impacts strict mode type checking.

3.3.6 global

1 The value of `global` is the global object that contains the property `global`.

NOTE There may be multiple global objects in a program, and 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.

Implementation

```
const global = …
```

COMPATIBILITY NOTE `global` is new in the 4th Edition of this Standard.

3.4 Function Properties of the Global Object

3.4.1 eval

1 *It is likely that the description in this section needs to be broken up and scattered over several parts of the final specification, but for the time being it's best if everything is centralized here. I've added more expository and background material than the spec really ought to have; we'll clean this up by and by. --lars*

3.4.1.1 Overview and background

1 The global object has properties named `eval` and `intrinsic::eval`. Those properties initially hold the same value, a function, and that function -- the `eval` function -- can be called in all the ways that any other function in the language can be called. There are however some run-time restrictions (described below) on when those calls are valid.

2 In addition, there is an operator in the language that is also known under the names `eval` and `intrinsic::eval`. The `eval` operator has access to the lexical environment of its context; it can look up and introduce bindings in the environment of its context.

3 Together, the `eval` function and the `eval` operator provide run-time evaluation functionality that (a) is compatible with the functionality mandated for `eval` by the 3rd Edition and (b) handles all important known uses of `eval` on the web.

4 The 3rd Edition only mandates the equivalent of the operator form of the 4th Edition, yet it describes `eval` as a function that has the ability to inspect and modify its caller's lexical environment. In practice, an implementation that supports only the functionality mandated by the 3rd Edition will not support the web well, and as a consequence several implementations of ECMAScript in web browsers provide `eval` as a true function that actually has the ability to inspect and modify its caller's environment, no matter who the caller of `eval` is and regardless of the name under which `eval` has been called.

5 In other words, the 3rd Edition form of `eval`, implemented in full generality, makes it generally impossible to know if any particular scope contains a binding for any particular name, since any function call in the scope may be a call to `eval`, which may introduce new names in the scope.

6 As an illustration, the following program prints "20" in Mozilla Firefox, even though casual inspection of the program would lead one to conclude that the `x` referenced in the body of `g` is the constant binding in the outer scope:

```
const x = 10;
function g(f,s) {
    f(s);
    return x;
}

document.writeln(g(eval,"var x=20"));
```

7 In conclusion, the primary reason for splitting the definition of `eval` into function and operator forms in the 4th Edition is in order to be able to control the extent to which `eval` can introduce new names in scopes, while at the same time remaining compatible with existing programs.

8 The secondary reason for splitting the definition of `eval` into function and operator forms is that the behavior illustrated above largely precludes some standard code generation strategies. A typical approach in lexically scoped languages is to translate variable references at compile time to *(rib,offset)* pairs; at run-time, scope object number *rib* (where the innermost is number zero) is fetched and property number *offset* is fetched out of it. That approach is only possible if the environment is known at compile time. If there is a chance that `eval` can introduce new names into a scope at run-time then lookup in that scope must always be by name; with an operator form of `eval` it is possible to know at compile time whether a scope may be thus affected.

9 Several other features and clarifications have been incorporated into the 4th Edition in an attempt to constrain the effects of `eval`. While the splitting of `eval` into an operator form and a function form makes it clearer when a non-global environment might have new bindings introduced, it does not prevent such bindings from being introduced.

10 The additional features and clarifications are:

- If the version number passed as the second argument to `eval` is greater than 3, then the program being evaluated is given a fresh variable object in which it can create bindings; as a consequence, no bindings can be introduced into the caller's environment (except by assignment to non-existent global variables).
- `eval` is prevented from changing the DontDelete attribute on existing bindings when a binding form is evaluated.
- The operator form of `eval` is disallowed inside classes.
- The operator form of `eval` is disallowed in strict mode.

3.4.1.2 eval (program, version=...)

Description

1 The eval function and the eval operator (described fully below) are invoked on a *program*, which is a value of any type, and a *version*, which is intended to be a nonnegative integer (defaulting to 3).

2 If the *program* is a string then it must represent valid source code according to the nonterminal *Program* (see Grammar), with the proviso that the keyword set recognized during lexical analysis is determined by *version*, as follows. Convert version to an integer as with the ToInt32 operation. If the converted value of *version* is 3 or less then the keyword set is the set of reserved words in the 3rd Edition of this Standard (E262-3 section 7.5.2). Otherwise, if the converted value of *version* is *n* then the keyword set is the set of reserved and contextually reserved words in the *n*th Edition of this Standard.

NOTE Program arguments to `eval` that use e.g. `let` as an identifier will continue to work in a 4th Edition implementation (where `let` is a keyword) as long as no *version* is passed to `eval`, or the value of the version passed is 3 or less.

3 If the converted value of *version* is 4 or greater then the evaluation takes place in a fresh variable object.

NOTE In other words, `eval` will be unable to introduce bindings in its caller's variable object if *version* is 4 or greater.

COMPATIBILITY NOTE Unlike in the 3rd Edition of this Standard, `eval` is not allowed to change a property from being DontDelete to being deletable. That restriction belongs in the section on adding bindings to the variable object (10.1.3 in 3rd Ed) and is only mentioned here for the time being.

Returns

4 If the *program* is a string then the result of compiling and evaluating *program* as a *Program* is returned. Otherwise, *program* is returned unchanged.

Implementation

```
function eval( program, version=3 ) …
```

3.4.1.3 The `eval` operator

1 *There are two possible designs for the operator form. One is that an expression of the form* `eval(s)` *is* always *taken as the operator form, regardless of the binding of* `eval` *in the context of the expression; the other is that an expression of that form* may *be the operator form, and that it is the operator form only if the binding of* `eval` *is the original, global binding. Since the former design would be incompatible with 3rd Edition, we use the latter.*

2 It will always be lexically apparent when `eval` is *possibly* being used as an operator, but in the general case it is not possible to determine until run-time whether it is *actually* being used as an operator.

NOTE For the intrinsic form of the operator it is possible to determine this at compile time, in the absence of the use of `with` in the enclosing context.

Description

3 The operator form is *possibly* being used in an expression *E* if *E* has the form of a *CallExpression* (including the parentheses bracketing the arguments) and the *MemberExpression* that denotes the function to be called has the form of the unqualified identifier `eval` or the qualified identifier `intrinsic::eval`. That is, apart from any superflous parentheses, *E* has the form *M(P, ...)*.

4 If the possible use of the operator form of `eval` appears in any context inside a class, a **SyntaxError** is thrown.

5 If the possible use of the operator form of `eval` appears in any context inside block in which strict mode is in effect (even inside a block that overrides strict mode by decreeing standard mode), a **SyntaxError** is thrown.

NOTE The purpose of these restriction is partly to avoid inconsistencies (in strict mode) and partly to signal that the use of the `eval` operator is potentially harmful to program integrity.

FIXME A less restrictive, but probably equally safe, alternative would be to allow `eval` to be used inside a class provided that a version parameter was being passed and its value was a constant known to be 4 or greater.

6 If the *program* operand to `eval` contains definitions for classes, interfaces, packages, namespaces, types, or units, then a `SyntaxError` is thrown (even if the use of the `eval` operator is at the outer program's top level where these forms would normally be allowed).

FIXME Should that be EvalError?

FIXME It seems it would not be entirely unreasonable to lift that restriction if the *version* is greater than 3, but it's unclear as yet what the problems might be with e.g. allowing classes to have elaborate scope chains around them.

Returns

7 The eval operator returns an ECMAScript value.

Implementation

8 In the case that a possible use of the operator form is detected, *M(P, ...)* is evaluated as follows.

9 (The implementation of the `eval` operator is presented as pseudo-code because it is not expressible in ECMAScript. Eventually, it may be presented as Standard ML code.)

```
look up M in the environment yielding the value V
if V is the pre-defined eval function and
   the binding object O holding V is an ES global object and
   the global object on the scope chain of V is O then
     invoke eval as follows:
       evaluate the P in order to yield argument value A
       if there are no A values, then
         return undefined
       if the first A value is not a string, then
         return the first A
       if there is a second A then let K=int(A), else let K=3
       evaluate the program denoted by the first A as follows:
         if K <= 3
           the scope chain is the lexical chain in effect at
               the point of invocation of M
           the variable object is the innermost variable object in effect
               (which is to say that it excludes binding objects
               introduced for "let", "catch", named function expressions,
               and "switch type")
         else
           the scope chain is the lexical chain in effect at the point of
               invocation of M, extended by a new variable object W
           the variable object is W
         fi
         the value of "this" is the global object O
         the keyword set is determined by K
else
  evaluate the P in order to yield a list of arguments A
  invoke V as a function on the arguments A
fi
```

NOTE The requirement that the global object of V be O precludes sharing of `eval` functions among multiple global environments. It is possible that that requirement is not actually needed for consistent operation.

3.4.1.4 The `eval` function

1 If the operator form is not detected syntactically then `eval` is either being called as a function under a different name or it is being invoked as a method on an object. The implementation does not need to handle this case syntactically; it is handled as a regular function call.

2 The following description applies to both the public `eval` function and the intrinsic `eval` function, both defined in the global object.

Description

3 The global `eval` function evaluates its first argument as a program in the global scope of the `eval` function.

Returns

4 The global `eval` function returns the value computed by the program that is evaluated, or its first argument if the first argument is not a string.

Implementation

5 The body of the pre-defined `eval` function *V* is evaluated in the context of a list of argument values *A* as follows.

6 (The implementation of the `eval` function is presented as pseudo-code because it is not expressible in ECMAScript. Eventually, it may be presented as ECMAScript code with the addition of "magic" run-time system hooks.)

```
if the "this" object O is an ES global object and
   the global object on the scope chain of V is O then
     if there are no A values, then
       return undefined
     if the first A value is not a string, then
       return the first A
     if there is a second A then let K=int(A), else let K=3
     evaluate the program denoted by the first A as follows:
       if K<=3 then
         the scope chain holds O only
         the variable object is O
       else
         the scope chain holds O extended by a new variable object W
         the variable object is W
       fi
       the value of "this" is O
       the keyword set is determined by K
else
  throw EvalError
fi
```

NOTE The requirement that the global object of V be O precludes sharing of `eval` functions among multiple global environments. It is possible that that requirement is not actually needed for consistent operation.

3.4.1.5 Restrictions on the use of the `eval` property

1 If the global `eval` property is assigned to, an `EvalError` exception may be thrown.

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, the radix 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 const function parseInt(s: string, r: double=0): AnyNumber {
    let i;

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

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

    let maybe_hexadecimal = false;
    r = intrinsic::toInt(r);
    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.intrinsic::substring(2);
    }

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

    if (s == "")
        return NaN;
    return sign * 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 helper function `isTrimmableSpace(c)` computes whether `c` is a space character that can be trimmed off the beginning of the string, see helper:isTrimmableSpace.

5 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.intrinsic::toUpperCase();
    if (c >= '0' && c <= '9')
        return (c.intrinsic::charCodeAt(0) - '0'.intrinsic::charCodeAt(0)) < r;
    if (c >= 'A' && c <= 'Z')
        return (c.intrinsic::charCodeAt(0) - 'A'.intrinsic::charCodeAt(0) + 10) < r;
    return false;
}
```

3.4.3 parseInt (s, r=…)

Description

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

Returns

2 The `parseInt` function returns a number.

Implementation

```
public function parseInt(s, r=0)
    intrinsic::parseInt(string(s), double(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 const 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

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

3.4.6 intrinsic::isNaN (n)

Description

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

Returns

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

Implementation

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

3.4.7 isNaN (x)

Description

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

Returns

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

Implementation

```
public function isNaN(x)
    intrinsic::isNaN(Number(x));
```

3.4.8 intrinsic::isFinite (n)

Description

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

Returns

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

Implementation

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

3.4.9 isFinite (x)

Description

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

Returns

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

Implementation

```
public function isFinite(x)
    intrinsic::isFinite(Number(x));
```

3.4.10 intrinsic::isIntegral (n)

Description

1 The intrinsic `isIntegral` function tests whether a numeric value *n* is integral (a finite integer).

Returns

2 The intrinsic `isIntegral` function returns **true** if *n* is integral, and otherwise returns **false**.

Implementation

```
intrinsic const function isIntegral(v:AnyNumber): boolean
    intrinsic::isFinite(v) && Math.intrinsic::floor(v) == v;
```

3.4.11 isIntegral (x)

Description

1 The `isIntegral` function converts its argument *x* to a number, then calls the intrinsic `isIntegral` function on the converted value.

Returns

2 The `isIntegral` function returns **true** if *x* converted to a number is integral, and otherwise returns **false**.

Implementation

```
__ES4__ function isIntegral(v)
    intrinsic::isIntegral(Number(v));
```

3.4.12 intrinsic::isInt (n)

Description

1 The intrinsic `isInt` function tests whether a numeric value *n* is an `int` value (a finite integer in the range -2^{31} to $2^{31}-1$, inclusive).

Returns

2 The intrinsic `isInt` function returns **true** if *n* is an `int` value, and otherwise returns **false**.

Implementation

```
intrinsic const function isInt(n:AnyNumber) : boolean
    intrinsic::isIntegral(n) && n >= -0x7FFFFFFF && n <= 0x7FFFFFFF;
```

3.4.13 isInt (x)

Description

1 The `isInt` function converts its argument *x* to a number, then calls the intrinsic `isInt` function on the converted value.

Returns

2 The `isInt` function returns **true** if *x* converted to a number is an `int` value, and otherwise returns **false**.

Implementation

```
__ES4__ function isInt(x)
    intrinsic::isInt(Number(x));
```

3.4.14 intrinsic::isUint (n)

Description

1 The intrinsic `isUint` function tests whether the numeric value *n* is a `uint` value (a finite integer in the range 0 to $2^{32}-1$, inclusive).

Returns

2 The intrinsic `isUint` function returns **true** if *n* is a `uint` value, and otherwise returns **false**.

Implementation

```
intrinsic const function isUint(n:AnyNumber) : boolean
    intrinsic::isIntegral(n) && n >= 0 && n <= 0xFFFFFFFF;
```

3.4.15 isUint (x)

Description

1 The `isUint` function converts its argument *x* to a number, then calls the intrinsic `isUint` function on the converted value.

Returns

2 The `isUint` function returns **true** if *x* converted to a number is a `uint` value, and otherwise returns **false**.

Implementation

```
__ES4__ function isUint(x)
    intrinsic::isUint(Number(x));
```

3.4.16 intrinsic::toInt (n)

Description

1 The intrinsic `toInt` function converts its argument *n* to an `int` value using the ToInt32 algorithm (see ToInt32).

Returns

2 The intrinsic `toInt` function returns an `int` value.

Implementation

```
intrinsic const function toInt(n:AnyNumber) : double
    n | 0;
```

3.4.17 toInt (x)

Description

1 The `toInt` function converts its argument *x* to a number, then calls the intrinsic `toInt` function on the converted value.

Returns

2 The `toInt` function returns an `int` value.

Implementation

```
__ES4__ function toInt(x)
    intrinsic::toInt(Number(x));
```

3.4.18 intrinsic::toUint (n)

Description

1 The intrinsic `toUint` function converts its argument *n* to a `uint` value using the ToUint32 algorithm (see ToUint32).

Returns

2 The intrinsic `toUint` function returns a `uint` value.

Implementation

```
intrinsic const function toUint(n:AnyNumber) : double
    n >>> 0;
```

3.4.19 toUint (x)

Description

1 The `toUint` function converts its argument *x* to a number, then calls the intrinsic `toUint` function on the converted value.

Returns

2 The `toUint` function returns a `uint` value.

Implementation

```
__ES4__ function toUint(x) : boolean
    intrinsic::toUint(Number(x));
```

3.4.20 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 (Ticket #170.) Upgrade to Unicode 5 in the following sections, and upgrade to handling the entire (21-bit) 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.intrinsic::indexOf(C) != -1) {
            R = R + C;
            k = k + 1;
            continue;
        }

        let V = C.intrinsic::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].intrinsic::charCodeAt(0);
            V = (V - 0xD800) * 0x400 + (V2 - 0xDC00) + 0x10000;
        }

        let octets = toUTF8(V);
        for ( let j=0 ; j < octets.length ; j++ )
            R = R + "%" + 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`.

```
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 = decodeHexEscape(s, k);
        k = k + 3;

        if ((B & 0x80) == 0) {
            let C = string.intrinsic::fromCharCode(B);
            if (reservedSet.intrinsic::indexOf(C) != -1)
                R = R + s.intrinsic::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 = decodeHexEscape(s, k);
            if ((B & 0xC0) != 0x80)
                throw new URIError(/* Invalid encoded character */);
            k = k + 3;
            octets.intrinsic::push(B);
        }
        let V = 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.intrinsic::fromCharCode(H, L);
        }
        else {
            let C = string.intrinsic::fromCharCode(V);
            if (reservedSet.intrinsic::indexOf(C))
                R = R + s.intrinsic::substring(start, k);
            else
                R = R + C;
        }
    }
    return R;
}

helper function decodeHexEscape(s, k) {
    if (k + 2 >= s.length ||
        s[k] != "%" ||
        (!isDigitForRadix(s[k+1], 16) &&
         !isDigitForRadix(s[k+1], 16)))
        throw new URIError(/* Invalid escape sequence */);
    return intrinsic::parseInt(s.intrinsic::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) {
    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(/* Unconvertible 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 = uriAlpha + uriDigit + uriMark;
```

3.4.20.1 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 const function decodeURI(encodedURI: string)
    decode(encodedURI, uriReserved + "#");
```

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

3.4.20.2 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

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

3.4.20.3 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 const function decodeURIComponent(encodedURIComponent)
    decode(encodedURIComponent, "");
```

3.4.20.4 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

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

3.4.20.5 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, three, or four escape sequences representing the UTF-8 encoding of the character.

Returns

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

Implementation

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

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

3.4.20.6 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

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

3.4.20.7 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, three, or four escape sequences representing the UTF-8 encoding of the character.

Returns

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

Implementation

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

3.4.20.8 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

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

3.4.21 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 a nonnegative integer below 2^{32}.

Implementation

```
intrinsic const function hashcode(o): double {
    switch type (o) {
    case (x: null)       { return 0 }
    case (x: undefined)  { return 0 }
    case (x: AnyBoolean) { return Number(x) }
    case (x: AnyNumber)  { return intrinsic::toUint(x) }
    case (x: AnyString)  { return stringHash(x) }
    case (x: Namespace)  { return namespaceHash(x) }
    case (x: Name)       { return nameHash(x) }
    case (x: *)          { return objectHash(x) }
    }
}
```

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

6 The implementation should strive to compute different hashcodes for values 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.

3.5 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 `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`, `double`, and `decimal` classes are defined in sections class Number, 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 `Vector` class is defined in section class Vector

- 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 of all the nominal types that are treated as property names by the iteration protocol and the pre-defined objects:

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

FIXME Removed `int` and `uint` from this type if/when we agree that that's the right thing.

3.6.2 AnyNumber

1 The type `AnyNumber` is a union type of all the nominal types that are treated as numbers by the language:

```
__ES4__ type AnyNumber = (double|decimal|Number);
```

3.6.3 AnyString

1 The type `AnyString` is a union type of all the nominal types that are treated as strings by the language:

```
__ES4__ type AnyString = (string|String);
```

3.6.4 AnyBoolean

1 The type `AnyBoolean` is a union type of all the nominal types that are treated as booleans by the language:

```
__ES4__ type AnyBoolean = (boolean|Boolean);
```

3.6.5 Callable

1 The type `Callable` is a record type describing any object that can be called as a function:

```
__ES4__ type Callable = { meta::invoke: * }
```

4 The class `Object`

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:

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

    static public const length = 1

    intrinsic function toString() : string …
    intrinsic function toLocaleString() : string …
    intrinsic function valueOf() : Object …
    intrinsic function hasOwnProperty(name: EnumerableId): boolean …
    intrinsic function isPrototypeOf(value): boolean …
    intrinsic function propertyIsEnumerable(name: EnumerableId): boolean …
    intrinsic function __defineProperty__(name: EnumerableId, value,
enumerable:boolean=true, removable:boolean=true, writable:boolean=true): void …
}
```

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

```
    toString:             function () …
    toLocaleString:       function () …
    valueOf:              function () …
    hasOwnProperty:       function (name) …
    isPrototypeOf:        function (value) …
    propertyIsEnumerable: function (name) …
    __defineProperty__:   function (name, value, enumerable=undefined,
removable=undefined, writable=undefined) …
```

FIXME It is likely that `__defineProperty__` should become a static method on the `Object` object and that its parameters should be passed in some other way, for example as individual values fetched from `Object` (`Object.WRITABLE`, and so on) or as a set of bits or'ed together from bit values fetched from `Object`.

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 helper function `makeObject` that the ECMAScript implementation will invoke when it evaluates `new Object`.

4 The helper function `makeObject` only is 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.

```
helper 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 (x: (null|undefined)) {
        return magic::createObject();
    }
    case (o: Object) {
        return o;
    }
    }
}
```

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

```
static meta function invoke(value=undefined)
    new Object(value);
```

4.3 Methods on `Object` instances

1 The intrinsic methods on `Object` instances delegate to private methods that are also called by the prototype methods.

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"`, a single space character (U+0032), the class name of the object, and `"]"`.

Implementation

```
intrinsic function toString() : string
    private::toString();

private function toString(): string
    "[object " + getClassName() + "]";
```

3 The helper function `getClassName` returns the name for the class. This method is overridden in some of the pre-defined classes in order to provide backward compatibility with the 3rd Edition of this Standard: It is overridden by the class Error.

```
helper function getClassName()
    getClassName(this);
```

4 The function `helper::getClassName` extracts the class name from the object. See helper: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
    private::toLocaleString();

private function toLocaleString()
    this.toString();
```

4.3.3 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 (see 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 `valueOf` method returns an object value.

Implementation

```
intrinsic function valueOf() : Object
    private::valueOf();

private function valueOf(): Object
    this;
```

4.3.4 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]]` (see 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(name: EnumerableId): boolean
    hasOwnProperty(this, name);

private function hasOwnProperty(name: EnumerableId): boolean
    hasOwnProperty(this, name);
```

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

4 The helper function `toEnumerableId` returns its argument if it is one of the member types of `EnumerableId` (`int`, `uint`, `string`, and `Name`) and otherwise converts the argument to `string`.

```
helper function toEnumerableId(x) {
    switch type (x) {
    case (x: EnumerableId) { return x; }
    case (x: *)            { return string(x); }
    }
}
```

4.3.5 intrinsic::isPrototypeOf (value)

Description

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

Returns

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

Implementation

```
intrinsic function isPrototypeOf(value): boolean
    private::isPrototypeOf(value);

private function isPrototypeOf(value): boolean {
    if (!(value is Object))
        return false;

    let obj = value;
    while (true) {
        obj = getPrototype(obj);
        if (obj === null || obj === undefined)
            return false;
        if (obj === this)
            return true;
    }
}
```

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

4.3.6 intrinsic::propertyIsEnumerable (name)

Description

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

Implementation

```
intrinsic function propertyIsEnumerable(name: EnumerableId): boolean
    private::propertyIsEnumerable(name);
```

```
private function propertyIsEnumerable(name) {
    if (!hasOwnProperty(this, name))
        return false;
    return !getPropertyIsEnumerable(this, name);
}
```

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

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

4.3.7 intrinsic::__defineProperty__ (name, value, enumerable=…, removable=…, writable=…)

Description

1 The intrinsic `__defineProperty__` method creates a new dynamic property named *name* on this object, giving it the value *value* and attributes determined by the parameters *enumerable*, *removable*, and *writable*. If the property already exists, or if a non-writable property with the same name exists on an object in the prototype chain of this object, then a **TypeError** exception is thrown.

NOTE The name `__defineProperty__` (with the leading and trailing underscores) has been chosen in order to minimize the risk of name collisions with existing content on the web.

IMPLEMENTATION NOTE The name `__defineProperty__` mirrors the names of the non-standard methods `__defineGetter__` and `__defineSetter__` that are provided by some implementations. Those implementations may wish to extend the non-standard methods so that they provide control of at least enumerability and deletability in a manner compatible with `__defineProperty__`.

Returns

2 The intrinsic `__defineProperty__` method returns nothing.

Implementation

```
intrinsic function __defineProperty__(name: EnumerableId, value, enumerable:boolean=true,
removable:boolean=true, writable:boolean=true): void
    private::__defineProperty__(name, value, enumerable, removable, writable);

private function __defineProperty__(name, value, enumerable, removable, writable) {
    if (hasOwnProperty(this, name))
        throw new TypeError(/* Property exists */);

    let obj = getPrototype(this);
    while (obj != null) {
        if (hasOwnProperty(obj, name) && !getPropertyIsWritable(obj, name))
            throw new TypeError(/* non-Writable property in prototype chain */);
        obj = getPrototype(obj);
    }

    this[name] = value;
    setPropertyIsEnumerable(this, name, enumerable);
    setPropertyIsRemovable(this, name, removable);
    setPropertyIsWritable(this, name, writable);
}
```

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

4 The function `helper::getPrototype` extracts the `[[Prototype]]` property from the object. See helper:getPrototype.

5 The functions `helper::getPropertyIsDontEnum`, `helper::getPropertyIsDontDelete`, and `helper::getPropertyIsReadOnly` retrieve the attribute flags of the property. See helper:getPropertyIsDontEnum, helper:getPropertyIsDontDelete, and helper:getPropertyIsReadOnly.

6 The functions `helper::setPropertyIsDontEnum`, `helper::setPropertyIsDontDelete`, and `helper::setPropertyIsReadOnly` set the attribute flags of the property. See helper:setPropertyIsDontEnum, helper:setPropertyIsDontDelete, and helper:setPropertyIsReadOnly.

4.4 Methods on the `Object` prototype object

Description

1 The methods on the `Object` prototype object all perform simple type adjustments and then perform the same actions as the corresponding intrinsic methods.

Returns

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

Implementation

```
public prototype function toString()
    this.private::toString();

public prototype function toLocaleString()
    this.private::toLocaleString();

public prototype function valueOf()
    this.private::valueOf();

public prototype function hasOwnProperty(name)
    this.private::hasOwnProperty(toEnumerableId(name));

public prototype function isPrototypeOf(value)
    this.private::isPrototypeOf(value);

public prototype function propertyIsEnumerable(name)
    this.private::propertyIsEnumerable(toEnumerableId(name));

public prototype function __defineProperty__(name, value, enumerable=undefined,
removable=undefined, writable=undefined)
    this.private::__defineProperty__(toEnumerableId(name),
                                     value,
                                     enumerable === undefined ? true : boolean(enumerable)
,
                                     removable === undefined ? true : boolean(removable),
                                     writable === undefined ? true : boolean(writable));
```

5 The class `Function`

1 The class `Function` is a dynamic, non-final, direct 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` property can be called as a function.

4 The structural type `Callable` (see type:Callable) matches every object that has a `meta::invoke` property.

5.1 Synopsis

1 The class `Function` provides the following interface:

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

    static public function apply(fn: Callable, thisArg: Object=null, argArray:
Object=null) …
    static public function bind(method: Callable, thisObj: Object, ...args) …
    static public function call(fn: Callable, thisObj: Object=null, ...args) …

    static public const length = 1

    meta final function invoke( … ) …

    override intrinsic function toString() : string …

    intrinsic function apply(thisArg: Object=null, argArray: Object=null) …
    intrinsic function bind(thisObj: Object, ...args) …
    intrinsic function call(thisObj: Object=null, ...args) …

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

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

```
    meta::invoke: function () …
    length:       0
    toString:     function () …
    apply:        function(thisArg, argArray) …
    bind:         function(thisArg, ...args) …
    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 separated by "," 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 first character of the comma-separated concatenation of the p_i is a left parenthesis then the list of parameters must be parseable as a $FormalParameterList_{opt}$ enclosed in parentheses and optionally followed by a colon and a return type.

4 Otherwise, the list of parameters must be parsable as a $FormalParameterList_{opt}$.

5 If the list of parameters is not parseable as outlined in the previous two paragraphs, or if the body is not parsable as a *FunctionBody*, then a **SyntaxError** exception is thrown (see the grammar in section ECMAScript grammar).

6 Regardless of the form of the parameter list, it may include type annotations, default parameter values, and rest arguments.

FIXME It appears likely that the `Function` constructor needs to accept a version parameter so that the keyword set can be controlled, as is the case for `eval`.

Returns

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

Implementation

```
public function Function(...args)
    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);
    initializeFunction(this, __ES4__::global, parameters, body);
}
```

8 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.

9 A `prototype` object 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")
```

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)
    new Function(...args);
```

FIXME Ticket #357: That particular definition makes use of the prefix "spread" operator, which has not yet been formally accepted into the language.

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

Description

1 The static `apply` method takes arguments *fn*, *thisArg*, and *argArray* and invokes *fn* in the standard manner, passing *thisArg* as the value for `this` and the members of *argArray* as the individual argument values.

Returns

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

Implementation

```
static public function apply(fn: Callable, thisArg: Object=null, argArray: Object=null) {
    if (argArray === null)
        argArray = [];
    return apply(fn, thisArg, argArray);
}
```

NOTE The magic `apply` function performs the actual invocation (see magic::apply). This code will eventually change to use the prefix "spread" operator.

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

Description

1 The static `bind` method takes arguments *fn*, *thisArg*, and optionally some *args*.

Returns

2 The `bind` method returns a `Function` object that accepts some arguments *moreargs* and which calls *fn* with *thisArg* as the `this` object and the values of *args* and *moreargs* as actual arguments.

Implementation

```
static public function bind(method: Callable, thisObj: Object, ...args)
    bind(method, thisObj, args);

static helper function bind(method, thisObj, args) {
    return function (...moreargs)
               method.apply(thisObj, args.concat(moreargs));
}
```

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

Description

1 The static `call` method takes arguments *fn* and *thisArg* and optionally some *args* and invokes *fn* in the standard manner, passing *thisArg* as the value for `this` and the members of *args* as the individual argument values.

Returns

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

Implementation

```
static public function call(fn: Callable, thisObj: Object=null, ...args)
    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 (Ticket #173.) While it is necessary that the `invoke` method is completely magic in `Function` instances, it's not clear that 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 in the reference implementation). Therefore it should not be `final`.

Returns

4 The meta method `invoke` returns the value produces by the first `return` statement that is evaluated during the evaluation of 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* or *FunctionExpression*. Note in particular that the use and placement of white space, line terminators, and semicolons within the representation string is implementation-dependent.

COMPATIBILITY NOTE ES3 required the syntax to be that of a *FunctionDeclaration* only, but that made it impossible to produce a string representation for functions created from unnamed function expressions.

Returns

2 The intrinsic `toString` method returns a string.

Implementation

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

3 The private function `toString` is implementation-dependent.

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

Description

1 The intrinsic `apply` method calls the static `apply` method with the 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: Object=null, argArray: Object=null)
    Function.apply(this, thisArg, argArray);
```

5.3.4 intrinsic::bind (thisObj=…, ...args)

Description

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

Returns

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

Implementation

```
intrinsic function bind(thisObj: Object, ...args)
    Function.bind(this, thisObj, args);
```

5.3.5 intrinsic::call (thisObj=…, ...args)

Description

1 The intrinsic `call` method calls the static `apply` method with the 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: Object=null, ...args)
    Function.apply(this, thisObj, args);
```

5.4 Value properties of `Function` instances

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**:

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

5.6 Methods on the `Function` prototype object

1 The methods on the `Function` prototype object perform simple type adjustments and then perform the same actions as their intrinsic counterparts:

```
public prototype function toString(this: Function)
    this.private::toString();

public prototype function apply(this: Callable, thisArg=undefined, argArray=undefined)
    Function.apply(this,
                   thisArg === undefined ? null : thisArg,
                   argArray === undefined ? null : argArray);

public prototype function bind(this: Callable, thisObj, ...args)
    Function.bind(this, thisObj, args);

public prototype function call(this: Callable, thisObj=undefined, ...args)
    Function.apply(this,
                   thisObj === undefined ? null : thisObj,
                   args);
```

5.7 Value properties on the `Function` prototype object

5.7.1 length

1 The initial value of the `length` prototype property is 0.

Implementation

2 `public prototype var length : double = 0;`

COMPATIBILITY NOTE The "length" property of the prototype is not obviously required by the 3rd Edition of this Standard, but MSIE, Firefox, Opera, and Safari all provide it.

6 The class `Name`

1 The class `Name` is a final, nullable, non-dynamic, direct subclass of `Object` that reflects a property name as a pair of `Namespace` and `string` values.

COMPATIBILITY NOTE The `Namespace` class is new in the 4th Edition of this Standard.

6.1 Synopsis

1 The class `Name` provides the following interface:

```
__ES4__ final class Name extends Object
{
    public function Name(...args) …
    static meta function invoke(...args): Name …

    static public const length = 2

    override intrinsic function toString() : string …

    public const qualifier:  Namespace
    public const identifier: string
}
```

2 The `Name` prototype object provides the following direct properties:

```
    toString: function (this: Name) …
```

6.2 Operators

1 Two `Name` objects are equal (by `==` and `===`) if and only if their `qualifier` properties are equal and their `identifier` properties are equal (by the operator used to compare the `Name` objects).

6.3 Methods on the `Name` class object

6.3.1 new Name (x)

Returns

1 When the `Name` constructor is called with one argument *x* then *x* must be either a `Name` object, a `string`, a `String`, or an integer in the range 0 through 2^{32}-1. If *x* is a `Name` object then *x* is returned. Otherwise *x* is converted to a string and a `Name` object is returned whose `qualifier` is the public namespace and whose `identifier` is the converted value of *x*.

Implementation

```
public function Name( id ) …
```

6.3.2 new Name(x, y)

Returns

1 When the `Name` constructor is called with two arguments *x* and *y* it returns a `Name` object constructed from its arguments. The value of *x* must be a `Namespace` object. The value of *y* must be a `string`, a `String`, or an integer in the range 0 through 2^{32}-1. The `qualifier` of the returned value is *x*. The `identifier` of the returned value is the value of *y* converted to string.

Implementation

```
public function Name( ns: Namespace?, id ) …
```

6.3.3 Name (...args)

Description

1 The `Name` class object called as a function creates a `Name` object by invoking the `Name` constructor on its argument(s).

Returns

2 The `Name` class object called as a function returns a `Name` object.

Implementation

```
static meta function invoke( ...args ): Name!
    new Name(...args);
```

6.4 Methods on `Name` instances

6.4.1 intrinsic::toString ( )

Description

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

Returns

2 The intrinsic `toString` method returns a string.

Implementation

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

private function toString() : string {
    if (qualifier === public)
        return identifier;
    return string(qualifier) + "::" + identifier;
}
```

6.5 Value properties of `Name` instances

6.5.1 qualifier

1 The `qualifier` property holds the namespace value for this `Name` object. If `qualifier` is null then the implied namespace is the compatibility namespace **noNS**.

6.5.2 identifier

1 The `identifier` property holds the identifier value for this `Name` object. It is never null.

6.6 Methods on the `Name` prototype object

Description

1 The methods on the `Name` prototype object perform the same operations as their corresponding intrinsic methods perform.

Returns

2 The methods on the `Name` prototype object return what their corresponding intrinsic methods return.

Implementation

```
public prototype function toString(this : Name)
    this.private::toString();
```

7 The class `Namespace`

1 The class `Namespace` is a final, non-dynamic, nullable, direct subclass of `Object`.

NOTE `Namespace` values can be created by `new` expressions in the user program or by the evaluation of a `namespace` definition, which creates a new namespace and a constant binding for it.

COMPATIBILITY NOTE The `Namespace` class is new in the 4th Edition of this Standard.

7.1 Synopsis

1 The class `Namespace` provides the following interface:

```
__ES4__ final class Namespace extends Object
{
    public function Namespace(name=undefined) …
    static meta function invoke(x) …

    static public const length = 1

    override intrinsic function toString(): string …

    const name: (string|undefined) …
}
```

2 The `Namespace` prototype object provides the following direct properties:

```
    toString: function () …
```

7.2 Operators

1 The operators == and === compare forgeable `Namespace` objects by comparing their names as obtained by the `name` accessor, see below. Forgeable namespaces with the same name are equal by those operators.

2 In all other cases, `Namespace` objects are equal only to themselves.

7.3 Methods on the `Namespace` class object

7.3.1 new Namespace(name=…)

Returns

1 When the `Namespace` constructor is called with no arguments or with the argument **undefined** it returns a new unforgeable `Namespace` object. The returned object is unequal to every previously existing `Namespace` object.

2 When the `Namespace` constructor is called with an argument *name* that is not **undefined** it converts *name* to string and returns a new forgeable namespace whose name is the converted value.

Implementation

```
public function Namespace(name=undefined) …
```

7.3.2 Namespace(x)

Returns

1 The `Namespace` class object called as a function returns a `Namespace` object. If *x* is a `Namespace` object then it is returned. Otherwise a new `Namespace` object is constructed by invoking the `Namespace` constructor on *x*.

Implementation

```
static meta function invoke( x ): Namespace! {
    if (x is Namespace!)
        return x;
    return new Namespace(x);
}
```

7.4 Methods on `Namespace` instances

7.4.1 intrinsic::toString()

Description

1 The intrinsic `toString` method converts the `Namespace` object to a string. If the `Namespace` object is forgeable (it was created with an explicit name) then the string returned by `toString` contains the name as a substring.

Returns

2 The `toString` method returns an implementation-defined string.

3 Suppose the intrinsic `toString` method is invoked on two namespaces *N1* and *N2* yielding strings *T1* and *T2*, respectively. *T1* and *T2* are equal if and only if *N1* is equal to *N2* (by === or ==).

4 Suppose the intrinsic `toString` method is invoked on two different forgeable namespaces *N1* and *N2* created from strings *S1* and *S2*, yielding strings *T1* and *T2*, respectively. *T1* and *T2* have the same relationship (determined by the relational operators) as *S1* and *S2*.

7.5 Value Properties on `Namespace` instances

7.5.1 name

Description

1 If this `Namespace` object is a forgeable namespace then the value of the property `name` is the string name with which the namespace was constructed.

2 If this `Namespace` object is an unforgeable namespace then the value of the property `name` is **undefined**.

7.6 Methods on the `Namespace` prototype object

Description

1 The methods on the `Namespace` prototype object delegate to their corresponding intrinsic methods.

Returns

2 The methods on the `Namespace` prototype object return what their corresponding intrinsic methods return.

Implementation

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

8 The class `Array`

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* (in the form of a string value) is an array index if and only if *string(uint(P))* is equal to *P* and *uint(P)* is not equal to 2^{32}-1.

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.)

8.1 Synopsis

1 The Array class provides the following interface:

```
dynamic class Array extends Object
{
    function Array(...args) …
    static meta 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:AnyNumber=0): AnyNumber …
    static function join(object/*: Object!*/, separator: string=","): string …
    static function lastIndexOf(object/*:Object!*/, value, from:AnyNumber=NaN): AnyNumber
…
    static function map(object/*:Object!*/, mapper/*:function*/, thisObj:Object=null):
Array …
    static function pop(object/*:Object!*/) …
    static function push(object/*: Object!*/, ...args): double …
    static function reverse(object/*: Object!*/)/*: Object!*/ …
    static function shift(object/*: Object!*/) …
    static function slice(object/*: Object!*/, start: AnyNumber, end: AnyNumber, step:
AnyNumber) …
    static function some(object/*:Object!*/, checker/*:function*/, thisObj:Object=null):
boolean …
    static function sort(object/*: Object!*/, comparefn) …
    static function splice(object/*: Object!*/, start: AnyNumber, deleteCount:
AnyNumber, ...items): Array …
    static function unshift(object/*: Object!*/, ...items) : double …

    static const length = 1

    override intrinsic function toString():string …
    override intrinsic function toLocaleString():string …
    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:AnyNumber=0): AnyNumber …
    intrinsic function join(separator: string=","): string …
    intrinsic function lastIndexOf(value, from:AnyNumber=NaN): AnyNumber …
    intrinsic function map(mapper:Mapper, thisObj:Object=null): Array …
    intrinsic function pop() …
    intrinsic function push(...args): double …
    intrinsic function reverse()/*: Object!*/ …
    intrinsic function shift() …
    intrinsic function slice(start: AnyNumber, end: AnyNumber, step: AnyNumber): Array …
```

```
        intrinsic function some(checker:Checker, thisObj:Object=null): boolean …
        intrinsic function sort(comparefn:Comparator):Array …
        intrinsic function splice(start: AnyNumber, deleteCount: AnyNumber, ...items): Array …
        intrinsic function unshift(...items): double …

        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:         …
```

FIXME We've since also included `reduce` and `reduceRight` as static, intrinsic, and prototype methods.

8.2 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.

8.2.1 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 AnyNumber) {
            if (intrinsic::toUint(item) === item)
                this.length = intrinsic::toUint(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];
    }
}
```

8.2.2 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

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

8.2.3 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
    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`.

8.2.4 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;
}
```

8.2.5 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;
}
```

8.2.6 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);
}
```

8.2.7 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:AnyNumber=0): AnyNumber {
    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;
}
```

8.2.8 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=intrinsic::toUint(object.length) ; i < limit ; i++) {
        if (i > 0)
            out += separator;
        let x = object[i];
        if (x !== undefined && x !== null)
            out += string(x);
    }

    return out;
}
```

8.2.9 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:AnyNumber=NaN): AnyNumber {
    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;
}
```

8.2.10 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;
}
```

8.2.11 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 = intrinsic::toUint(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;
    }
}
```

8.2.12 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): double
    Array.push(object, args);
```

```
helper static function push(object/*:Object!*/, args: Array): double {
    let len = intrinsic::toUint(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`.

8.2.13 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 = intrinsic::toUint(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;
}
```

NOTE Property deletion is observable to objects that implement the `meta::delete` method, and may not be omitted from this algorithm.

8.2.14 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 = intrinsic::toUint(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;
}
```

8.2.15 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: AnyNumber, end: AnyNumber, step:
AnyNumber) {
    let len = intrinsic::toUint(object.length);

    step = int(step);
    if (step == 0)
        step = 1;

    if (intrinsic::isNaN(start))
        start = step > 0 ? 0 : (len-1);
    else
        start = clamp(start, len);

    if (intrinsic::isNaN(end))
        end = step > 0 ? len : (-1);
    else
        end = clamp(end, len);

    let out = new Array;
    for (let i = start; step > 0 ? i < end : i > end; i += step)
        out.push(object[i]);

    return out;
}

helper function clamp(val: AnyNumber, len: double): double {
    val = toInteger(val);
    if (val < 0)
        val += len;
    return intrinsic::toUint( Math.min( Math.max( val, 0 ), len ) );
}
```

8.2.16 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;
}
```

8.2.17 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 sorting-logic), 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 ≤ i < len*
2. *0 ≤ j < len*
3. *object* does not have a property with name `string(i)`
4. *P* is obtained by following one or more `[[Prototype]]` properties starting at `this`
5. *P* has a property with name `string(j)`

4 If the behavior of `sort` is not implementation-defined then the array is sorted as described in section sorting-logic.

Returns

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

Implementation

6 The static `sort` method calls on the generic sorting engine, passing a function to compare elements of *object*.

```
static function sort(object/*: Object!*/, comparefn) {

    function compare(j, k) {
        if (!(j in object) && !(k in object))
            return 0;
        if (!(j in object))
            return 1;
        if (!(k in object))
            return -1;

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

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

        if (comparefn !== undefined)
            return comparefn(x, y);

        x = String(x);
        y = String(y);
        if (x < y) return -1;
        if (x > y) return 1;
        return 0;
    }

    let len = intrinsic::toUint(object.length);
    sortEngine(object, 0, len-1, compare);
    return object;
}
```

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.

8.2.17.1 The sorting engine

1 The sorting engine sorts the numerically named properties of an object between two indices `low` and `high` inclusive, using a sort-specific function `sortCompare` to compare elements at two indices:

```
informative function sortEngine(object, low, high, sortCompare) …
```

2 The sorting engine perform an implementation-dependent sequence of calls to the `[[Get]]`, `[[Put]]`, and `[[Delete]]` methods of *object* and to *sortCompare*, where the first argument for each call to `[[Get]]`, `[[Put]]`, or `[[Delete]]`, and both arguments to *sortCompare*, are nonnegative integers greater than or equal to *low* and less than or equal to *high*.

3 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 in the range *low* to *high* inclusive, such that for every nonnegative integer *j* in that range, 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* in that range, if *sortCompare(j,k) < 0*, then *π(j) < π(k).*

4 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.

5 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.

8.2.18 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: AnyNumber, deleteCount:
AnyNumber, ...items): Array
    Array.splice(object, start, deleteCount, items);

helper static function splice(object/*: Object!*/, start: AnyNumber, deleteCount:
AnyNumber, items: Array) {
    let out = new Array();
    let len = intrinsic::toUint(object.length);

    start = clamp( start, len );
    deleteCount = clamp( deleteCount, len - start );

    let end = start + deleteCount;

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

    let insertCount = items.length;
    let shiftAmount = insertCount - deleteCount;

    if (shiftAmount < 0) {
        shiftAmount = -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 {
        for (let i = len; i > end; ) {
            --i;
            object[i + shiftAmount] = object[i];
        }
    }

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

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

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

8.2.19 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) : double
    Array.unshift(this, object, items);

helper static function unshift(object/*: Object!*/, items: Array) : double {
    let len = intrinsic::toUint(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;
}
```

8.3 Method Properties of Array Instances

8.3.1 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 to 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.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:AnyNumber=0): AnyNumber
    Array.indexOf(this, value, from);

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

intrinsic function lastIndexOf(value, from:AnyNumber=NaN): AnyNumber
    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): double
    Array.push(this, args);

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

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

intrinsic function slice(start: AnyNumber, end: AnyNumber, step: AnyNumber): Array
    Array.slice(this, start, end, step);

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: AnyNumber, deleteCount: AnyNumber, ...items): Array
    Array.splice(this, start, deleteCount, items);

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

8.4 Value properties of Array instances

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

8.4.1 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.

8.5 Method properties on the `Array` prototype object

8.5.1 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();
```

8.5.2 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 method 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.

8.5.3 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.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, Number(from));

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

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

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

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

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

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

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

prototype function slice(start, end, step)
    Array.slice(this, Number(start), Number(end), Number(step))

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.splice(this, Number(start), Number(deleteCount), items);

prototype function unshift(...items)
    Array.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.

9 String classes

1 ECMAScript provides a primitive string representation in the class `string`. It is primitive in the sense that this representation is directly operated upon by operators of the language, and in the sense that it is a final and non-dynamic class for which ECMAScript implementations may provide efficient representations.

2 ECMAScript also provides the class `String`, which is a dynamic non-final class that holds `string` values. Instances of `String` are converted to `string` when operated upon by operators of the language.

9.1 The type `AnyString`

1 The type `AnyString` is a union type that contains the two built-in string types. By standard subtyping rules it also includes all classes that extend `String`.

```
__ES4__ type AnyString = (string|String!);
```

10 The class `String`

1 The class `String` is a dynamic, nullable, non-final subclass of `Object`. It is a container for `string` values. Instances of `String` are converted to `string` when operated upon by the operators of the language.

2 The class `String` can be extended and the extending classes can provide novel representations for string values.

10.0.1 Synopsis

1 The class `String` provides the following interface:

FIXME Optional arguments need to be handled better in these interfaces.

```
dynamic class String
{
    function String(value="") …
    static meta function invoke(value="") …

    static function fromCharCode(...args) …
    static function charAt(self, pos) …
    static function charCodeAt(self, pos) …
    static function concat(self, ...args) : string …
    static function indexOf(self, searchString, position): double …
    static function lastIndexOf(self, searchString, position) : double …
    static function localeCompare(self, that) : double …
    static function match(self, regexp) : Array …
    static function replace(self, searchValue, replaceValue) : string …
    static function search(self, regexp) : double …
    static function slice(self, start, end, step): string …
    static function split(self, separator, limit): Array! …
    static function substring(self, start, end): string …
    static function toLowerCase(self): string …
    static function toLocaleLowerCase(self): string …
    static function toUpperCase(self): string …
    static function toLocaleUpperCase(self): string …
    static function trim(self) : string …

    static const length: uint = 1

    override intrinsic function toString() : string …
    override intrinsic function valueOf() : string …

    intrinsic function charAt(pos: double = 0) : string …
    intrinsic function charCodeAt(pos: double = 0) : double …
    intrinsic function concat(...args) : string …
    intrinsic function indexOf(searchString: AnyString, position: double = 0.0) : double …
    intrinsic function lastIndexOf(searchString: AnyString, position: double) : double …
    intrinsic function localeCompare(that: AnyString) : double …
    intrinsic function match(regexp: RegExp!) : Array …
    intrinsic function replace(s: (RegExp!|AnyString), r: (AnyString|function(...)
:AnyString)) : string …
    intrinsic function search(regexp: RegExp!) : double …
    intrinsic function slice(start: AnyNumber, end: AnyNumber, step: AnyNumber): string …
    intrinsic function split(separator:(AnyString|RegExp!), limit: double = double.MAX_
VALUE) : Array! …
    intrinsic function substring(start: double, end: double=Infinity) : string …
    intrinsic function toLowerCase(): string …
    intrinsic function toLocaleLowerCase(): string …
    intrinsic function toUpperCase() : string …
    intrinsic function toLocaleUpperCase() : string …
    intrinsic function trim() : string …

    function get length() : double …
    meta function get(n) …
}
```

2 The `String` prototype object provides the following direct properties:

FIXME Optional arguments need to be handled better in these interfaces.

```
    toString:           function (this:Strings) …
    valueOf:            function (this:Strings) …
    charAt:             function (pos) …
    charCodeAt:         function (pos) …
    concat:             function (...strings) …
    indexOf:            function (searchString, pos) …
    lastIndexOf:        function (searchString, pos) …
    localeCompare:      function (that) …
    match:              function (regexp) …
    replace:            function (searchValue, replaceValue) …
    search:             function (regexp) …
    slice:              function (start, end) …
    split:              function (separator, limit) …
    substring:          function (start, end) …
    toLowerCase:        function () …
    toLocaleLowerCase:  function () …
    toUpperCase:        function () …
    toLocaleUpperCase:  function () …
    trim:               function () …
```

3 The `String` prototype object is also used as the prototype object for the class `string`.

10.1 Methods on the `String` class object

10.1.1 new String (value=…)

Description

1 The `String` constructor initializes a new `String` object by storing *value*, converted to `string`, in a private property. *Value* defaults to the empty string.

10.1.2 String(value=…)

Description

1 The `String` class object called as a function converts *value* to `string` (not to `String`). *Value* defaults to the empty string.

Returns

2 The `String` class object called as a function returns a `string` object.

Implementation

```
static meta function invoke(value="")
    string(value);
```

10.1.3 Methods that delegate to `string` methods

Description

1 The intrinsic methods `charAt`, `charCodeAt`, `concat`, `indexOf`, `lastIndexOf`, `localeCompare`, `match`, `replace`, `search`, `slice`, `split`, `substring`, `toLowerCase`, `toLocaleLowerCase`, `toUpperCase`, `toLocaleUpperCase`, and `trim` all delegate to the corresponding static methods on the `string` class.

Returns

2 These static methods return what their corresponding static methods on the `string` class return.

Implementation

```
static function charAt(self, pos)
    string.charAt(self, pos);

static function charCodeAt(self, pos)
    string.charCodeAt(self, pos);

static function concat(self, ...args) : string
    string.concat(self, args);

static function indexOf(self, searchString, position): double
    string.indexOf(self, searchString, position);

static function lastIndexOf(self, searchString, position) : double
    string.lastIndexOf(self, searchString, position);

static function localeCompare(self, that) : double
    string.localeCompare(self, that);

static function match(self, regexp) : Array
    string.match(self, regexp);

static function replace(self, searchValue, replaceValue) : string
    string.replace(self, searchValue, replaceValue);

static function search(self, regexp) : double
    string.search(self, regexp);

static function slice(self, start, end, step): string
    string.slice(self, Number(start), Number(end), Number(step));

static function split(self, separator, limit): Array!
    string.split(self, separator, limit);

static function substring(self, start, end): string
    string.substring(self, start, end);

static function toLowerCase(self): string
    string.toLowerCase(self);

static function toLocaleLowerCase(self): string
    string.toLocaleLowerCase(self);

static function trim(self) : string
    string.trim(self);
```

10.2 Methods on `String` instances

10.2.1 toString

Returns

1 The intrinsic `toString` method returns this `String` object converted to `string`. For the class `String` itself this results in the extraction of the private string value held by the `String`. Subclasses of `String` can represent strings differently by overriding `toString`.

Implementation

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

10.2.2 valueOf

Returns

1 The intrinsic `valueOf` method returns the result of calling the intrinsic `toString` method.

Implementation

```
override intrinsic function valueOf() : string
    val;
```

10.2.3 Methods that delegate to `string` methods

Description

1 The intrinsic methods `charAt`, `charCodeAt`, `concat`, `indexOf`, `lastIndexOf`, `localeCompare`, `match`, `replace`, `search`, `slice`, `split`, `substring`, `toLowerCase`, `toLocaleLowerCase`, `toUpperCase`, `toLocaleUpperCase`, and `trim` all delegate to the corresponding static methods on the `string` class, passing `this` as the first argument in all cases.

Returns

2 These intrinsic methods return what their corresponding static methods on the `string` class return.

Implementation

```
intrinsic function charAt(pos: double = 0) : string
    string.charAt(val, pos);

intrinsic function charCodeAt(pos: double = 0) : double
    string.charCodeAt(val, pos);

intrinsic function concat(...args) : string
    string.concat(val, args);

intrinsic function indexOf(searchString: AnyString, position: double = 0.0) : double
    string.indexOf(val, searchString, position);

intrinsic function lastIndexOf(searchString: AnyString, position: double) : double
    string.lastIndexOf(val, searchString, position);

intrinsic function localeCompare(that: AnyString) : double
    string.localeCompare(val, that);

intrinsic function match(regexp: RegExp!) : Array
    string.match(val, regexp);

intrinsic function replace(s: (RegExp!|AnyString), r: (AnyString|function(...):AnyString))
  : string
    string.replace(val, searchValue, replaceValue);

intrinsic function search(regexp: RegExp!) : double
    string.search(val, r);

intrinsic function slice(start: AnyNumber, end: AnyNumber, step: AnyNumber): string
    string.slice(val, start, end, step);

intrinsic function split(separator:(AnyString|RegExp!), limit: double = double.MAX_VALUE)
  : Array!
    string.split(val, separator, limit);

intrinsic function substring(start: double, end: double=Infinity) : string
    string.substring(val, start, end);

intrinsic function toLowerCase(): string
    string.toLowerCase(val);

intrinsic function toLocaleLowerCase(): string
    string.toLocaleLowerCase(val);

intrinsic function toUpperCase() : string
    string.toUpperCase(val);

intrinsic function toLocaleUpperCase() : string
    string.toLocaleUpperCase(val);

intrinsic function trim() : string
    string.trim(string(val));
```

NOTE The second parameter to the intrinsic method `localeCompare` and the first parameter to the intrinsic methods `toLocaleLowerCase` and `toLocaleUpperCase` are likely to be used in a future version of this standard; it is recommended that implementations do not use these parameter position for anything else.

10.3 Methods on the `String` prototype object

10.3.1 toString ( )

Returns

1 Returns the result of invoking the intrinsic `toString` method.

2 The `toString` function is not generic; it throws a **TypeError** exception if its this value is not a `String` or `string` object. Therefore, it cannot be transferred to other kinds of objects for use as a method.

Implementation

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

10.3.2 valueOf ( )

Returns

1 Returns this result of invoking the intrinsic `valueOf` method.

2 The `valueOf` function is not generic; it throws a **TypeError** exception if its this value is not a `String` or `string` object. Therefore, it cannot be transferred to other kinds of objects for use as a method.

Implementation

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

10.3.3 Methods that delegate to `string` methods

Description

1 The methods `charAt`, `charCodeAt`, `concat`, `indexOf`, `lastIndexOf`, `localeCompare`, `match`, `replace`, `search`, `slice`, `split`, `substring`, `toLowerCase`, `toLocaleLowerCase`, `toUpperCase`, `toLocaleUpperCase`, and `trim` on the `String` prototype object all delegate to the corresponding static methods on the `string` class, passing `this` as the first argument in all cases.

2 These methods are all generic, they do not require that their `this` object is a `String`. Therefore, they can be transferred to other kinds of objects for use as methods.

Returns

3 These methods on the `String` prototype object all return the values returned by their corresponding static methods on the `string` class.

Implementation

```
prototype function charAt(pos)
    string.charAt(this.toString(), pos);

prototype function charCodeAt(pos)
    string.charCodeAt(this.toString(), pos);

prototype function concat(...args)
    string.concat(this.toString(), args);

prototype function indexOf(searchString, position)
    string.indexOf(this.toString(), searchString, position);

prototype function lastIndexOf(searchString, position)
    string.lastIndexOf(this.toString(), searchString, position);

prototype function localeCompare(that)
    string.localeCompare(this.toString(), that);

prototype function match(regexp)
    string.match(this.toString(), regexp);

prototype function replace(searchValue, replaceValue)
    string.replace(this.toString(), searchValue, replaceValue);

prototype function search(regexp)
    string.search(this.toString(), regexp);

prototype function slice(start, end, step)
    string.slice(this.toString(), Number(start), Number(end), Number(step));
```

```
prototype function split(separator, limit)
    string.split(this.toString(), separator, limit);

prototype function substring(start, end)
    string.substring(this.toString(), start, end);

prototype function toLowerCase()
    string.toLowerCase(this.toString());

prototype function toLocaleLowerCase()
    string.toLocaleLowerCase(this.toString());

prototype function trim()
    string.trim(this.toString());
```

NOTE The second parameter to the prototype method `localeCompare` and the first parameter to the prototype methods `toLocaleLowerCase` and `toLocaleUpperCase` are likely to be used in a future version of this standard; it is recommended that implementations do not use these parameter position for anything else.

11 The class `string`

1 The class `string` is a final, non-nullable, non-dynamic subclass of Object that represents an immutable indexable sequence of Unicode characters. The property `"length"` holds the number of characters in this sequence. The property `"0"` names the first character, the property `"1"` names the second character, and so on, up to property `"length"`-1. Single characters are represented as `string` objects with length equal to one.

2 The `string` class has the same prototype object as the `String` class; changes made to the prototype object of one class are visible on the prototype object of the other class.

COMPATIBILITY NOTE The class `string` is new in the 4th Edition of this Standard, but `string` models the "string values" in the 3rd Edition.

11.1 Synopsis

1 The class `string` provides the following interface:

```
final class string!
{
    function string(value="") …
    static meta function invoke(value="") …

    static function fromCharCode(...codes) …
    static function charAt(self, pos) : string …
    static function charCodeAt(self, pos) : double …
    static function concat(self, ...args) …
    static function indexOf(self, searchString, position) : double …
    static function lastIndexOf(self, searchString, position) : double …
    static function localeCompare(self, that) : double …
    static function match(self, regexp): Array …
    static function replace(self, s, r): string …
    static function search(self, regexp): double …
    static function slice(object, start: AnyNumber=NaN, end: AnyNumber=NaN, step:
AnyNumber=1) …
    static function split(self, separator, limit) : Array! …
    static function substring(self, start, end) : string …
    static function toLowerCase(self): string …
    static function toLocaleLowerCase(self): string …
    static function toUpperCase(self): string …
    static function toLocaleUpperCase(self) …
    static function trim(s): string …

    static const length: uint = 1

    override intrinsic function toString() : string …
    override intrinsic function valueOf() : string …

    intrinsic function charAt(pos: double = 0) : string …
    intrinsic function charCodeAt(pos: double = 0) : double …
    intrinsic function concat(...args) : string …
    intrinsic function indexOf(searchString: string, position: double = 0.0) : double …
    intrinsic function lastIndexOf(searchString: string, position: double) : double …
    intrinsic function localeCompare(that : string) : double …
    intrinsic function match(regexp: RegExp) : Array …
    intrinsic function replace(searchValue: (string|RegExp!), …
    intrinsic function search(regexp: RegExp!) : double …
    intrinsic function slice(start: AnyNumber=NaN, end: AnyNumber=NaN, step: AnyNumber=1):
```

```
 string …
    intrinsic function split(separator:(string|RegExp!), limit: double = double.MAX_VALUE)
: Array! …
    intrinsic function substring(start: double, end: double=this.length) : string …
    intrinsic function toLowerCase() : string …
    intrinsic function toLocaleLowerCase() : string …
    intrinsic function toUpperCase() : string …
    intrinsic function toLocaleUpperCase() : string …
    intrinsic function trim() : string …

    function get length() : double …
    meta function get(pos) …
}
```

11.2 Static Methods on the `string` Class

11.2.1 new string (value=…)

Description

1 The `string` constructor initializes a new `string` object by storing an implementation-dependent string representation of *value* in a private property. The default *value* is the empty string.

Implementation

2 The `string` constructor is implementation-dependent.

11.2.2 string (value=…)

Description

1 The `string` class object called as a function converts *value* to `string` as by the `ToString` operator. The default *value* is the empty string.

Returns

2 The `string` class object called as a function returns a `string`.

Implementation

```
static meta function invoke(value="")
    (value is string) ? value : new string(value);
```

FIXME (Ticket #176.) The use of `magic::newString` is an optimization that confuses the spec; `new string(x)` would have been better.

11.2.3 fromCharCode (…codes)

Description

1 The static `fromCharCode` method creates a `string` containing as many characters as there are elements in *codes*. Each element of *codes* specifies the Unicode code point value of one character of the resulting string, with the first argument specifying the first character, and so on, from left to right.

FIXME (Ticket #170.) The code below assumes a 21-bit Unicode representation. What happens in a system that only has 16-bit unicode? We'd like to be backwards compatible. If so, the upper bits are ignored. This conflicts with how `\u{...}` is handled, though: it creates two code points.

Returns

2 The static `fromCharCode` method returns the computed `string`.

Implementation

```
static function fromCharCode(...codes)
    string.fromCharCode(codes);

helper static function fromCharCode(codes: Array): string {
    let s = "";
    for (let i=0, limit=codes.length ; i < limit ; ++i)
        s += fromCharCode(intrinsic::toUint(codes[i] & 0x1FFFFF));
    return s;
}
```

11.2.4 charAt (self, pos)

Description

1 The static `charAt` method converts *self* to `string` and extracts the character at index *pos*.

Returns

2 The static `charAt` method returns a `string`.

Implementation

```
static function charAt(self, pos) : string {
    let S    = string(self);
    let ipos = toInteger(pos);
    if (ipos < 0 || ipos >= S.length)
        return "";
    return fromCharCode(charCodeAt(S, intrinsic::toUint(ipos)));
}
```

FIXME (Ticket #176.) The use of `magic::charCodeAt` is an optimization that complicates the spec; `string.charCodeAt(x)` would have been better.

11.2.5 charCodeAt (self, pos)

Description

1 The static `charCodeAt` method converts *self* to `string` and extracts the code point value of the character at index *pos*.

Returns

2 The static `charCodeAt` method returns a number.

Implementation

```
static function charCodeAt(self, pos) : double {
    let S = string(self);
    let ipos = toInteger(pos);
    if (ipos < 0 || ipos >= S.length)
        return NaN;
    return charCodeAt(S, intrinsic::toUint(ipos));
}
```

11.2.6 concat (self, ...strings)

Description

1 The static `concat` method computes a `string` value consisting of the characters of *self* (converted to `string`) followed by the characters of each of the elements of *strings* (where each argument is converted to `string`).

Returns

2 The static `concat` method returns the concatenated `string`.

Implementation

```
static function concat(self, ...args)
    string.concat(self, args);

helper static function concat(self, strings) : string {
    let S = string(self);
    let n = strings.length;
    for (let i=0; i < n ; i++)
        S += string(strings[i]);
    return S;
}
```

11.2.7 indexOf (self, searchString, position)

Description

1 The static `indexOf` method searches *self* (converted to string) for occurrences of *searchString* (converted to `string`), at positions that are greater than or equal to *position* (converted to integer).

Returns

2 The static `indexOf` method returns the smallest index at which a match was found, or -1 if there was no match.

Implementation

```
static function indexOf(self, searchString, position) : double {
    let S     = string(self);
    let SS    = string(searchString);
    let pos   = toInteger(position);
    let slen  = S.length;
    let m     = Math.min(Math.max(pos, 0), slen);
    let sslen = SS.length;
    let lim   = slen - sslen + 1;
```

FIXME (Ticket #176.) The use of `magic::charCodeAt` is an optimization that complicates the spec; using `string.charCodeAt` would have been better.

11.2.8 lastIndexOf (self, searchString, position)

Description

1 The static `lastIndexOf` method searches *self* (converted to string) for occurrences of *searchString* (converted to `string`), at positions that are smaller than or equal to *position* (converted to integer).

Returns

2 The static `lastIndexOf` method returns the greatest index at which a match was found, or -1 if there was no match.

Implementation

```
static function indexOf(self, searchString, position) : double {
    let S     = string(self);
    let SS    = string(searchString);
    let pos   = toInteger(position);
    let slen  = S.length;
    let m     = Math.min(Math.max(pos, 0), slen);
    let sslen = SS.length;
    let lim   = slen - sslen + 1;
```

FIXME (Ticket #176.) The use of `magic::charCodeAt` is an optimization that complicates the spec; using `string.charCodeAt` would have been better.

11.2.9 localeCompare (self, other)

Description

1 The static `localeCompare` method compares *self* (converted to `string`) with *other* (converted to string) in a locale-sensitive manner. The two strings are compared in an implementation-defined fashion. The comparison is intended to order strings in the sort order specified by the system default locale.

Returns

2 The static `localeCompare` method returns a number other than **NaN** that represents the result of the comparison. The result will be negative, zero, or positive, depending on whether *self* comes before *other* in the sort order, the strings are equal, or *self* comes after *other* in the sort order, respectively.

3 The static `localeCompare` method is a consistent comparison function (as defined in sort:consistent_comparator) on the set of all strings. Furthermore, `localeCompare` returns 0 or -0 when comparing two strings that are considered canonically equivalent by the Unicode standard.

4 The actual return values are left implementation-defined to permit implementers to encode additional information in the result value, but the function is required to define a total ordering on all strings and to return 0 when comparing two strings that are considered canonically equivalent by the Unicode standard.

Implementation

5 The static `localeCompare` method is implementation-defined.

NOTE This function is intended to rely on whatever language-sensitive comparison functionality is available to the ECMAScript environment from the host environment, and to compare according to the rules of the host environment's current locale. It is strongly recommended that this function treat strings that are canonically equivalent according to the Unicode standard as identical (in other words, compare the strings as if they had both been converted to Normalised Form C or D first). It is also recommended that this function not honour Unicode compatibility equivalences or decompositions. If no language-sensitive comparison at all is available from the host environment, this function may perform a bitwise comparison.

NOTE The third 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.

11.2.10 match (self, regexp)

Description

1 The static `match` method searches *self* (converted to `string`) for occurrences of *regexp* (converted to `RegExp`).

Returns

2 If the `global` flag on *regexp* is **false**, the `match` method returns the result obtained by invoking the intrinsic `exec` method on *regexp* with *self* as a parameter.

3 If the `global` flag on *regexp* is **true**, the `match` method returns an array of strings containing the substrings of *self* that were matched by *regexp*, in order.

Implementation

```
static function match(self, regexp): Array {
    let S = string(self);
    let R = (regexp is RegExp) ? regexp : new RegExp(regexp);

    if (!R.global)
        return R.exec(S);

    let matches = [];

    R.lastIndex = 0;
    while (true) {
        let oldLastIndex = R.lastIndex;
        let res = R.exec(S);

        if (res === null)
            break;

        matches.push(res[0]);
        if (R.lastIndex === oldLastIndex)
            ++R.lastIndex;
    }
    if (matches.length == 0)
        return null;
    else
        return matches;
}
```

11.2.11 replace (self, searchValue, replaceValue)

Description

1 The static `replace` method computes a string from *self* (converted to `string`) by replacing substrings matching *searchValue* (converted to `string` if not `RegExp`) by instances of *replaceValue* (converted to `string` if not a function).

2 If *replaceValue* is a function, then it is called once for each matched substring on arguments providing details about the match, and the value returned from this call is converted to `string` if necessary and replaces the matched substring.

3 If *replaceValue* is not a function then a string to replace a matched substring is derived from *replaceValue* by replacing characters of *replaceValue* (converted to `string`) as specified in the following table. These `$` replacements are done left-to-right, and, once such a replacement is performed, the new replacement text is not subject to further replacements. For example, `"$1,$2".replace(/(\$(\d))/g, "$$1-$1$2")` returns `"$1-$11,$1-$22"`. A `$` in *replaceValue* that does not match any of the forms below is left as is.

Characters	Replacement text
$$	$
$&	The matched substring.
$`	The portion of *self* that precedes the matched substring.
$'	The portion of *self* that follows the matched substring.
$n	The *n*th capture, where *n* is a single digit 1-9 and `$n` is not followed by a decimal digit. If $n \leq m$ and the *n*th capture is undefined, use the empty string instead. If $n > m$, the result is implementation-defined.
$nn	The *nn*th capture, where *nn* is a two-digit decimal number 01-99. If $nn \leq m$ and the *nn*th capture is undefined, use the empty string instead. If $nn > m$, the result is implementation-defined.

NOTE In the above table, *m* is the length of the search result's capture array.

Returns

4 The static `replace` function returns a `string` object that is the concatenation of the unmatched portions of *self* and the computed replace values for the matched portions of *self*, in order.

Implementation

```
static function replace(self, s, r): string {

    let function substituteFunction(start: double, end: double, m: double, cap: Array) :
string {
        let A = [];
        A[0] = S.substring(start, end);
        for ( let i=0 ; i < m ; i++ )
            A[i+1] = cap[i+1];
        A[m+2] = start;
        A[m+3] = S;
        return string(replaceFun.apply(null, A));
    }

    let function substituteString(start: double, end: double, m: double, cap: Array) :
string {
        let s   = "";
        let i   = 0;
        let r   = /\$(?:(\$)|(\&)|(\`)|(\')|([0-9]{1,2}))/g;
        let res;

        while ((res = r.exec(replaceString)) !== null) {
            s += replaceString.substring(i, r.lastIndex - res[0].length);
            i = r.lastIndex;

            if (res[1])      s += "$";
            else if (res[2]) s += S.substring(start, end);
            else if (res[3]) s += S.substring(0, start);
            else if (res[4]) s += S.substring(end);
            else {
                let n = parseInt(res[5]);
                if (n <= m && cap[n] !== undefined)
                    s += cap[n];
            }
        }
        s += replaceString.substring(i);

        return s;
    }
```

```
        let function match( regexp, i : double ) : [double, CapArray]  {
            while (i <= S.length) {
                let res : MatchResult = regexp.match(S, i);
                if (res !== null) {
                    res.captures[0] = S.substring(i,res.endIndex);
                    return [i, res.captures];
                }
                ++i;
            }
            return [0, null];
        }

        let S             = string(self);
        let replaceString = (r is string) ? r cast string : null;
        let replaceFun    = (r is Function) ? r cast Function : null;

        let substitute : function (double, double, double, Array) : string =
            replaceFun !== null ? substituteFunction : substituteString;

        if (s !== null && s is RegExp) {
            let regexp = s cast RegExp;
            let m      = regexp.nCapturingParens;

            if (!regexp.global) {
                let [i, res] = match(regexp, 0);

                if (res === null)
                    return S;

                let end = i + res[0].length;
                return S.substring(0,i) + substitute(i, end, m, res) + S.substring(end);
            }
            else {
                let newstring = "";
                let prevEnd   = 0;

                regexp.lastIndex = 0;
                while (true) {
                    let oldLastIndex : double = regexp.lastIndex;
                    let [i,res] = match(regexp, intrinsic::toUint(oldLastIndex));

                    if (res === null)
                        break;

                    newstring += S.substring(prevEnd, i);

                    let end = i + res[0].length;
                    regexp.lastIndex = end;
                    if (regexp.lastIndex == oldLastIndex)
                        regexp.lastIndex++;
                    newstring += substitute(i, end, m, res);
                    prevEnd = end;
                }
                newstring += S.substring(prevEnd, S.length);

                return newstring;
            }
        }
        else {
            let searchString = string(s);
            let pos          = S.indexOf(searchString, 0);

            if (pos === -1)
                return S;

            let end = pos + searchString.length;
            return S.substring(0,pos) + substitute(pos, end, 0, []) + S.substring(end);
        }
    }
```

FIXME (Ticket #177.) The code above needs to be factored into a top-level function with the auxiliary functions following it; values of names now free in the nested functions must be passed as parameters.

11.2.12 search (self, regexp)

Description

1 The static `search` method searches *self* (converted to `string`) for the first occurrence of the search term *regexp* (converted to `RegExp`).

NOTE This method ignores the `lastIndex` and global properties of *regexp*. The `lastIndex` property of *regexp* is left unchanged.

Returns

2 The static `search` method returns a number indicating the index at which a match was made, or -1 if there was no match.

Implementation

```
static function search(self, regexp): double {
    let S = string(self);
    let R = (regexp is RegExp) ? regexp : new RegExp(regexp);

    for ( let i=0, limit=S.length ; i < limit ; i++ )
        if (R.match(S, i) !== null)
            return i;
    return -1;
}
```

11.2.13 slice (self, start, end)

Description

1 The static `slice` method extracts a substring of *self* (converted to `string`) from *start* and up to but not including *end* (both converted to integer). Both *start* and *end* may be negative.

Returns

2 The static `slice` method returns a `string`.

Implementation

```
static function slice(object, start: AnyNumber=NaN, end: AnyNumber=NaN, step:
AnyNumber=1) {

    let len = intrinsic::toUint(object.length);

    step = int(step);
    if (step == 0)
        step = 1;

    if (intrinsic::isNaN(start))
        start = step > 0 ? 0 : (len-1);
    else
        start = clamp(start, len);

    if (intrinsic::isNaN(end))
        end = step > 0 ? len : (-1);
    else
        end = clamp(end, len);

    let out = new string();
    for (let i = start; step > 0 ? i < end : i > end; i += step)
        out += object[i];

    return out;
}
```

11.2.14 split (self, separator, limit)

Description

1 The static `split` method extracts substrings from *self* (converted to `string`), where substrings are separated by instances of *separator* (converted to `string` if not a `RegExp`). At most *limit* substrings are extracted.

2 Occurrences of *separator* are not part of any substring in the result.

3 The value of *separator* may be an empty string, an empty regular expression, or a regular expression that can match an empty string. In this case, *separator* does not match the empty substring at the beginning or end of the input string, nor does it match the empty substring at the end of the previous separator match. (For example, if *separator* is the empty string, the string is split up into individual characters; the length of the result array equals the length of the string, and each substring contains one character.) If *separator* is a regular expression, only the first match at a given position of the `this` string is considered, even if backtracking could yield a non-empty-substring match at that position. (For example, `string.split("ab",/a*?/)` evaluates to the array `["a","b"]`, while `string.split("ab",/a*/)` evaluates to the array `["","b"]`.)

4 If *self* is (or converts to) the empty string, the result depends on whether *separator* can match the empty string. If it can, the result contains no elements. Otherwise, the result contains one element, which is the empty string.

5 If *separator* is a regular expression that contains capturing parentheses, then each time *separator* is matched the results (including any undefined results) of the capturing parentheses are spliced into the result. For example,

```
"A<B>bold</B>and<CODE>coded</CODE>".split(/<(\/)?([^<>]+)>/)
```

evaluates to the array

```
["A", undefined, "B", "bold", "/", "B", "and", undefined, "CODE", "coded", "/", "CODE",
""].
```

6 If *separator* is **undefined**, then the result contains just one string, which is *self* (converted to `string`).

Returns

7 The static `split` method returns a new `Array` object holding the extracted substrings, in order.

Implementation

```
static function split(self, separator, limit) : Array! {
```

FIXME (Ticket #178.) The exposition leaves something to be desired. Should split `splitMatch` out as a separate helper function, at least.

NOTE The static `split` method ignores the value of *separator*`.global` for separators that are `RegExp` objects.

11.2.15 substring (self, start, end)

Description

1 The static `substring` method extracts a substring from *self* (converted to `string`) from *start* up to but not including *end* (converted to number).

Returns

2 The static `substring` method returns a `string`.

Implementation

```
static function substring(self, start, end) : string {
    let S   = string(self);
    let len = S.length;

    start = toInteger(start);
    end = end === undefined ? len : toInteger(end);

    start = Math.min(Math.max(start, 0), len);
    end = Math.min(Math.max(end, 0), len);

    if (start > end)
        [start, end] = [end, start];

    let s = "";
    for ( let i=start ; i < end ; i++ )
        s += S[i];

    return s;
}
```

11.2.16 toLowerCase (self)

Description

1 The static `toLowerCase` method converts the characters of *self* (converted to string) to lower case. The characters are converted one by one. The result of each conversion is the original character, unless that character has a Unicode lowercase equivalent, in which case the lowercase equivalent is used instead.

NOTE The result should be derived according to the case mappings in the Unicode character database (this explicitly includes not only the `UnicodeData.txt` file, but also the `SpecialCasings.txt` file that accompanies it in Unicode 2.1.8 and later).

Returns

2 The static `toLowerCase` method returns a `string`.

Implementation

```
static function toLowerCase(self): string {
    let S = string(self);
    let s = "";
    for ( let i=0, limit=S.length ; i < limit ; i++ ) {
        let u = Unicode::toLowerCaseCharCode(charCodeAt(S,intrinsic::toUint(i)));
        if (u is double)
            s += fromCharCode(intrinsic::toUint(u));
        else {
            for ( let j=0 ; j < u.length ; j++ )
                s += fromCharCode(u[j]);
        }
    }
    return s;
}
```

FIXME (Ticket #176.) The use of `magic::charCodeAt` and `magic::fromCharCode` is a confusing optimization.

FIXME (Ticket #179.) Cross reference to the Unicode library somehow, or put the unicode stuff into the `helper` namespace.

11.2.17 toLocaleLowerCase (self)

Description

1 The static `toLocaleLowerCase` method works exactly the same as the static `toLowerCase` method except that it is intended to yield the correct result for the host environment's current locale, rather than a locale-independent result. There will only be a difference in the few cases (such as Turkish) where the rules for that language conflict with the regular Unicode case mappings.

Returns

2 The static `toLocaleLowerCase` method returns a `string`.

Implementation

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

NOTE The second 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.

11.2.18 toUpperCase (self)

Description

1 The static `toUpperCase` method converts the characters of *self* (converted to string) to upper case. The characters are converted one by one. The result of each conversion is the original character, unless that character has a Unicode uppercase equivalent, in which case the uppercase equivalent is used instead.

NOTE The result should be derived according to the case mappings in the Unicode character database (this explicitly includes not only the `UnicodeData.txt` file, but also the `SpecialCasings.txt` file that accompanies it in Unicode 2.1.8 and later).

Returns

2 The static `toUpperCase` method returns a `string`.

Implementation

```
static function toUpperCase(self): string {
    let S   = string(self);
    let s   = "";

    for ( let i=0, limit=S.length ; i < limit ; i++ ) {
        let u = Unicode::toUpperCaseCharCode(charCodeAt(S,intrinsic::toUint(i)));
        if (u is double)
            s += fromCharCode(intrinsic::toUint(u));
        else {
            for ( let j=0 ; j < u.length ; j++ )
                s += fromCharCode(u[j]);
        }
    }
    return s;
}
```

NOTE Because both `toUpperCase` and `toLowerCase` have context-sensitive behaviour, the functions are not symmetrical. In other words, `string.toLowerCase(string.toUpperCase(s))` is not necessarily equal to `string.toLowerCase(s)`.

11.2.19 toLocaleUpperCase (self)

Description

1 The static `toLocaleUpperCase` method works exactly the same as the static `toUpperCase` method except that it is intended to yield the correct result for the host environment's current locale, rather than a locale-independent result. There

will only be a difference in the few cases (such as Turkish) where the rules for that language conflict with the regular Unicode case mappings.

Returns

2 The static `toLocaleUpperCase` method returns a `string`.

Implementation

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

NOTE The second 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.

11.2.20 trim (self)

Description

1 The static `trim` method extracts a substring from *self* (converted to `string`) such that the extracted string contains no whitespace characters at either end.

Returns

2 The static `trim` method returns a `string`.

Implementation

```
static function trim(s): string {
    s = string(s);

    let len = s.length;
    let i, j;

    for ( i=0 ; i < len && Unicode::isTrimmableSpace(s.charAt(i)) ; i++ )
        ;
    for ( j=len-1 ; j >= i && Unicode::isTrimmableSpace(s.charAt(j)) ; j-- )
        ;
    return s.substring(i,j+1);
}
```

FIXME (Ticket #179.) Reference to Unicode library -- handle this somehow.

11.3 Methods on `string` instances

11.3.1 intrinsic::toString

Returns

1 The intrinsic `toString` method returns this string value: the object itself.

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

11.3.2 intrinsic::valueOf

Returns

1 The intrinsic `valueOf` method returns this string value: the object itself.

```
override intrinsic function valueOf() : string
    this;
```

11.3.3 Methods that delegate to static methods

Description

1 The intrinsic methods `charAt`, `charCodeAt`, `concat`, `indexOf`, `lastIndexOf`, `localeCompare`, `match`, `replace`, `search`, `slice`, `split`, `substring`, `toLowerCase`, `toLocaleLowerCase`, `toUpperCase`, `toLocaleUpperCase`, and `trim` all delegate to the corresponding static methods on the `string` class.

Returns

2 These intrinsic methods return what their corresponding static methods on the `string` class return.

Implementation

```
intrinsic function charAt(pos: double = 0) : string
    string.charAt(this, pos);
```

```
intrinsic function charCodeAt(pos: double = 0) : double
    string.charCodeAt(this, pos);

intrinsic function concat(...args) : string
    string.concat(this, args);

intrinsic function indexOf(searchString: string, position: double = 0.0) : double
    string.indexOf(this, searchString, position);

intrinsic function lastIndexOf(searchString: string, position: double) : double
    string.lastIndexOf(this, searchString, position);

intrinsic function localeCompare(that : string) : double
    string.localeCompare(this, that);

intrinsic function match(regexp: RegExp) : Array
    string.match(this, regexp);

intrinsic function replace(searchValue: (string|RegExp!),
                           replaceValue: (string|function(...):string)) : string
    string.replace(this, searchValue, replaceValue);

intrinsic function search(regexp: RegExp!) : double
    string.search(this, regexp);

intrinsic function slice(start: AnyNumber=NaN, end: AnyNumber=NaN, step: AnyNumber=1):
string
    string.slice(this, start, end, step);

intrinsic function split(separator:(string|RegExp!), limit: double = double.MAX_VALUE):
Array!
    string.split(this, separator, limit)

    static function split(self, separator, limit) : Array! {
    type matcher = (string|RegExp!);

    let function splitMatch(R: matcher, S: string, q: double) : [double, [string]]? {
        switch type (R) {
            case (x: string) {
                let r = x.length;
                if (q + r <= S.length && S.substring(q, q + r) === R)
                    return [q+r, []];
                else
                    return null;
            }
            case (x: RegExp!) {
                let mr: MatchResult = x.match(S, q);
                if (mr === null)
                    return null;
                else
                    return [mr.endIndex, mr.captures];
            }
            }
    }

    let A   = new Array;
    let lim = limit === undefined ? double.MAX_VALUE : double(limit);
    let S   = string(self);
    let s   = S.length;
    let p   = 0;
    let R;

    if (separator !== null && separator is RegExp)
        R = separator;
    else
        R = string(separator);

    if (lim === 0)
        return A;

    if (separator === undefined) {
        A[0] = S;
        return A;
    }

    if (s === 0) {
        let z = splitMatch(R, S, 0);
        if (z === null)
            A[0] = S;
        return A;
    }

    for ( let q = p ; q !== s ; ) {
        let z = splitMatch(R, S, q);
        if (z === null) {
            ++q;
            continue;
        }

        let [e,cap] = z;
        if (e === p) {
            ++q;
            continue;
        }

        A[A.length] = S.substring(p, q);
        if (A.length === lim)
            return A;

        p = e;

        for ( let i=1 ; i < cap.length ; i++ ) {
            A[A.length] = cap[i];
            if (A.length === lim)
                return A;
        }

        q = p;
    }

    A[A.length] = S.substring(p, s);
    return A;
}

intrinsic function substring(start: double, end: double=this.length) : string
    string.substring(this, start, end);

intrinsic function toLowerCase() : string
    string.toLowerCase(this);

intrinsic function toLocaleLowerCase() : string
    string.toLowerCase(this);

intrinsic function toUpperCase() : string
    string.toUpperCase(this);

intrinsic function toLocaleUpperCase() : string
    string.toLocaleUpperCase(this);

intrinsic function trim() : string
    string.trim(this);
```

NOTE The second parameter to the intrinsic method `localeCompare` and the first parameter to the intrinsic methods `toLocaleLowerCase` and `toLocaleUpperCase` are likely to be used in a future version of this standard; it is recommended that implementations do not use these parameter position for anything else.

12 Boolean classes

1 ECMAScript provides a primitive truth value representation in the class `boolean`. It is primitive in the sense that this representation is directly operated upon by the operators of the language, and in the sense that the class `boolean` is a final and non-dynamic class for which ECMAScript implementations may provide efficient representations.

2 ECMAScript also provides the class `Boolean`, which is a dynamic non-final class that holds `boolean` values.

NOTE Instances of `Boolean` are not normally converted to `boolean` when operated upon by operators of the language.

12.1 The type `AnyBoolean`

1 The type `AnyBoolean` is a union containing all the built-in boolean types. By standard subtyping rules it also includes all classes that extend `Boolean`.

```
__ES4__ type AnyBoolean = (boolean|Boolean!);
```

13 The class `Boolean`

1 The class `Boolean` is a dynamic, nullable, non-final subclass of `Object` that holds a boolean value in the form of a `boolean` object. Instances of `Boolean` are converted to `boolean` when operated upon by operators of the language.

2 The class `Boolean` can be extended and the extending classes can provide novel representations for boolean values.

13.1 Synopsis

1 The class `Boolean` provides the following interface:

```
dynamic class Boolean
{
    function Boolean(x=false) : val = boolean(x) {} …
    static meta function invoke(x=false) : boolean …

    static const length: uint = 1

    override intrinsic function toString() : string …
    override intrinsic function valueOf() : boolean …
}
```

2 The `Boolean` prototype object provides the following direct properties:

```
    toString: function (this: Booleans) …
    valueOf:  function (this: Booleans) …
```

3 The `Boolean` prototype object is also the prototype object of the class `boolean`.

13.2 Methods on the `Boolean` class object

13.2.1 new Boolean (value=…)

Description

1 The `Boolean` constructor intializes a new `Boolean` object by storing *value*, converted to `boolean`, in a private property. The default *value* is **false**.

Implementation

2 The `Boolean` constructor is implementation-defined.

13.2.2 Boolean(value=…)

Description

1 The `Boolean` class object called as a function converts *value* to `boolean` (not `Boolean`).

Returns

2 The `boolean` class object called as a function returns a `boolean` object.

Implementation

```
static meta function invoke(x=false) : boolean
    boolean(x);
```

13.3 Methods on `Boolean` instances

13.3.1 intrinsic::toString ( )

Description

1 The intrinsic `toString` method converts this boolean value to a string.

Returns

2 The intrinsic `toString` method returns a string.

Implementation

```
override intrinsic function toString() : string
    intrinsic::valueOf().intrinsic::toString();
```

13.3.2 intrinsic::valueOf ( )

Description

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

Returns

2 The intrinsic `valueOf` method returns a `boolean` object (not a `Boolean` object).

Implementation

```
override intrinsic function valueOf() : boolean
    val;
```

13.4 Methods on the `Boolean` prototype object

Description

1 The methods on the `Boolean` prototype object invoke their intrinsic counterparts.

Returns

2 The methods on the `Boolean` prototype object return what their intrinsic counterparts return.

Implementation

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

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

14 The class `boolean`

1 The class `boolean` is a non-dynamic, non-nullable, final subclass of `Object`. It represents a boolean value (**true** or **false**).

COMPATIBILITY NOTE The class `boolean` is new in the 4th Edition of this Standard, but `boolean` models the "boolean values" in the 3rd Edition.

14.1 Synopsis

1 The class `boolean` provides the following interface:

```
final class boolean!
{
    function boolean(value=false) …
    static meta function invoke(x=false) : boolean …

    static const length: uint = 1

    override intrinsic function toString() : string …
    override intrinsic function valueOf() : boolean …
}
```

2 The `boolean` prototype object is the same as the `Boolean` prototype object (Boolean.prototype).

14.2 Methods on the `boolean` class object

14.2.1 new boolean (value=…)

Description

1 The `boolean` constructor intializes a new `boolean` object by storing an implementation-dependent representation of the truth value of *value*, as computed by `ToBoolean` (see ToBoolean), in a private property. The default *value* is **false**.

Implementation

2 The `boolean` constructor is implementation-defined.

14.2.2 boolean(value=…)

Description

1 The `boolean` class object called as a function converts *value* to `boolean`.

Returns

2 The `boolean` class object called as a function returns a `boolean` object.

Implementation

```
static meta function invoke(x=false) : boolean
    (x is boolean) ? x : new boolean(x);
```

14.3 Methods on `boolean` instances

14.3.1 intrinsic::toString ( )

Description

1 The intrinsic `toString` method converts this boolean value to a string, either `"true"` or `"false"`.

Returns

2 The intrinsic `toString` method returns the string.

Implementation

```
override intrinsic function toString() : string
    this ? "true" : "false";
```

14.3.2 intrinsic::valueOf ( )

Description

1 The intrinsic `valueOf` method returns a `boolean` instance: the object on which the method was invoked.

Returns

2 The intrinsic `valueOf` method returns its `this` object.

Implementation

```
override intrinsic function valueOf() : boolean
    this;
```

15 Number classes

1 ECMAScript provides a variety of primitive number representations. They are primitive in the sense that these are the representations directly operated upon by the operators of the language, and also in the sense that they are represented by final non-dynamic classes for which ECMAScript implementations may provide efficient representations.

2 The class `double` represents 64-bit IEEE-format binary floating point numbers approximately in the range -1.7976931348623157 × 10^{308} to +1.7976931348623157 × 10^{308}.

3 The class `decimal` represents 128-bit IEEE-format decimal floating point numbers in the range -(10^{34}-1) × 10^{6111} to (10^{34}-1) × 10^{6111}.

COMPATIBILITY NOTE The 3rd Edition of this Standard provided only one kind of primitive number value, represented as 64-bit IEEE-format binary floating point.

4 ECMAScript also provides the class `Number`, which is a dynamic non-final class that represents 64-bit IEEE-format binary floating point numbers.

15.1 The type `AnyNumber`

1 The type `AnyNumber` is a union type that contains all the number types in the language.

```
__ES4__ type AnyNumber = (double|decimal|Number!);
```

16 The class `Number`

1 The class `Number` is a dynamic, nullable, non-final direct subclass of `Object` that holds a `double` value.

2 All intrinsic methods of `Number` obtain the number value stored in the object by calling the intrinsic `valueOf` method. If the class `Number` is extended then the extending class can override the intrinsic `valueOf` method in order to provide new ways of representing the number value stored in the class.

3 The intrinsic `valueOf` method is not constrained to return a `double` value, it can return any primitive number type.

16.1 Synopsis

1 The class `Number` provides the following interface:

```
dynamic class Number
{
    function Number(value=0) …
    static meta function invoke(value=0) …

    static const MAX_VALUE: double = double.MAX_VALUE
    static const MIN_VALUE: double = double.MIN_VALUE
    static const NaN: double = double.NAN
    static const NEGATIVE_INFINITY: double = double.NEGATIVE_INFINITY
    static const POSITIVE_INFINITY: double = double.POSITIVE_INFINITY
    static const length: uint = 1

    override intrinsic function toString(radix = 10) : string …
    override intrinsic function toLocaleString() : string …
    override intrinsic function valueOf(): (double|decimal) …

    intrinsic function toFixed(fractionDigits=0): string …
    intrinsic function toExponential(fractionDigits=undefined) : string …
    intrinsic function toPrecision(precision=undefined) : string …
}
```

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

```
    toString:       function (this: Numeric, radix) …
    toLocaleString: function (this: Numeric) …
    valueOf:        function (this: Numeric) …
    toFixed:        function (this: Numeric, fractionDigits) …
    toExponential:  function (this: Numeric, fractionDigits) …
    toPrecision:    function (this: Numeric, precision) …
```

16.2 Methods on the `Number` class object

16.2.1 new Number(value=…)

Description

1 The `Number` constructor initialises the newly created `Number` object by storing *value* (which defaults to +0), converted to `double`, in a private property.

FIXME It is likely that `Number` should not be constrained to hold `double` values, but that it should be able to hold any numeric type and that its methods should work properly on any numeric type, in a type-specific manner.

16.2.2 Number(value=…)

Description

1 When the `Number` class object is called as a function it performs a type conversion: if *value* (which defaults to +0) is not a primitive number type it is converted to `double`.

Returns

2 The `Number` class object called as a function returns *value* converted to a primitive number type.

Implementation

```
static meta function invoke(value=0) {
    if (value is AnyNumber)
        return value;
    return double(value);
}
```

16.3 Value properties on the `Number` class object

16.3.1 MAX_VALUE

1 The value of `MAX_VALUE` is the largest positive finite value represented by the `double` class.

16.3.2 MIN_VALUE

1 The value of `MIN_VALUE` is the smallest positive value represented by the `double` class.

16.3.3 NaN

1 The value of `NaN` is the not-a-number value represented by the `double` class.

16.3.4 NEGATIVE_INFINITY

1 The value of `NEGATIVE_INFINITY` is the value -∞ as represented by a `double` object.

16.3.5 POSITIVE_INFINITY

1 The value of `POSITIVE_INFINITY` is the value +∞ as represented by a `double` object.

16.4 Methods on `Number` instances

16.4.1 intrinsic::toString (radix=…)

Description

1 The intrinsic `toString` method converts this number value to a string representation in a base given by *radix*.

2 If radix is the number 10 or undefined, then the result is as for the `ToString` operator.

3 If radix is an integer from 2 to 36, but not 10, the result is an implementation-dependent string

Returns

4 The intrinsic `toString` method returns a string.

Implementation

```
override intrinsic function toString(radix = 10) : string
    intrinsic::valueOf().intrinsic::toString(radix);
```

NOTE The intrinsic `toString` method operates by obtaining a primitive number value, which it then converts to string by invoking the intrinsic `toString` method on the primitive value.

16.4.2 intrinsic::toLocaleString ( )

Description

1 The intrinsic `toLocaleString` method converts this number value to a string value that represents the number value formatted according to the conventions of the host environment's current locale.

Returns

2 The intrinsic `toLocaleString` method returns an implementation-dependent string.

Implementation

3 The intrinsic `toLocaleString` method is implementation-dependent, and it is permissible, but not encouraged, for it to return the same thing as the intrinsic `toString` method.

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.

16.4.3 intrinsic::valueOf ( )

Description

1 The intrinsic `valueOf` method returns the number value represented by this `Number` object.

Returns

2 The intrinsic `valueOf` method returns a primitive number value.

Implementation

```
override intrinsic function valueOf(): (double|decimal)
    val;
```

16.4.4 Intrinsic methods that delegate to methods on primitive types

Description

1 The intrinsic `toFixed`, `toExponential`, and `toPrecision` methods operate by obtaining a primitive number value from the intrinsic `valueOf` method, then invoking the appropriate method on the primitive value.

Returns

2 The intrinsic `toFixed`, `toExponential`, and `toPrecision` methods return what their delegates return.

Implementation

```
intrinsic function toFixed(fractionDigits=0): string
    intrinsic::valueOf().intrinsic::toFixed(fractionDigits);

intrinsic function toExponential(fractionDigits=undefined) : string
    intrinsic::valueOf().intrinsic::toExponential(fractionDigits);

intrinsic function toPrecision(precision=undefined) : string
    intrinsic::valueOf().intrinsic::toPrecision(precision);
```

16.5 Methods on the `Number` prototype object

Description

1 The methods on the `Number` prototype object are constrained to being invoked on members of the type `Numeric`. All operate by calling the corresponding intrinsic method on the `this` object.

NOTE The `Number` prototype object is also the prototype object for `int`, `uint`, `double`, and `decimal`.

Returns

2 The methods on the `Number` prototype object return what their corresponding intrinsic methods return.

Implementation

```
prototype function toString(this: AnyNumber, radix=10)
    this.intrinsic::toString(radix);

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

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

prototype function toFixed(this:AnyNumber, fractionDigits)
    this.intrinsic::toFixed(fractionDigits);

prototype function toExponential(this: AnyNumber, fractionDigits)
    this.intrinsic::toExponential(fractionDigits);

prototype function toPrecision(this: AnyNumber, precision)
    this.intrinsic::toPrecision(precision);
```

17 The class `double`

1 The class `double` is a final, non-nullable, non-dynamic direct subclass of `Object` that represents 64-bit ("double precision") IEEE binary floating point number values in the range $-(1-(1/2)^{53}) \times 2^{1024}$ to $+(1-(1/2)^{53}) \times 2^{1024}$ inclusive (approximately the range $-1.7976931348623157 \times 10^{308}$ to $+1.7976931348623157 \times 10^{308}$, inclusive), plus the three special values -∞, +∞, and NaN.

COMPATIBILITY NOTE The class `double` is new in the 4th Edition of this Standard, but `double` models the "number values" in the 3rd Edition.

17.1 Synopsis

1 The class `double` provides the following interface:

```
final class double!
{
    function double(value=false) …
    static meta function invoke(x=0d) …

    static const MAX_VALUE: double = …
    static const MIN_VALUE: double = …
    static const NaN: double = …
    static const NEGATIVE_INFINITY: double = …
    static const POSITIVE_INFINITY: double = …
    static const E: double = …
    static const LN10: double = …
```

```
        static const LN2: double = …
        static const LOG2E: double = …
        static const LOG10E: double = …
        static const PI: double = …
        static const SQRT1_2: double = …
        static const SQRT2: double = …
        static const length: uint = 1

        override intrinsic function toString(radix = 10) : string …
        override intrinsic function toLocaleString() : string …
        override intrinsic function valueOf() : double …

        intrinsic function toFixed(fractionDigits=0) : string …
        intrinsic function toExponential(fractionDigits=undefined) : string …
        intrinsic function toPrecision(precision=undefined) : string …
    }
```

2 The `double` prototype object is identical to the `Number` prototype object (Number.prototype).

17.2 Methods on the `double` class object

17.2.1 new double(value=…)

Description

1 The `double` constructor initialises the newly created `double` object by storing an implementation-dependent representation of the double-precision value of *value*, converted to a number by the ToNumber operator, in a private property. The default *value* is 0.

Implementation

2 The `double` constructor is implementation-dependent.

17.2.2 double(value=…)

Description

1 When the `double` class object is called as a function it performs a type conversion: it converts *value* (which defaults to +0) to `double`.

Returns

2 The `double` class object called as a function returns *value* converted to `double`.

Implementation

```
static meta function invoke(x=0d)
    (x is double) ? x : new double(x);
```

FIXME (Ticket #176.) The optimization used here, `magic::newDouble` for `new double`, makes the spec harder than it needs to be.

17.3 Value properties on the `double` class object

17.3.1 MAX_VALUE

1 The value of `MAX_VALUE` is the largest positive finite value represented by the `double` class, type, which is approximately $1.7976931348623157 \times 10^{308}$.

17.3.2 MIN_VALUE

1 The value of `MIN_VALUE` is the smallest positive value represented by the `double` class, which is approximately 5×10^{-324}.

17.3.3 NaN

1 The value of `NaN` is the not-a-number value represented by a `double` instance.

17.3.4 NEGATIVE_INFINITY

1 The value of `NEGATIVE_INFINITY` is the value -∞ as represented by a `double` instance.

17.3.5 POSITIVE_INFINITY

1 The value of `POSITIVE_INFINITY` is the value +∞ as represented by a `double` instance.

17.3.6 E

1 The value of `E` is the `double` value for *e*, the base of the natural logarithms, which is approximately 2.7182818284590452354.

17.3.7 LN10

1 The value of `LN10` is the `double` value for the natural logarithm of 10, which is approximately 2.302585092994046.

17.3.8 LN2

1 The value of `LN2` is the `double` value for the natural logarithm of 2, which is approximately 0.6931471805599453.

17.3.9 LOG2E

1 The value of `LOG2E` is the `double` value for the base-2 logarithm of *e*, the base of the natural logarithms; this value is approximately 1.4426950408889634.

NOTE The value of `double.LOG2E` is approximately the reciprocal of the value of `double.LN2`.

17.3.10 LOG10E

1 The value of `LOG10E` is the double value for the base-10 logarithm of *e*, the base of the natural logarithms; this value is approximately 0.4342944819032518.

NOTE The value of `double.LOG10E` is approximately the reciprocal of the value of `double.LN10`.

17.3.11 PI

1 The value of `PI` is the `double` value for π, the ratio of the circumference of a circle to its diameter, which is approximately 3.1415926535897932.

17.3.12 SQRT1_2

1 The value of `SQRT1_2` is the `double` value for the square root of 1/2, which is approximately 0.7071067811865476.

NOTE The value of `double.SQRT1_2` is approximately the reciprocal of the value of `double.SQRT2`.

17.3.13 SQRT2

1 The value of `SQRT2` is the `double` value for the square root of 2, which is approximately 1.4142135623730951.

17.4 Methods on `double` instances

17.4.1 intrinsic::toString (radix=…)

Description

1 The intrinsic `toString` method converts this number value to a string representation in a base given by *radix*.

2 If radix is the number 10 or undefined, then the result is as for the `ToString` operator.

3 If radix is an integer from 2 to 36, but not 10, the result is an implementation-dependent string

Returns

4 The intrinsic `toString` method returns a string.

Implementation

```
override intrinsic function toString(radix = 10) : string {
    if (radix === 10 || radix === undefined)
        return string(this);
    if (radix is AnyNumber && radix >= 2 && radix <= 36 && intrinsic::isIntegral(radix))
```

```
        return toString(int(radix));
    throw new TypeError("Invalid radix argument to double.toString");
}
```

17.4.2 intrinsic::toLocaleString ( )

Description

1 The intrinsic `toLocaleString` method converts this number value to a string value that represents the value of the integer formatted according to the conventions of the host environment's current locale.

Returns

2 The intrinsic `toLocaleString` method returns an implementation-dependent string.

Implementation

3 The intrinsic `toLocaleString` method is implementation-dependent, and it is permissible, but not encouraged, for it to return the same thing as the intrinsic `toString` method.

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.

17.4.3 intrinsic::valueOf ( )

Description

1 The intrinsic `valueOf` method returns the number value represented by this `double` object: the object itself.

Returns

2 The intrinsic `valueOf` method returns its `this` object.

Implementation

```
override intrinsic function valueOf() : double
    this;
```

17.4.4 intrinsic::toFixed (fractionDigits=…)

Description

1 The intrinsic `toFixed` method converts the this number value to a string in fixed-point notation with *fractionDigits* digits after the decimal point. If *fractionDigits* is **undefined**, 0 is assumed.

Returns

2 The intrinsic `toFixed` method returns the fixed-point notation string representation of this number value.

Implementation

```
intrinsic function toFixed(fractionDigits=0) : string {
    let x = this;
    let f = toInteger(fractionDigits);
    if (f < 0 || f > 20)
        throw new RangeError();

    if (isNaN(x))
        return "NaN";
    let s = "";
    if (x < 0) {
        s = "-";
        x = -x;
    }

    if (x >= Math.pow(10,21))
        return s + string(m);

    let n = toFixedStep10(x, f);
    let m = n == 0 ? "0" : string(n);
    if (f == 0)
        return s + m;
    let k = m.length;
    if (k <= f) {
        m = "00000000000000000000".substring(0,f+1-k) + m;
        k = f+1;
    }
    return s + m.substring(0,k-f) + "." + m.substring(k-f);
}
```

FIXME (Ticket #184.) Note that calling anything "step 10" no longer makes sense.

Note also that "step 10" is informative and needs to be documented and implemented as such.

3 An implementation is permitted to extend the behaviour of toFixed for values of *fractionDigits* less than 0 or greater than 20. In this case `toFixed` would not necessarily throw **RangeError** for such values.

NOTE The output of `toFixed` may be more precise than `toString` for some values because `toString` only prints enough significant digits to distinguish the number from adjacent number values. For example, `(1000000000000000128).toString()` returns "1000000000000000100", while `(1000000000000000128).toFixed(0)` returns "1000000000000000128".

17.4.5 intrinsic::toExponential (fractionDigits=…)

Description

1 The intrinsic `toExponential` method converts this number value to a string in exponential notation with one digit before the significand's decimal point and *fractionDigits* digits after the significand's decimal point. If *fractionDigits* is **undefined**, include as many significand digits as necessary to uniquely specify the number (just like in `ToString` except that in this case the number is always output in exponential notation).

Returns

2 The intrinsic `toExponential` method returns the exponential notation string representation of this number value.

Implementation

```
intrinsic function toExponential(fractionDigits=undefined) : string {
    return "**toExponential: FIXME**";
}
```

FIXME (Ticket #185.) Implement this function.

3 An implementation is permitted to extend the behaviour of `toExponential` for values of *fractionDigits* less than 0 or greater than 20. In this case `toExponential` would not necessarily throw **RangeError** for such values.

NOTE For implementations that provide more accurate conversions than required by the rules above, it is recommended that the following alternative version of step 19 be used as a guideline:

Let e, n, and f be integers such that $f \geq 0$, $10^f \leq n < 10^{f+1}$, the number value for $n \times 10^{e-f}$ is *x*, and *f* is as small as possible. If there are multiple possibilities for *n*, choose the value of *n* for which $n \times 10^{e-f}$ is closest in value to *x*. If there are two such possible values of *n*, choose the one that is even.

FIXME (Ticket #186.) "Step 19" is obsolete.

17.4.6 intrinsic::toPrecision (precision=…)

Description

1 The intrinsic `toPrecision` method converts this number value to a string, either in exponential notation with one digit before the significand's decimal point and *precision*-1 digits after the significand's decimal point or in fixed notation with *precision* significant digits. If precision is **undefined**, call `ToString` (operator:ToString) instead.

Returns

2 The intrinsic `toPrecision` method returns the selected string representation of this number value.

Implementation

```
intrinsic function toPrecision(precision=undefined) : string {
    return "**toPrecision: FIXME**";
}
```

FIXME (Ticket #185.) Implement this function.

3 An implementation is permitted to extend the behaviour of `toPrecision` for values of precision less than 1 or greater than 21. In this case `toPrecision` would not necessarily throw **RangeError** for such values.

18 The class `decimal`

1 The class `decimal` is a final, non-nullable, non-dynamic direct subclass of `Object` that represents 128-bit IEEE decimal floating point number values in the range $-(10^{34}-1) \times 10^{6111}$ to $(10^{34}-1) \times 10^{6111}$ inclusive, plus the three special values -∞, +∞, and NaN.

COMPATIBILITY NOTE The class `decimal` is new in the 4th Edition of this Standard.

18.1 Synopsis

1 The class `decimal` provides the following interface:

```
final class decimal!
{
    function decimal(value=false) …
    static meta function invoke(x=0m) …

    static const MAX_VALUE: decimal = …
    static const MIN_VALUE: decimal = …
    static const NaN: decimal = …
    static const NEGATIVE_INFINITY: decimal = …
    static const POSITIVE_INFINITY: decimal = …
    static const E: decimal = …
    static const LN10: decimal = …
    static const LN2: decimal = …
    static const LOG2E: decimal = …
    static const LOG10E: decimal = …
    static const PI: decimal = …
    static const SQRT1_2: decimal = …
    static const SQRT2: decimal = …
    static const length: uint = 1

    override intrinsic function toString(radix = 10) : string …
    override intrinsic function toLocaleString() : string …
    override intrinsic function valueOf() : decimal …

    intrinsic function toFixed(fractionDigits=0) : string …
    intrinsic function toExponential(fractionDigits=undefined) : string …
    intrinsic function toPrecision(precision=undefined) : string …
}
```

2 The `decimal` prototype object is identical to the `Number` prototype object (Number.prototype).

18.2 Methods on the `decimal` class object

18.2.1 new decimal(value=…)

Description

1 The `decimal` constructor initialises the newly created `decimal` object by storing an implementation-dependent representation of the decimal value of *value*, as converted by `ToNumber`, in a private property. The default *value* is +0.

Implementation

2 The `decimal` constructor is implementation-dependent.

18.2.2 decimal(value=…)

Description

1 When the `decimal` class object is called as a function it performs a type conversion: it converts *value* (which defaults to +0) to `decimal`.

Returns

2 The `decimal` class object called as a function returns *value* converted to `decimal`.

Implementation

```
static meta function invoke(x=0m)
    (x is decimal) ? x : new decimal(x);
```

18.3 Value properties on the `decimal` class object

18.3.1 MAX_VALUE

1 The value of `MAX_VALUE` is the largest positive finite value represented by the `decimal` class, type, which is $(10^{34}-1) \times 10^{6111}$.

18.3.2 MIN_VALUE

1 The value of `MIN_VALUE` is the smallest positive value represented by the `decimal` class, which is 10^{-6143}.

18.3.3 NaN

1 The value of `NaN` is the not-a-number value represented by a `decimal` instance.

18.3.4 NEGATIVE_INFINITY

1 The value of `NEGATIVE_INFINITY` is the value -∞ as represented by a `decimal` instance.

18.3.5 POSITIVE_INFINITY

1 The value of `POSITIVE_INFINITY` is the value +∞ as represented by a `decimal` instance.

18.3.6 E

1 The value of `E` is the `decimal` value for *e*, the base of the natural logarithms, which is approximately 2.718281828459045235360287471352662.

18.3.7 LN10

1 The value of `LN10` is the `decimal` value for the natural logarithm of 10, which is approximately 2.302585092994045684017991454684364.

18.3.8 LN2

1 The value of `LN2` is the `decimal` value for the natural logarithm of 2, which is approximately 0.6931471805599453094172321214581766.

18.3.9 LOG2E

1 The value of `LOG2E` is the `decimal` value for the base-2 logarithm of *e*, the base of the natural logarithms; this value is approximately 1.442695040888963407359924681001892.

NOTE The value of `decimal.LOG2E` is approximately the reciprocal of the value of `decimal.LN2`.

18.3.10 LOG10E

1 The value of `LOG10E` is the decimal value for the base-10 logarithm of *e*, the base of the natural logarithms; this value is approximately 0.4342944819032518276511289189166051.

NOTE The value of `decimal.LOG10E` is approximately the reciprocal of the value of `decimal.LN10`.

18.3.11 PI

1 The value of `PI` is the `decimal` value for π, the ratio of the circumference of a circle to its diameter, which is approximately 3.141592653589793238462643383279503.

18.3.12 SQRT1_2

1 The value of `SQRT1_2` is the `decimal` value for the square root of 1/2, which is approximately 0.7071067811865475244008443621048490.

NOTE The value of `decimal.SQRT1_2` is approximately the reciprocal of the value of `decimal.SQRT2`. 1/sqrt(2) = 0.7071067811865475244008443621048490392848359376884740365883398689953674 as computed by math.sch -->

18.3.13 SQRT2

1 The value of `SQRT2` is the `decimal` value for the square root of 2, which is approximately 1.414213562373095048801688724209698.

18.4 Methods on `decimal` instances

18.4.1 intrinsic::toString (radix=…)

Description

1 The intrinsic `toString` method converts this number value to a string representation in a base given by *radix.*

2 If radix is the number 10 or undefined, then the result is as for the `ToString` operator.

3 If radix is an integer from 2 to 36, but not 10, the result is an implementation-dependent string

Returns

4 The intrinsic `toString` method returns a string.

Implementation

```
override intrinsic function toString(radix = 10) : string {
    if (radix === 10 || radix === undefined)
        return string(this);
    if (radix is AnyNumber && radix >= 2 && radix <= 36 && intrinsic::isIntegral(radix))
        return toString(int(radix));
    throw new TypeError("Invalid radix argument to decimal.toString");
}
```

18.4.2 intrinsic::toLocaleString ( )

Description

1 The intrinsic `toLocaleString` method converts this number value to a string value that represents the value of the integer formatted according to the conventions of the host environment's current locale.

Returns

2 The intrinsic `toLocaleString` method returns an implementation-dependent string.

Implementation

3 The intrinsic `toLocaleString` method is implementation-dependent, and it is permissible, but not encouraged, for it to return the same thing as the intrinsic `toString` method.

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.

18.4.3 intrinsic::valueOf ( )

Description

1 The intrinsic `valueOf` method returns the number value represented by this `decimal` object: the object itself.

Returns

2 The intrinsic `valueOf` method returns its `this` object.

Implementation

```
override intrinsic function valueOf() : decimal
    this;
```

18.4.4 intrinsic::toFixed (fractionDigits=…)

Description

1 The intrinsic `toFixed` method converts this number value to a string in fixed-point notation with *fractionDigits* digits after the decimal point. If *fractionDigits* is **undefined**, 0 is assumed.

Returns

2 The intrinsic `toFixed` method returns the fixed-point notation string representation of this number value.

Implementation

```
intrinsic function toFixed(fractionDigits=0) : string
    double(this).intrinsic::toFixed(fractionDigits);
```

FIXME (Ticket #188.) That implementation is bogus.

3 An implementation is permitted to extend the behaviour of toFixed for values of *fractionDigits* less than 0 or greater than 20. In this case `toFixed` would not necessarily throw **RangeError** for such values.

FIXME (Ticket #188.) Greater number of fractionDigits is possible for decimal.

NOTE The output of `toFixed` may be more precise than `toString` for some values because `toString` only prints enough significant digits to distinguish the number from adjacent number values. For example, `(1000000000000000128).toString()` returns "1000000000000000100", while `(1000000000000000128).toFixed(0)` returns "1000000000000000128".

FIXME (Ticket #188.) Better example / more precision (for decimal).

18.4.5 intrinsic::toExponential (fractionDigits=…)

Description

1 The intrinsic `toExponential` method converts this number value to a string in exponential notation with one digit before the significand's decimal point and *fractionDigits* digits after the significand's decimal point. If *fractionDigits* is **undefined**, include as many significand digits as necessary to uniquely specify the number (just like in `ToString` except that in this case the number is always output in exponential notation).

Returns

2 The static `toExponential` method returns the exponential notation string representation of this number value.

Implementation

```
intrinsic function toExponential(fractionDigits=undefined) : string
    double(this).intrinsic::toExponential(fractionDigits);
```

FIXME (Ticket #188.) That implementation is bogus.

3 An implementation is permitted to extend the behaviour of `toExponential` for values of *fractionDigits* less than 0 or greater than 20. In this case `toExponential` would not necessarily throw **RangeError** for such values.

FIXME (Ticket #188.) Greater number of fractionDigits is possible for decimal.

NOTE For implementations that provide more accurate conversions than required by the rules above, it is recommended that the following alternative version of step 19 be used as a guideline:

Let e, n, and f be integers such that $f \geq 0$, $10^f \leq n < 10^{f+1}$, the number value for $n \times 10^{e-f}$ is x, and f is as small as possible. If there are multiple possibilities for n, choose the value of n for which $n \times 10^{e-f}$ is closest in value to x. If there are two such possible values of n, choose the one that is even.

FIXME (Ticket #188.) "Step 19" is obsolete.

18.4.6 intrinsic::toPrecision (precision=…)

Description

1 The intrinsic `toPrecision` method converts this number value to a string, either in exponential notation with one digit before the significand's decimal point and *precision*-1 digits after the significand's decimal point or in fixed notation with *precision* significant digits. If precision is **undefined**, call `ToString` (operator:ToString) instead.

Returns

2 The intrinsic `toPrecision` method returns the selected string representation of this number value.

Implementation

```
intrinsic function toPrecision(precision=undefined) : string
    double(this).intrinsic::toPrecision(precision);
```

FIXME (Ticket #188.) That implementation is bogus.

3 An implementation is permitted to extend the behaviour of `toPrecision` for values of precision less than 1 or greater than 21. In this case `toPrecision` would not necessarily throw **RangeError** for such values.

FIXME (Ticket #188.) Greater precision possible for decimal.

19 The `Math` Object

1 The global `Math` object is a single object that has some named properties, some of which are functions. The `Math` object is the only instance of an internal helper class called `Math`.

2 The `Math` object acts as a container for built-in mathematics-related functions and constants.

19.1 Synopsis

1 For convenience of notation the definition of the Math object uses the helper type name `PrimitiveNumber`.

```
helper type PrimitiveNumber = (double|decimal);
```

2 The intrinsic methods on the math object are restricted to arguments of the type `PrimitiveNumber`.

3 The `Math` object provides the following interface:

```
helper dynamic final class Math extends Object
{
    intrinsic function abs(x: helper::PrimitiveNumber): helper::PrimitiveNumber …
    intrinsic function acos(x: helper::PrimitiveNumber): helper::PrimitiveNumber …
    intrinsic function atan(x: helper::PrimitiveNumber): helper::PrimitiveNumber …
    intrinsic function atan2(y: helper::PrimitiveNumber, x: helper::PrimitiveNumber):
helper::PrimitiveNumber …
    intrinsic function ceil(x: helper::PrimitiveNumber): helper::PrimitiveNumber …
    intrinsic function cos(x: helper::PrimitiveNumber): helper::PrimitiveNumber …
    intrinsic function exp(x: helper::PrimitiveNumber): helper::PrimitiveNumber …
    intrinsic function floor(x: helper::PrimitiveNumber): helper::PrimitiveNumber …
    intrinsic function log(x: helper::PrimitiveNumber): helper::PrimitiveNumber …
    intrinsic function max(x: helper::PrimitiveNumber, y: helper::PrimitiveNumber):
helper::PrimitiveNumber …
    intrinsic function min(x: helper::PrimitiveNumber, y: helper::PrimitiveNumber):
helper::PrimitiveNumber …
    intrinsic function pow(x: helper::PrimitiveNumber, y: helper::PrimitiveNumber):
helper::PrimitiveNumber …
    intrinsic function random(): double …
    intrinsic function round(x: helper::PrimitiveNumber): helper::PrimitiveNumber …
    intrinsic function sin(x: helper::PrimitiveNumber): helper::PrimitiveNumber …
    intrinsic function sqrt(x: helper::PrimitiveNumber): helper::PrimitiveNumber …
    intrinsic function tan(x: helper::PrimitiveNumber): helper::PrimitiveNumber …

    const E: double = double.E
    const LN10: double = double.LN10
    const LN2: double = double.LN2
    const LOG2E: double = double.LOG2E
    const LOG10E: double = double.LOG10E
    const PI: double = double.PI
    const SQRT1_2: double = double.SQRT1_2
    const SQRT2: double = double.SQRT2
}
```

4 The constant values `E`, `LN10`, `LN2`, `LOG2E`, `LOG10E`, `PI`, `SQRT1_2`, and `SQRT2` in the `Math` class are of type `double` for compatibility with 3rd Edition.

NOTE New code may find it more convenient to access these constant values through the `double` or `decimal` classes, as appropriate, to obtain values with the best precision for the particular type.

5 The `Math` object additionally provides the following dynamic function properties. These functions are not restricted in the types of arguments they accept, but convert all their arguments to a primitive number.

```
    abs:    function (x) … ,
    acos:   function (x) … ,
    asin:   function (x) … ,
    atan:   function (x) … ,
    atan2:  function (y,x) … ,
    ceil:   function (x) … ,
    cos:    function (x) … ,
    exp:    function (x) … ,
    floor:  function (x) … ,
    log:    function (x) … ,
    max:    function (...xs) … ,
    min:    function (...xs) … ,
    pow:    function (x,y) … ,
    random: function () … ,
    round:  function (x) … ,
    sin:    function (x) … ,
    sqrt:   function (x) … ,
    tan:    function (x) …
```

6 The `[[Prototype]]` object of the `Math` object does not contain a `constructor` property.

NOTE The constraint on `constructor` is for backward compatibility and is also necessary to insure that the math object is a singleton object. But note that `Math.constructor` is still defined, it is accessible through the prototype chain and is `Object.constructor`.

19.2 Primitive operations on numbers

FIXME (Ticket #189.) Describe the following helper and informative functions here: isPositive, isPositiveZero, isNegativeZero, isOddInteger.

19.3 Intrinsic function properties of the Math object

1 In the function descriptions below, the symbols NaN, -0, +0, -∞ and +∞ refer to the number values described in 8.5.

FIXME Clean up the cross-reference later.

NOTE The behaviour of the functions `acos`, `asin`, `atan`, `atan2`, `cos`, `exp`, `log`, `pow`, `sin`, and `sqrt` is not precisely specified here except to require specific results for certain argument values that represent boundary cases of interest. For other argument values, these functions are intended to compute approximations to the results of familiar mathematical functions, but some latitude is allowed in the choice of approximation algorithms. The general intent is that an implementer should be able to use the same mathematical library for ECMAScript on a given hardware platform that is available to C programmers on that platform.

Although the choice of algorithms is left to the implementation, it is recommended (but not specified by this standard) that implementations use the approximation algorithms for IEEE 754 arithmetic contained in fdlibm, the freely distributable mathematical library from Sun Microsystems (`fdlibm-comment@sunpro.eng.sun.com`). This specification also requires specific results for certain argument values that represent boundary cases of interest.

NOTE The functions defined in this section preserve the representation of the argument(s) in the result where this is reasonable. All functions map `double` and `decimal` arguments to `double` and `decimal` results, respectively.

19.3.1 intrinsic::abs (x)

Description

1 The intrinsic `abs` function computes the absolute value of the number *x*, which has the same magnitude as *x* but has positive sign.

Returns

2 The intrinsic `abs` function returns the absolute value of *x*. The representation of the result is the same as the representation of *x*.

Implementation

3
```
intrinsic function abs(x: PrimitiveNumber): PrimitiveNumber {
    switch type (x) {
    case (n: double) {
        if (isNaN(n)) return n;
        if (x == 0) return 0;
        return n < 0 ? -n : n;
    }
    case (n: decimal) {
        if (isNaN(n)) return n;
        if (x == 0m) return 0m;
        return n < 0m ? -n : n;
    }
    }
}
```

19.3.2 intrinsic::acos (x)

Description

1 The intrinsic `acos` function computes an implementation-dependent approximation to the arc cosine of the number *x*. The result is expressed in radians and ranges from +0 to +π.

Returns

2 The intrinsic `acos` function returns a floating-point number.

Implementation

```
intrinsic function acos(x: PrimitiveNumber): PrimitiveNumber {
    switch type (x) {
    case (n: double) {
        if (isNaN(n) || n > 1 || n < -1) return NaN;
        if (n == 1) return 0;
        return acosDouble(n);
    }
    case (n: decimal) {
        if (isNaN(n) || n > 1m || n < 1m) return decimal.NaN;
        if (n == 1m) return 0m;
        return acosDecimal(n);
    }
    }
}
```

3 The informative functions `acosDouble` and `acosDecimal` implement representation-preserving approximate computation of the arc cosine of their argument.

```
informative function acosDouble(x: double): double …
informative function acosDecimal(x: decimal): decimal …
```

19.3.3 intrinsic::asin (x)

Description

1 The intrinsic `asin` function computes an implementation-dependent approximation to the arc sine of the number *x*. The result is expressed in radians and ranges from -π/2 to +π/2.

Returns

2 The intrinsic `asin` function returns a floating-point number.

Implementation

```
intrinsic function asin(x: PrimitiveNumber): PrimitiveNumber {
    switch type (x) {
    case (n: double) {
        if (isNaN(n) || n > 1 || n < -1) return NaN;
        if (n == 0) return n;
        return asinDouble(n);
    }
    case (n: decimal) {
        if (isNaN(n) || n > 1m || n < 1m) return decimal.NaN;
        if (n == 0m) return n;
        return asinDecimal(n);
    }
    }
}
```

NOTE The intrinsic `asin` function preserves the sign of *x* if x is 0.

3 The informative functions `asinDouble` and `asinDecimal` implement representation-preserving approximate computation of the arc sine of their argument.

```
informative function asinDouble(x: double): double …
informative function asinDecimal(x: decimal): decimal …
```

19.3.4 intrinsic::atan (x)

Description

1 The intrinsic `atan` function computes an implementation-dependent approximation to the arc tangent of the number *x*. The result is expressed in radians and ranges from -π/2 to +π/2.

Returns

2 The intrinsic `atan` function returns a floating-point number.

Implementation

```
intrinsic function atan(x: PrimitiveNumber): PrimitiveNumber {
    switch type (x) {
    case (n: double) {
        if (isNaN(n) || n == 0) return n;
        if (!isFinite(n))
            return copysign(double.PI / 2, n);
        return atanDouble(n);
    }
    case (n: decimal) {
        if (isNaN(n) || n == 0m) return n;
        if (!isFinite(n))
            return copysign(decimal.PI / 2m, n);
        return atanDecimal(n);
    }
    }
}
```

NOTE The intrinsic `atan` function preserves the sign of *x* if x is 0.

3 The informative functions `atanDouble` and `atanDecimal` implement representation-preserving approximate computation of the arc tangent of their argument.

```
informative function atanDouble(x: double): double …
informative function atanDecimal(x: decimal): decimal …
```

19.3.5 intrinsic::atan2 (y, x)

Description

1 The intrinsic `atan2` function computes an implementation-dependent approximation to the arc tangent of the quotient *y*/*x* of the numbers *y* and *x*, where the signs of *y* and *x* are used to determine the quadrant of the result. Note that it is intentional and traditional for the two-argument arc tangent function that the argument named *y* be first and the argument named *x* be second. The result is expressed in radians and ranges from -π to +π.

Returns

2 The intrinsic `atan2` function returns a floating-point number. The result is `decimal` of *y* or *x* is `decimal`, otherwise `double`.

Implementation

```
intrinsic function atan2(y: PrimitiveNumber, x: PrimitiveNumber): PrimitiveNumber {
    if (y is decimal && !(x is decimal))
        x = decimal(x);
    else if (x is decimal && !(y is decimal))
        y = decimal(y);

    let Type = (x is double) ? double : decimal;

    if (isNaN(x) || isNaN(y))
        return Type.NaN;
    if (y > 0 && x == 0)
        return Type.PI/2;
    if (isPositiveZero(y))
        return isPositive(x) ? Type(+0) : Type.PI;
    if (isNegativeZero(y))
        return isPositive(x) ? Type(-0) : -Type.PI;
    if (y < 0 && x == 0)
        return -Type.PI/2;
    if (y != 0 && isFinite(y) && !isFinite(x) && x > 0)
        return Type(copysign(0, y));
    if (y != 0 && isFinite(y) && !isFinite(x) && x < 0)
        return copysign(Type.PI, y);
    if (!isFinite(y) && isFinite(x))
        return copysign(Type.PI/2, y);
    if (!isFinite(y) && !isFinite(x))
        return copysign(x > 0 ? Type.PI/4 : 3*Type.PI/4, y);

    if (Type == double)
        return atan2Double(y, x);
    return atan2Decimal(y, x);
}
```

NOTE An implementation is free to produce approximations for all computations involving `PI` in the preceding algorithm.

3 The informative functions `atan2Double` and `atan2Decimal` implement representation-preserving approximate computation of the arc tangent of the quotient of their arguments.

```
informative function atan2Double(y: double, x: double): double …
informative function atan2Decimal(y: decimal, x: decimal): decimal …
```

19.3.6 intrinsic::ceil (x)

Description

1 The intrinsic `ceil` function computes the smallest (closest to -∞) number value that is not less than *x* and is equal to a mathematical integer. If *x* is already an integer, the result is *x*.

NOTE The value of `Math.ceil(`*x*`)` is the same as the value of `-Math.floor(-`*x*`)`.

Returns

2 The intrinsic `ceil` function returns a number in the same representation as *x*.

Implementation

```
intrinsic function ceil(x: PrimitiveNumber): PrimitiveNumber {
    switch type (x) {
    case (n: double) {
        if (!isFinite(n) || n == 0) return n;
        if (-1 < n && n < 0) return -0;
        return ceilDouble(n);
    }
    case (n: decimal) {
        if (!isFinite(n) || n == 0m) return n;
        if (-1m < n && n < 0m) return -0m;
        return ceilDecimal(n);
    }
    }
}
```

3 The informative functions `ceilDouble` and `ceilDecimal` implement representation-preserving computation of the ceiling of their argument.

```
informative function ceilDouble(x: double): double …
informative function ceilDecimal(x: decimal): decimal …
```

19.3.7 intrinsic::cos (x)

Description

1 The intrinsic `cos` method computes an implementation-dependent approximation to the cosine of the number *x*. The argument is expressed in radians.

Returns

2 The intrinsic `cos` function returns a floating-point number.

Implementation

```
intrinsic function cos(x: PrimitiveNumber): PrimitiveNumber {
    switch type (x) {
    case (n: double) {
        if (!isFinite(n)) return NaN;
        if (n == 0) return 1;
        return cosDouble(n);
    }
    case (n: decimal) {
        if (!isFinite(n)) return decimal.NaN;
        if (n == 0m) return 1m;
        return cosDecimal(n);
    }
    }
}
```

3 The informative functions `cosDouble` and `cosDecimal` implement representation-preserving approximate computation of the cosine of their argument.

```
informative function cosDouble(x: double): double …
informative function cosDecimal(x: decimal): decimal …
```

19.3.8 intrinsic::exp (x)

Description

1 The intrinsic `exp` function computes an implementation-dependent approximation to the exponential function of the number *x* (e^x, where *e* is the base of the natural logarithms).

Returns

2 The intrinsic `exp` function returns a floating-point number.

Implementation

```
intrinsic function exp(x: PrimitiveNumber): PrimitiveNumber {
    switch type (x) {
    case (n: double) {
        if (isNaN(n)) return n;
        if (n == 0) return 1d;
        if (n == Infinity) return Infinity;
        if (n == -Infinity) return 0;
        return expDouble(n);
    }
    case (n: decimal) {
        if (isNaN(n)) return n;
        if (n == 0m) return 1m;
        if (n == decimal.POSITIVE_INFINITY) return decimal.POSITIVE_INFINITY;
        if (n == decimal.NEGATIVE_INFINITY) return 0m;
        return expDecimal(n);
    }
    }
}
```

3 The informative functions `expDouble` and `expDecimal` implement representation-preserving approximate computation of the exponential function of their argument.

```
informative function expDouble(x: double): double …
informative function expDecimal(x: decimal): decimal …
```

19.3.9 intrinsic::floor (x)

Description

1 The intrinsic `floor` function computes the greatest (closest to +∞) number value that is not greater than *x* and is equal to a mathematical integer. If *x* is already an integer, the result is *x*.

Returns

2 The intrinsic `floor` function returns a number in the same representation as *x*.

Implementation

```
intrinsic function floor(x: PrimitiveNumber): PrimitiveNumber {
    switch type (x) {
    case (n: double) {
        if (!isFinite(n) || n == 0) return n;
        if (0 < n && n < 1) return +0;
        return floorDouble(n);
    }
    case (n: decimal) {
        if (!isFinite(n) || n == 0m) return n;
        if (0m < n && n < 1m) return +0m;
        return floorDecimal(n);
    }
    }
}
```

NOTE The value of `Math.floor(x)` is the same as the value of `-Math.ceil(-x)`.

3 The informative functions `floorDouble` and `floorDecimal` implement representation-preserving computation of the floor of their argument.

```
informative function floorDouble(x: double): double …
informative function floorDecimal(x: decimal): decimal …
```

19.3.10 intrinsic::log (x)

Description

1 The intrinsic `log` function computes an implementation-dependent approximation to the natural logarithm of the number *x*.

Returns

2 The intrinsic `log` function returns a floating-point number.

Implementation

```
intrinsic function log(x: PrimitiveNumber): PrimitiveNumber {
    switch type (x) {
    case (n: double) {
        if (isNaN(n) || n < 0) return NaN;
        if (n == 0) return -Infinity;
        if (n == 1) return +0;
        if (n == Infinity) return n;
        return logDouble(n);
    }
    case (n: decimal) {
        if (isNaN(n) || n < 0m) return decimal.NaN;
        if (n == 0m) return decimal.NEGATIVE_INFINITY;
        if (n == 1m) return +0m;
        if (n == decimal.POSITIVE_INFINITY) return n;
        return logDecimal(n);
    }
    }
}
```

3 The informative functions `logDouble` and `logDecimal` implement representation-preserving approximate computation of the natural logarithm of their argument.

```
informative function logDouble(x: double): double …
informative function logDecimal(x: decimal): decimal …
```

19.3.11 intrinsic::max (x, y)

Description

1 The intrinsic `max` method selects the numerically largest (closest to +∞) value among *x* and *y*. +0 is considered larger than -0.

Returns

2 The intrinsic `max` method returns either *x* or *y*.

Implementation

```
intrinsic function max(x: PrimitiveNumber, y: PrimitiveNumber): PrimitiveNumber {
    if (isNaN(x)) return x;
    if (isNaN(y)) return y;
    if (x > y) return x;
    if (y > x) return y;
    if (x != 0) return x;

    let x_sign = sign(x),
        y_sign = sign(y);
    if (x_sign > y_sign) return x;
    if (y_sign > x_sign) return y;
    return x;
}
```

NOTE If *x* and *y* are numerically equal (and of the same sign if they are both 0) then the implementation is free to return either one of them.

19.3.12 intrinsic::min (x, y)

Description

1 The intrinsic `min` method selects the numerically smallest (closest to -∞) number among *x* and *y*. -0 is considered smaller than +0.

Returns

2 The intrinsic `min` method returns either *x* or *y*.

Implementation

```
intrinsic function min(x: PrimitiveNumber, y: PrimitiveNumber): PrimitiveNumber {
    if (isNaN(x)) return x;
    if (isNaN(y)) return y;
    if (x < y) return x;
    if (y < x) return y;
    if (x != 0) return x;

    let x_sign = sign(x),
        y_sign = sign(y);
    if (x_sign < y_sign) return x;
    if (y_sign < x_sign) return y;
    return x;
}
```

NOTE If *x* and *y* are numerically equal (and of the same sign if they are both 0) then the implementation is free to return either one of them.

19.3.13 intrinsic::pow (x, y)

Description

1 The intrinsic pow function computes an implementation-dependent approximation to the result of raising *x* to the power *y*.

2 The intrinsic pow function produces a result in the representation of *x*.

Returns

3 The intrinsic pow function returns a number.

Implementation

```
intrinsic function pow(x: PrimitiveNumber, y: PrimitiveNumber): PrimitiveNumber {
    if (x is decimal && !(y is decimal))
        y = decimal(y);
    else if (y is decimal && !(x is decimal))
        x = decimal(x);

    let Type = (x is double) ? double : decimal;

    if (isNaN(y)) return Type.NaN;
    if (y == 0) return Type(1);
    if (isNaN(x) && y != 0) return Type.NaN;
    if (abs(x) > 1 && y == Infinity) return Type.POSITIVE_INFINITY;
    if (abs(x) > 1 && y == -Infinity) return Type(+0);
    if (abs(x) == 1 && y == Infinity) return Type.NaN;
    if (abs(x) == 1 && y == -Infinity) return Type.NaN;
    if (abs(x) < 1 && y == Infinity) return Type(+0);
    if (abs(x) < 1 && y == -Infinity) return Type.POSITIVE_INFINITY;
    if (x == Infinity && y > 0) return Type.POSITIVE_INFINITY;
    if (x == Infinity && y < 0) return Type(+0);
    if (x == -Infinity && y > 0 && isOddInteger(y)) return Type.NEGATIVE_INFINITY;
    if (x == -Infinity && y > 0 && !isOddInteger(y)) return Type.POSITIVE_INFINITY;
    if (x == -Infinity && y < 0 && isOddInteger(y)) return Type(-0);
    if (x == -Infinity && y < 0 && !isOddInteger(y)) return Type(+0);
    if (x == 0 && y > 0) return Type(+0);
    if (x == 0 && y < 0) return Type.POSITIVE_INFINITY;
    if (isNegativeZero(x) && y > 0 && isOddInteger(y)) return Type(-0);
    if (isNegativeZero(x) && y > 0 && !isOddInteger(y)) return Type(+0);
    if (isNegativeZero(x) && y < 0 && isOddInteger(y)) return Type.NEGATIVE_INFINITY;
    if (isNegativeZero(x) && y < 0 && !isOddInteger(y)) return Type.POSITIVE_INFINITY;
    if (x < 0 && isFinite(x) && isFinite(y) && !isIntegral(y)) return Type.NaN;

    if (Type == double)
        return powDouble(x, y);
    return powDecimal(x, y);
}
```

4 The informative functions `powDouble` and `powDecimal` implement representation-dependent computation of the value x^y.

```
informative function powDouble(x: double, y: double): double …
informative function powDecimal(x: decimal, y: decimal): decimal …
```

19.3.14 intrinsic::random ( )

Description

1 The intrinsic `random` function computes a `double` value with positive sign, greater than or equal to 0 but less than 1, chosen randomly or pseudo randomly with approximately uniform distribution over that range, using an implementation-dependent algorithm or strategy. This function takes no arguments.

Returns

2 The intrinsic `random` function returns a `double`.

Implementation

3 The intrinsic `random` function is implementation-dependent.

19.3.15 intrinsic::round (x)

Description

1 The intrinsic `round` function computes the number value that is closest to *x* and is equal to a mathematical integer. If two integer number values are equally close to *x*, then the result is the number value that is closer to +∞. If *x* is already an integer, the result is *x*.

Returns

2 The intrinsic `round` function returns a number, the representation of which is always the same as the representation of the input *x*.

Implementation

```
intrinsic function round(x: PrimitiveNumber): PrimitiveNumber {
    switch type (x) {
    case (n: double) {
        if (!isFinite(n) || n == 0) return n;
        if (0 < n && n < 0.5) return +0;
        if (-0.5 < n && n < 0) return -0;
        return roundDouble(n);
    }
    case (n: decimal) {
        if (!isFinite(n) || n == 0m) return n;
        if (0m < n && n < 0.5m) return +0m;
        if (-0.5m < n && n < 0m) return -0m;
        return roundDecimal(n);
    }
    }
}
```

3 The informative functions `roundDouble` and `roundDecimal` implement representation-preserving computation of the rounded value of their argument.

```
informative function roundDouble(x: double):double …
informative function roundDecimal(x: decimal):decimal …
```

NOTE The intrinsic `round` function preserves the sign of *x* if *x* is 0.

NOTE `Math.round(3.5)` returns 4, but `Math.round(-3.5)` returns -3.

NOTE The value of `Math.round(x)` is the same as the value of `Math.floor(x+0.5)`, except when *x* is -0 or is less than 0 but greater than or equal to -0.5; for these cases `Math.round(x)` returns -0, but `Math.floor(x+0.5)` returns +0.

19.3.16 intrinsic::sin (x)

Description

1 The intrinsic `sin` function computes an implementation-dependent approximation to the sine of the number *x*. The argument is expressed in radians.

Returns

2 The intrinsic `sin` function returns a floating-point number.

Implementation

```
intrinsic function sin(x: PrimitiveNumber): PrimitiveNumber {
    switch type (x) {
    case (n: double) {
        if (!isFinite(n)) return NaN;
        if (n == 0) return n;
        return sinDouble(n);
    }
    case (n: decimal) {
        if (!isFinite(n)) return decimal.NaN;
        if (n == 0m) return n;
        return sinDecimal(n);
    }
    }
}
```

3 The informative functions `sinDouble` and `sinDecimal` implement representation-preserving approximate computation of the sine of their argument.

```
informative function sinDouble(x: double):double …
informative function sinDecimal(x: decimal):decimal …
```

NOTE The intrinsic `sin` function preserves the sign of *x* if *x* is 0.

19.3.17 intrinsic::sqrt (x)

Description

1 The intrinsic `sqrt` method computes an implementation-dependent approximation to the square root of the number *x*.

Returns

2 The intrinsic `sqrt` method returns a number. The representation of the result is the same as the representation of *x*.

Implementation

```
intrinsic function sqrt(x: PrimitiveNumber): PrimitiveNumber {
    switch type (x) {
    case (n: double) {
        if (isNaN(n) || n < 0) return NaN;
        if (n == 0 || n == Infinity) return n;
        return sqrtDouble(n);
    }
    case (n: decimal) {
        if (isNaN(n) || n < 0m) return decimal.NaN;
        if (n == 0m || n == decimal.POSITIVE_INFINITY) return n;
        return sqrtDecimal(n);
    }
    }
}
```

19.3.18 intrinsic::tan (x)

Description

1 The intrinsic `tan` function computes an implementation-dependent approximation to the tangent of *x*. The argument is expressed in radians.

Returns

2 The intrinsic `tan` function returns a floating-point number.

Implementation

```
intrinsic function sin(x: PrimitiveNumber): PrimitiveNumber {
    switch type (x) {
    case (n: double) {
        if (!isFinite(n)) return NaN;
        if (n == 0) return n;
```

```
            return sinDouble(n);
        }
        case (n: decimal) {
            if (!isFinite(n)) return decimal.NaN;
            if (n == 0m) return n;
            return sinDecimal(n);
        }
        }
    }
```

3 The informative functions `tanDouble` and `tanDecimal` implement representation-preserving approximate computation of the tangent of their argument.

```
informative function tanDouble(x: double):double …
informative function tanDecimal(x: decimal):decimal …
```

NOTE The intrinsic `tan` function preserves the sign of *x* if *x* is 0.

19.4 Other function properties of the Math object

1 Every function listed in this section applies the `toPrimitiveNumber` function to each of its arguments (in left-to-right order if there is more than one) and then performs a computation on the resulting number value(s) by invoking the corresponding intrinsic method.

```
Math.public::abs =
    function (x) Math.abs(toPrimitiveNumber(x));

Math.public::acos =
    function (x) Math.acos(toPrimitiveNumber(x));

Math.public::asin =
    function (x) Math.asin(toPrimitiveNumber(x));

Math.public::atan =
    function (x) Math.atan(toPrimitiveNumber(x));

Math.public::atan2 =
    function (y,x)
        Math.atan2(toPrimitiveNumber(y), toPrimitiveNumber(x));

Math.public::ceil =
    function (x) Math.ceil(toPrimitiveNumber(x));

Math.public::cos =
    function (x) Math.cos(toPrimitiveNumber(x));

Math.public::exp =
    function (x) Math.exp(toPrimitiveNumber(x));

Math.public::floor =
    function (x) Math.floor(toPrimitiveNumber(x));

Math.public::log =
    function (x) Math.log(toPrimitiveNumber(x));

Math.public::pow =
    function (x, y)
        Math.pow(toPrimitiveNumber(x), toPrimitiveNumber(y));

Math.public::random =
    function () Math.random();

Math.public::round =
    function (x) Math.round(toPrimitiveNumber(x));

Math.public::sin =
    function (x) Math.sin(toPrimitiveNumber(x));

Math.public::sqrt =
    function (x) Math.sqrt(toPrimitiveNumber(x));

Math.public::tan =
    function (x) Math.atan(toPrimitiveNumber(x));
```

2 The `max` and `min` functions are more general than their corresponding intrinsic methods: they accept zero or more arguments and apply their corresponding intrinsic methods to the current result and the next argument, in left-to-right order.

```
Math.public::max =
    function max(...xs) {
        if (xs.length == 0)
            return -Infinity;
        let result = toPrimitiveNumber(xs[0]);
        for ( let i=1 ; i < xs.length; ++i ) {
            result = Math.max(result, toPrimitiveNumber(xs[i]));
            if (isNaN(result))
                break;
        }
        return result;
    };

Math.public::min =
    function min(...xs) {
        if (xs.length == 0)
            return Infinity;
        let result = toPrimitiveNumber(xs[0]);
        for ( let i=1 ; i < xs.length; ++i ) {
            result = Math.min(result, toPrimitiveNumber(xs[i]));
            if (isNaN(result))
                break;
        }
        return result;
    };
```

20 The class `Date`

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.

20.1 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) …
    static meta 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(str, reference:double=0.0) …
    static var UTC = …

    static const length: uint = 7;

    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) … ,
```

20.2 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.

20.2.1 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.

20.2.2 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];
```

20.2.3 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;
helper function HourFromTime(t : double) : double {

    let v = Math.floor(t / msPerHour) % hoursPerDay;
    if (v < 0)
        return v + hoursPerDay;
    return v;
}
```

20.2.4 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;
    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 (Ticket #190.) 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;
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;
    }
}
```

20.2.5 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.

20.2.6 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);
}
```

20.2.7 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.

20.2.8 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 (Ticket #129.) 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".

20.2.9 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 (Ticket #129.) 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`.

20.2.10 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())`.

20.2.11 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;
helper function DaysInYear(y : double) : double {

    if (y % 4 !== 0 || y % 100 === 0 && y % 400 !== 0)
        return 365;
    return 366;
}
```

20.2.12 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));
}
```

20.2.13 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);

    return FindDay(year, month) + date - 1;
}
```

20.2.14 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;
}
```

20.2.15 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.

20.3 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 =
    /^                                              \
     (?: (?P<year> - [0-9]+ | [0-9]{4} [0-9]* )    \
      (?: - (?P<month> [0-9]{2} )                  \
       (?: - (?P<day> [0-9]{2} ) )? )? )?          \
     T                                              \
     (?: (?P<hour> [0-9]{2} )                       \
      (?: : (?P<minutes> [0-9]{2} )                 \
       (?: : (?P<seconds> [0-9]{2} )                \
        (?: \. (?P<fraction> [0-9]+ ) )? )? )? )?   \
     (?: (?P<zulu> Z )                              \
       | (?P<offs>                                  \
          (?P<tzdir> \+ | - )                       \
          (?P<tzhr> [0-9]{2} )                      \
          (?: : (?P<tzmin> [0-9]{2} ) )? ) )?       \
    $/x;
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));
}
```

FIXME (Ticket #192.) Replace the regexp by a proper grammar, eventually.

FIXME The `T` is optional if the time stamp does not have a time part.

20.4 Methods on the Date class

20.4.1 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) {
    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 : double = 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**.

20.4.2 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

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

20.4.3 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 {
    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.UTCHours;
        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 fromDateString(s, reference);

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

20.4.4 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(str, reference:double=0.0) {
    return Date.parse(string(str), reference);
};
```

20.4.5 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)));
}
```

20.4.6 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**.

20.4.7 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.

20.5 Methods on Date instances

20.5.1 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.

20.5.2 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.

20.5.3 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.

20.5.4 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.

20.5.5 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.

20.5.6 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.

20.5.7 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.

20.5.8 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(UTCMilliseconds) +
            "Z");
}

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

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

20.5.9 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.

20.5.10 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();
```

20.5.11 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;
```

20.5.12 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));
```

20.5.13 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);
```

20.5.14 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));
```

20.5.15 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);
```

20.5.16 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));
```

20.5.17 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);
```

20.5.18 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));
```

20.5.19 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);
```

20.5.20 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));
```

20.5.21 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);
```

20.5.22 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));
```

20.5.23 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);
```

20.5.24 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));
```

20.5.25 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);
```

20.5.26 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));
```

20.5.27 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);
```

20.5.28 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;
```

20.5.29 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);
```

20.5.30 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)));
```

20.5.31 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));
```

20.5.32 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 (Ticket #193.) 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.

20.5.33 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));
```

20.5.34 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)));
```

20.5.35 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));
```

20.5.36 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)));
```

20.5.37 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));
```

20.5.38 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)));
```

20.5.39 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));
```

20.5.40 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)));
```

20.5.41 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));
```

20.5.42 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)));
```

20.5.43 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));
```

20.6 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();
```

20.7 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);
```

20.8 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.intrinsic::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 = this.getMilliseconds())
    this.intrinsic::setSeconds(double(sec), double(ms));

prototype function setUTCSeconds(this:Date, sec, ms = this.getUTCMilliseconds())
    this.intrinsic::setUTCSeconds(double(sec), double(ms));

prototype function setMinutes(this:Date, min, sec = this.getSeconds(), ms =
this.getMilliseconds())
    this.intrinsic::setMinutes(double(min), double(sec), double(ms));

prototype function setUTCMinutes(this:Date,
                                 min,
                                 sec = this.getUTCSeconds(),
                                 ms = this.getUTCMilliseconds())
    this.intrinsic::setUTCMinutes(double(min), double(sec), double(ms));

prototype function setHours(this:Date,
                            hour,
                            min=this.getMinutes(),
                            sec=this.getSeconds(),
                            ms=this.getMilliseconds())
    this.intrinsic::setHours(double(hour), double(min), double(sec), double(ms));

prototype function setUTCHours(this:Date,
                               hour,
                               min=this.getUTCMinutes(),
                               sec=this.getUTCSeconds(),
                               ms=this.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=this.getDate())
    this.intrinsic::setMonth(double(month), double(date));

prototype function setUTCMonth(this:Date, month, date=this.getUTCDate())
    this.intrinsic::setUTCMonth(double(month), double(date));

prototype function setFullYear(this:Date, year, month=this.getMonth(), date=this.getDate
())
    this.intrinsic::setFullYear(double(year), double(month), double(date));

prototype function setUTCFullYear(this:Date, year, month=this.getUTCMonth(),
date=this.getUTCDate())
    this.intrinsic::setUTCFullYear(double(year), double(month), double(date));
```

21 The class `RegExp`

```
FILE:                       spec/library/RegExp.html
DRAFT STATUS:               DRAFT 1 - ROUGH - 2008-06-25
```

```
REVIEWED AGAINST ES3:       NO
REVIEWED AGAINST ERRATA:    NO
REVIEWED AGAINST BASE DOC:  NO
REVIEWED AGAINST PROPOSALS: NO
REVIEWED AGAINST CODE:      NO
```

1 The class `RegExp` is a dynamic, nullable, non-final, direct subclass of `Object`.

2 A RegExp object contains a regular expression *pattern* and the associated *flags*.

NOTE The form and functionality of regular expressions is modelled after the regular expression facility in the Perl 5 programming language.

3 A regular expression is transformed ("compiled") into a *matcher* function that can be used to *match* an *input string*: to test whether the input string has a certain form or contains substrings of a certain form, where the form is defined by the regular expression.

4 The intrinsic `exec` method on a `RegExp` object drives the matching by invoking the matcher on a string and an offset within the string to determine whether the pattern would match starting at exactly that offset within the string, and, if it does match, what the values of the capturing parentheses would be.

5 Regular expression patterns are written down using a compact and rich source syntax that is separate from the syntax of the surrounding language. A grammar for this syntax is presented below (RegExp grammar).

6 The regular expression flags modify the meaning of the pattern in various ways, for example by specifying case-insensitivity, the meaning of white space, or how to perform the matching.

7 This Standard defines the meaning of regular expressions in two stages: declaratively as a mapping from surface syntax to abstract syntax trees, and then operationally (in ECMAScript itself) as an interpreter that performs matching of input strings by interpreting those abstract syntax trees.

8 Compiling a pattern may throw a `SyntaxError` exception; on the other hand, once the pattern is successfully compiled, applying the compiled pattern to find a match in a string cannot throw an exception (except for any host-defined exceptions that can occur anywhere such as out-of-memory).

9 The abstract syntax trees for regular expressions are represented as trees of ECMAScript objects. These objects are all instances of specific ECMAScript classes, which are presented below (RegExp ASTs).

21.1 Synopsis

1 The class `RegExp` provides the following interface:

```
dynamic class RegExp
{
    function RegExp( pattern, flags ) …
    static meta function invoke( pattern, flags ) …

    static const length: uint = 2

    override intrinsic function toString() : string …

    intrinsic function exec(s : string) : Array …
    intrinsic function test(s : string) : boolean …

    meta function invoke(s : string) : Array …

    const source: string = …
    const global: boolean = …
    const ignoreCase: boolean = …
    const multiline: boolean = …
    const extended: boolean = …
    const sticky: boolean = …

    final function get lastIndex() …
    final function set lastIndex(x) …
}
```

2 The `RegExp` prototype object provides the following direct dynamic properties:

```
    exec:     function (s) … ,
    test:     function (s) … ,
    toString: function () …
```

21.2 Surface syntax and mapping to abstract syntax trees

21.2.1 Grammar

1 The `RegExp` constructor applies the following grammar to the pattern string. A **SyntaxError** exception is thrown if the grammar cannot recognize the string as an expansion of the nonterminal *Pattern*.

2 The grammar acts as a transformer from segments of the pattern string into abstract syntax (sub)trees. A transformation computation of one of the following types is associated with each production in the grammar.

- The construction of an abstract syntax tree node. Construction appears as the call to a factory function for the node, denoted by an identifier in boldface with an initial capital letter. The factory functions map directly to the constructors for the respective abstract syntax tree classes, described in section RegExp.matching.
- The computation of a value, which may be a string, a number, or an abstract syntax tree node. Computation appears as the call to a helper function, denoted by an identifier in boldface with an initial lower-case letter.
- The construction or destructuring of value tuples that carry multiple values from a production to a surrounding production. Tuple construction and destructuring use the ECMAScript syntax for constructing and destructuring Array values.
- The extraction of a token value from the production. Extraction appears as the use of the name of the token in the computation.

3 The definition language also has side computations, side conditions, and error conditions. These are suffixed to a transformation computation.

- Side computations bind temporary names to values and are introduced by `where` clauses.
- Side conditions test the applicability of a production and are introduced by `provided` clauses.
- Error conditions throw a **SyntaxError** exception if they do not hold and are introduced by `requires` clauses.

```
Pattern ::
    Disjunction                  => RegExpMatcher( Disjunction )

Disjunction ::
    Alternative                  => Alternative
    Alternative | Disjunction    => Disjunct( Alternative, Disjunction )

Alternative ::
    [empty]                      => Empty()
    Alternative Term             => Conjunct( Alternative, Term )

Term ::
    Assertion                    => Assertion
    Atom                         => Atom
    Atom Quantifier              => Quantified(parenIndex,
                                               parenCount,
                                               atom,
                                               min,
                                               max,
                                               greedy)
                                    where [min, max, greedy] = Quantifier
                                    requires min ≤ max

Assertion ::
    ^                            => AssertStartOfInput()
    $                            => AssertEndOfInput()
    \ b                          => AssertWordBoundary()
    \ B                          => AssertNotWordBounary()

Quantifier ::
    QuantifierPrefix             => [min, max, true] where [min, max] = QuantifierPrefix
    QuantifierPrefix ?           => [min, max, false] where [min, max] = QuantifierPrefix

QuantifierPrefix ::
    *                            => [0, ∞]
    +                            => [1, ∞]
    ?                            => [0, 1]
    { DecimalDigits }            => [DecimalDigits, DecimalDigits]
    { DecimalDigits , }          => [DecimalDigits, ∞]
    { DecimalDigits1 , DecimalDigits2 }
                                 => [DecimalDigits1, DecimalDigits2]
```

```
Atom ::
    PatternCharacter             => CharacterSet( CharsetAdhoc( PatternCharacter ) )
    .                            => CharacterSet( CharsetComplement( charset_linebreak ) )
    \ DecimalEscape              => Backref( DecimalEscape )
                                    requires that the regular expression as a whole has
                                             at least DecimalEscape capturing parentheses
    \ CharacterEscape            => CharacterSet( CharsetAdhoc( CharacterEscape ) ) )
    \ CharacterClassEscape       => CharacterSet( CharacterClassEscape )
    CharacterClass               => CharacterSet( CharacterClass )
    ( Disjunction )              => Capturing( Disjunction, parenIndex+1 )
    ( ? : Disjunction )          => Disjunction
    ( ? = Disjunction )          => PositiveLookahead( Disjunction )
    ( ? ! Disjunction )          => NegativeLookahead( Disjunction )
    ( ? # [sequence matching [^)]*] ) => Empty()
    ( ? P < Identifier > Disjunction )
                                 => Capturing( Disjunction, capno( Identifier ) )
                                    where capno( Identifier ) is defined as parenIndex+1
    ( ? P = Identifier )         => Backref( capno( Identifier ) )

PatternCharacter ::
    SourceCharacter but not any of ^ $ \ . * + ? ( ) [ ] { } |
                                 => SourceCharacter

CharacterEscape ::
    ControlEscape                => ControlEscape
    c ControlLetter              => chr(ord( ControlLetter ) / 32)
    HexEscapeSequence            => HexEscapeSequence
    UnicodeEscapeSequence        => UnicodeEscapeSequence
    IdentityEscape               => IdentityEscape

ControlEscape ::
    f                            => '\u000C'
    n                            => '\u000A'
    r                            => '\u000D'
    t                            => '\u0009'
    v                            => '\u000B'

ControlLetter :: 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
                                 => ControlLetter

IdentityEscape ::
    SourceCharacter but not IdentifierPart
                                 => SourceCharacter

DecimalEscape ::
    DecimalIntegerLiteral [lookahead not in DecimalDigit]
                                 => dec( DecimalIntegerLiteral )

CharacterClassEscape ::
    d                          => charset_digit
    D                          => CharsetComplement( charset_digit )
    s                          => charset_space
    S                          => CharsetComplement( charset_space )
    w                          => charset_word
    W                          => CharsetComplement( charset_word )
    p { UnicodeClass }         => unicodeClass( UnicodeClass )
    P { UnicodeClass }         => CharsetComplement( unicodeClass( UnicodeClass ) )

CharacterClass ::
    [ CharacterClassBody ]     => merge( U, I )
                                  where [U, I] = CharacterClassBody

CharacterClassBody ::
    [lookahead not in {^}] ClassRanges
                               => ClassRanges
    ^ ClassRanges              => [[ CharsetComplement( merge( U, I ) ) ], []]
                                  where [U, I] = ClassRanges

ClassRanges ::
    [empty]                    => [[ CharsetEmpty() ], []]
    NonemptyClassranges        => NonemptyClassRanges
```

```
NonemptyClassRanges ::
    ClassRange                    => ClassRange
    ClassRange NonemptyClassRanges
                                  => [[union_1, union_2…], [intersection_1, intersection_2…]]
                                     where [union_1, intersection_1] = ClassRange
                                       and [union_2…, intersection_2…] = NonemptyClassRanges

ClassRange ::
    ClassAtom                     => [[ClassAtom],[]]
                                     provided the next production does not apply
    ClassAtom_1 - ClassAtom_2      => [[CharsetRange(ClassAtom_1, ClassAtom_2)],[]]
                                      requires that ClassAtom_1 and ClassAtom_2
                                               have one element each and that
                                               the code point value of ClassAtom_1
                                               is ≤ the code point value of ClassAtom_2
    & & [ CharacterClassBody ]    => [[], [ merge( U, I ) ]]
                                      where [U, I] = CharacterClassBody

ClassAtom ::
    -                              => CharsetAdhoc('-')
    \ DecimalEscape                => CharsetAdhoc( chr( DecimalEscape ) )
    \ b                            => CharsetAdhoc( '\u0008' )
    \ CharacterEscape              => CharsetAdhoc( CharacterEscape )
    \ CharacterClassEscape         => CharacterClassEscape
    SourceCharacter but not one of \ ] -
                                   => CharsetAdhoc( SourceCharacter )

UnicodeClass ::
    Identifier                     => Identifier
                                      provided Identifier spells one of these names:
                                      C Cc Cf Cn Co Cs L Ll Lm Lo Lt Lu M Mc Me Mn N Nd Nl
No
                                      P Pc Pd Pe Pf Pi Po Ps S Sc Sk Sm So Z Zl Zp Zs
```

4 *SourceCharacter*, *HexEscapeSequence*, *UnicodeEscapeSequence*, and *IdentityEscape* are defined as part of the general ECMAScript grammar (language.grammar) and all produce one-character strings.

5 To every expansion of a production there belongs two variables, *parenIndex* and *parenCount*. *ParenIndex* represents the number of left capturing parentheses in the entire regular expression that occur to the left of the production expansion's initial token. *ParenCount* represents the number of left capturing parentheses in the expansion of the production.

6 Character set unions and intersections are represented explicitly as data structures that reference the subsets that are the operands of the union or intersection operator, respectively; sets are not flattened.

7 There are four predefined character sets:

- *charset_linebreak* contains the Unicode line terminator characters <LF>, <CR>, <LS>, and <PS>.
- *charset_digit* contains the decimal digit characters `0` through `9`
- *charset_space* contains all the Unicode *WhiteSpace* (Unicode.whitespace) and *LineTerminator* (Unicode.lineterminator) characters
- *charset_word* contains the upper-case letters `A` through `Z`, the lower-case letters `a` through `z`, the decimal digit characters `0` through `9`, and the underscore `_`.

8 The helper function `chr` converts a Unicode code point value into the corresponding Unicode character (a one-character string).

9 The helper function `ord` converts a one-character string into a Unicode code point value.

10 The helper function `dec` converts the textual representation of a nonnegative decimal integer into its integer value.

11 The helper function `merge` creates a single character set from two collections of sets *U* and *I*, where the result set is the union of the sets of *U*, intersected with the intersection of the sets of *I* unless *I* is empty.

12 The helper function `capno` maps identifiers to capture numbers.

13 The helper function `unicodeClass` maps a one or two character Unicode class name to a character set containing the characters in that Unicode class.

14 Even if the pattern ignores case, the case of the two ends of a range is significant in determining which characters belong to the range. Thus, for example, the pattern `/[E-F]/i` matches only the letters `E`, `F`, `e`, and `f`, while the pattern `/[E-f]/i` matches all upper and lower-case ASCII letters as well as the symbols `[`, `\`, `]`, `^`, `_`, and `` ` ``.

15 A *ClassAtom* can use any of the escape sequences that are allowed in the rest of the regular expression except for `\b`, `\B`, and backreferences. Inside a CharacterClass, `\b` means the backspace character, while `\B` and backreferences raise errors. Using a backreference inside a *ClassAtom* causes an error.

16 *ClassRanges* can expand into single *ClassAtoms* and/or ranges of two *ClassAtoms* separated by dashes. In the latter case the *ClassRanges* includes all characters between the first ClassAtom and the second *ClassAtom*, inclusive; an error occurs if either *ClassAtom* does not represent a single character (for example, if one is `\w`) or if the first *ClassAtom*'s code point value is greater than the second *ClassAtom*'s code point value.

17 A `-` character can be treated literally or it can denote a range. It is treated literally if it is the first or last character of *ClassRanges*, the beginning or end limit of a range specification, or immediately follows a range specification.

21.2.2 White space and line comments

FIXME It has been agreed that comment syntax (both forms) is to be removed from the spec and that newlines in regex literals must be escaped by a backslash. Both decisions make it possible to retain the ES3 split between surface lexing and regex parsing, and probably also helps preserve programmers' sanity.

1 The grammar takes on one of two meanings depending on whether the `x` flag was supplied to the regular expression constructor.

2 If the `x` flag was not supplied then all white space is treated as literal characters (typically *SourceCharacter*) and the `#` character, outside the context of the `(?#` character sequence, does not mean anything special -- it is just another *SourceCharacter*.

3 If the `x` flag was supplied then white space is ignored in a number of contexts and the `#` character, outside the context of the `(?#` character sequence, starts a comment that ends when a line terminator character is seen (the line terminator is not part of the comment). White space and line comments act as token separators but are otherwise completely ignored. The multi-character tokens of the regular expression grammar inside which white space and line comments break the token are:

- the character sequences `(?:`, `(?=`, `(?!`, `(?#`, `(?P=`, `(?P<`, and `&&[`
- the character sequences starting with a backslash (`\`), except that white space and line comments are allowed immediately following the backslash
- *DecimalDigits* and *Identifier*

FIXME (Ticket #194.) The definition on where whitespace is ignored / not ignored needs to be stronger, particularly around backspace. Eg, `\p{N}` is the same as `\p{ N }` but not the same as `\p {N}`. We could go fascist and say that the middle of those three is not allowed, but that seems unnatural.

21.3 Abstract syntax trees

1 The abstract syntax trees for regular expressions are represented as trees of instances of the ECMAScript classes `Disjunct`, `Conjunct`, `AssertStartOfInput`, `AssertEndOfInput`, `AssertWordBoundary`, `AssertNotWordBoundary`, `Quantified`, `Capturing`, `Backref`, `PositiveLookahead`, `NegativeLookahead`, `CharacterSet`, and `Empty`.

2 These abstract syntax tree classes all implement the `Matcher` interface, which requires them to provide a `match` method that takes a matching context, a matching state, and a continuation, and returns a result:

```
interface Matcher {
    function match(ctx: Context, x: State, c: Continuation): MatchResult
}
```

NOTE The types `Context`, `State`, `Continuation`, and `MatchResult` are described later, along with the matching algorithm.

3 An additional set of classes, comprised of `CharsetEmpty`, `CharsetUnion`, `CharsetIntersection`, `CharsetComplement`, `CharsetRange`, and `CharsetAdhoc`, represents character sets and unions, intersections, and complements of characters sets. Each of these implements the `CharsetMatcher` interface, which takes a matching context and a single-character string and returns a truth value:

```
interface CharsetMatcher {
    function match(ctx: Context, s: string): boolean;
}
```

4 All abstract syntax tree classes are described in detail below.

21.4 Matching

1 The `match` method of the object that implements the `Matcher` interface attempts to match a middle subpattern (determined by the type of matcher and its internal settings) of the pattern against the input string, starting at the intermediate state given by its `State` argument. The `Continuation` argument is a closure that matches the rest of the pattern. After matching the subpattern of a pattern to obtain a new `State`, the matcher then calls `Continuation` on the new state to test

if the rest of the pattern can match as well. If it can, the matcher returns the state returned by the continuation; if not, the matcher may try different choices at its choice points, repeatedly calling the continuation until it either succeeds or all possibilities have been exhausted.

2 The interface to this machinery is the class `RegExpMatcher`, which takes as arguments an input, a start position, and some flags, and which constructs a matcher context, an internal state, and a final continuation and then invokes its internal matcher on these values, returning the result returned by the matcher.

3 A `Context` object describes constant values used by all the matchers during the matching.

```
class Context
{
    const input       : string,
          inputLength : uint,
          ignoreCase  : boolean,
          multiline   : boolean
}
```

4 `input` is the string being matched by the regular expression pattern; `inputLength` is the number of characters in `input`; `ignoreCase` is **true** if the matching is case-insensitive; and `multiline` is **true** if the matching allows the `^` assertion to match at the beginning of a line and the `$` assertion to match at the end of a line (and not just at the beginning and end of the input, respectively).

5 A `State` object represents partial match states in the regular expression matching algorithms.

```
class State
{
   const endIndex: uint,
         captures: CapArray
}

type CapArray = [(string,undefined)]
```

6 The `endIndex` is one plus the index of the last input character matched so far by the pattern, while `captures` holds the results of capturing parentheses. `Captures` is an array whose length is the number of left capturing parentheses in the *pattern*. The *n*th element of `captures` is either a string that represents the value obtained by the *n*th set of capturing parentheses or **undefined** if the *n*th set of capturing parentheses hasn't been reached yet. Due to backtracking, many states may be in use at any time during the matching process.

7 A `MatchResult` is either a `State` or the special token **failure** that indicates that the match failed:

```
type MatchResult = (State, …)
const failure = …
```

8 A Continuation function is a closure that takes a `Context` and a `State` and returns a `MatchResult`:

```
type Continuation = function(Context, State): MatchResult;
```

9 The continuation attempts to match the remaining portion (specified by the closure's already-bound arguments) of the pattern against the input string, starting at the intermediate state given by its `State` argument. If the match succeeds, the continuation returns the final `State` that it reached; if the match fails, the continuation returns **failure**.

21.4.1 RegExpMatcher

1 The class `RegExpMatcher` drives the matching. When its `match` method is invoked it creates a `Context`, a `State`, a `Continuation`, and then it invokes its matcher object---the result of compiling the *pattern*---on these values, returning the result returned by the matcher.

```
function RegExpMatcher(matcher, nCapturingParens)
    : matcher = matcher
    , nCapturingParens = nCapturingParens
{
}

const matcher:          Matcher,
      nCapturingParens: uint

function match( input: string, endIndex: double, multiline: boolean, ignoreCase: boolean )
    : MatchResult
    {
        return matcher.match(new Context(input, multiline, ignoreCase),
                             new State(endIndex, makeCapArray(nCapturingParens+1)),
                             function (ctx: Context, x: State): State? { return x } );
    }
```

2 Capture arrays are created by `makeCapArray` and copied and partly cleared by `copyCapArray`:

```
function makeCapArray(len: double): CapArray {
    let a = []: CapArray;
    for ( let i = 0 ; i < len ; i++ )
        a[i] = undefined;
    return a;
}

function copyCapArray(a: CapArray, parenIndex: double, parenCount: double): CapArray {
    let b = makeCapArray(a.length);
    for ( let i = 0 ; i < a.length ; i++ )
        b[i] = a[i];

    for ( let k = parenIndex+1 ; k <= parenIndex+parenCount ; k++ )
        b[k] = undefined;
    return b;
}
```

21.4.2 Disjunct

1 The class `Disjunct` represents a matcher that allows two alternatives:

```
function Disjunct(m1, m2) : m1=m1, m2=m2 {}

const m1: Matcher,
      m2: Matcher

public function match(ctx: Context, x: State, c: Continuation): MatchResult {
    let r = m1.match(ctx, x, c);
    if (r != failure)
        return r;
    return m2.match(ctx, x, c);
}
```

2 A `Disjunct` first tries to match the left alternative *m1* (followed by the sequel of the regular expression); if it fails, it tries to match the right alternative *m2* (followed by the sequel of the regular expression). If *m1*, *m2*, and the sequel all have choice points, all choices in the sequel are tried before moving on to the next choice in *m1*. If choices in *m1* are exhausted, *m2* is tried instead of *m1*. Any capturing parentheses inside a portion of the pattern skipped by `|` produce undefined values instead of strings. Thus, for example,

```
/a|ab/.exec("abc")
```

returns the result "a" and not "ab". Moreover,

```
/((a)|(ab))((c)|(bc))/.exec("abc")
```

returns the array

```
["abc", "a", "a", undefined, "bc", undefined, "bc"]
```

and not

```
["abc", "ab", undefined, "ab", "c", "c", undefined]
```

21.4.3 Conjunct

1 The class `Conjunct` represents a matcher that requires the matching of two consecutive terms:

```
function Conjunct(m1, m2) : m1=m1, m2=m2 {}

const m1: Matcher,
      m2: Matcher

public function match(ctx: Context, x: State, c: Continuation): MatchResult {
    return m1.match(ctx,
                    x,
                    (function (ctx: Context, y: State): MatchResult
                     m2.match(ctx, y, c)) );
}
```

2 A `Conjunct` tries simultaneoulsly to match the terms *m1* and *m2* on consecutive portions of the input string. If *m1*, *m2*, and the sequel of the regular expression all have choice points, all choices in the sequel are tried before moving on to the next choice in *m2*, and all choices in *m2* are tried before moving on to the next choice in *m1*.

21.4.3.1 AssertStartOfInput

1 The Start-of-Input assertion succeeds without consuming input if the current input position is at the start of the input or if the match is multiline and the current position is at the start of a line.

```
public function match(ctx: Context, x: State, c: Continuation): MatchResult {
    let e = x.endIndex;
    if (e == 0 || ctx.multiline && isTerminator(ctx.input[e-1]))
        return c(ctx, x);
    return failure;
}
```

2 The helper function `isTerminator` takes a character *c* and returns **true** if *c* is one of the Unicode line terminator characters <LF>, <CR>, <LS>, and <PS>.

21.4.3.2 AssertEndOfInput

1 The End-of-Input assertion succeeds without consuming input if the current input position is at the end of the input or if the match is multiline and the current position is at the end of a line.

```
public function match(ctx: Context, x: State, c: Continuation): MatchResult {
    let e = x.endIndex;
    if (e == ctx.inputLength || ctx.multiline && isTerminator(ctx.input[e]))
        return c(ctx, x);
    return failure;
}
```

21.4.3.3 AssertWordBoundary

1 The Word-Boundary assertion succeeds without consuming input if the current input position is inside a word and the previous position is outside a word, or vice versa.

```
public function match(ctx: Context, x: State, c: Continuation): MatchResult {
    let e = x.endIndex;
    if (isREWordChar(ctx, e-1) != isREWordChar(ctx, e))
        return c(ctx, x);
    return failure;
}
```

2 The test for word character also takes boundary conditions into consideration:

```
function isREWordChar(ctx: Context, e: double): boolean {
    if (e == -1 || e == ctx.inputLength)
        return false;
    let c = ctx.input[e];
    return isWordChar(ctx.input[e]);
}
```

3 The helper function `isWordChar` takes a character *c* and returns **true** if *c* is one of the upper-case ASCII letters `A` through `Z`, one of the lower-case ASCII letters `a` through `z`, one of the ASCII decimal digits `0` through `9`, or the ASCII underbar `_`.

21.4.3.4 AssertNotWordBoundary

1 The Not-Word-Boundary assertion succeeds without consuming input if neither the current input position is inside a word and the previous position is outside a word, nor vice versa.

```
public function match(ctx: Context, x: State, c: Continuation): MatchResult {
    let e = x.endIndex;
    if (isREWordChar(ctx, e-1) == isREWordChar(ctx, e))
        return c(ctx, x);
    return failure;
}
```

21.4.4 Quantified

1 The class `Quantified` represents a matcher that succeeds if its submatch *m* matches the input a certain number of times.

```
function Quantified(parenIndex, parenCount, m, min, max, greedy)
    : parenIndex=parenIndex
    , parenCount=parenCount
    , m=m
    , min=min
    , max=max
    , greedy=greedy
```

```
const parenIndex: uint,
      parenCount: uint,
      m: Matcher,
      min: double,
      max: double,
      greedy: boolean

public function match(ctx: Context, x: State, c: Continuation): MatchResult {

    function RepeatMatcher(min: double, max: double, x: State): MatchResult {
        function d(ctx: Context, y: State): MatchResult {
            if (min == 0 && y.endIndex == x.endIndex)
                return failure;
            else
                return RepeatMatcher(Math.max(0, min-1), max-1, y);
        }

        if (max == 0)
            return c(ctx, x);

        let xr = new State(x.endIndex, copyCapArray(x.captures, parenIndex, parenCount));

        if (min != 0)
            return m.match(ctx, xr, d);

        if (!greedy) {
            let z = c(ctx, x);
            if (z != failure)
                return z;
            return m.match(ctx, xr, d);
        }
        else {
            let z = m.match(ctx, xr, d);
            if (z != failure)
                return z;
            return c(ctx, x);
        }
    }

    return RepeatMatcher(min, max, x);
}
```

2 A pattern term *m* followed by a quantifier is repeated the number of times specified by the quantifier. A quantifier can be non-greedy, in which case *m* is repeated as few times as possible while still matching the sequel, or it can be greedy, in which case *m* is repeated as many times as possible while still matching the sequel. *M* is repeated rather than the input string that it matches, so different repetitions of *m* can match different input substrings.

3 If *m* and the sequel of the regular expression all have choice points, *m* is first matched as many (or as few, if non-greedy) times as possible. All choices in the sequel are tried before moving on to the next choice in the last repetition of *m.* All choices in the last (nth) repetition of *m* are tried before moving on to the next choice in the next-to-last (n-1)st repetition of *m*; at which point it may turn out that more or fewer repetitions of *m* are now possible; these are exhausted (again, starting with either as few or as many as possible) before moving on to the next choice in the (n-1)st repetition of Atom and so on.

4 Compare

```
/a[a-z]{2,4}/.exec("abcdefghi")
```

which returns `"abcde"` with

```
/a[a-z]{2,4}?/.exec("abcdefghi")
```

which returns `"abc"`.

5 Consider also

```
/(aa|aabaac|ba|b|c)*/.exec("aabaac")
```

which, by the choice point ordering above, returns the array

```
["aaba", "ba"]
```

and not any of:

```
["aabaac", "aabaac"]
["aabaac", "c"]
```

6 The above ordering of choice points can be used to write a regular expression that calculates the greatest common divisor of two numbers (represented in unary notation). The following example calculates the *gcd* of 10 and 15:

```
    "aaaaaaaaaa,aaaaaaaaaaaaaaa".replace(/^(a+)\1*,\1+$/,"$1")
```

which returns the *gcd* in unary notation `"aaaaa"`.

7 The helper function `copyCapArray` clears the portion of the captures array between *parenIndex+1* and *parenIndex+parenCount* inclusive each time *m* is repeated. We can see its behaviour in the regular expression

```
    /(z)((a+)?(b+)?(c))*/.exec("zaacbbbcac")
```

which returns the array

```
   ["zaacbbbcac", "z", "ac", "a", undefined, "c"]
```

and not

```
   ["zaacbbbcac", "z", "ac", "a", "bbb", "c"]
```

because each iteration of the outermost `*` clears all captured strings contained in *m*, which in this case includes capture strings numbered 2, 3, and 4.

8 The initial test of `RepeatMatcher`'s closure `d` states that, once the minimum number of repetitions has been satisfied, any more expansions of *m* that match the empty string are not considered for further repetitions. This prevents the regular expression engine from falling into an infinite loop on patterns such as:

```
   /(a*)*/.exec("b")
```

or the slightly more complicated:

```
   /(a*)b\1+/.exec("baaaac")
```

which returns the array

```
   ["b", ""]
```

21.4.5 PositiveLookahead

1 The positive lookahead matcher succeeds without consuming input if its contained matcher can match the input at the current location:

```
function PositiveLookahead(m) : m=m {}

const m: Matcher

public function match(ctx: Context, x: State, c: Continuation): MatchResult {
    let r = m.match(ctx,
                    x,
                    (function (ctx, y: State): MatchResult
                     y) );
    if (r == failure)
        return failure;
    return c(ctx, new State(x.endIndex, r.captures));
}
```

2 If *m* can match at the current position in several ways, only the first one is tried. Unlike other regular expression operators, there is no backtracking into a `(?=` form (this unusual behaviour is inherited from Perl). This only matters when the *m* contains capturing parentheses and the sequel of the pattern contains backreferences to those captures. For example,

```
    /(?=(a+))/.exec("baaabac")
```

matches the empty string immediately after the first `b` and therefore returns the array:

```
    ["", "aaa"]
```

3 To illustrate the lack of backtracking into the lookahead, consider:

```
    /(?=(a+))a*b\1/.exec("baaabac")
```

This expression returns

```
    ["aba", "a"]
```

and not:

```
    ["aaaba", "a"]
```

21.4.6 NegativeLookahead

1 The negative lookahead matcher succeeds without consuming input if its contained matcher fails to match the input at the current location:

```
function NegativeLookahead(m) : m=m {}

const m: Matcher

public function match(ctx: Context, x: State, c: Continuation): MatchResult {
    let r = m.match(ctx,
                    x,
                    (function (ctx, y: State): MatchResult
                     y) );
    if (r != failure)
        return failure;
    return c(ctx, x);
}
```

2 *M* can contain capturing parentheses, but backreferences to them only make sense from within *m* itself. Backreferences to these capturing parentheses from elsewhere in the pattern always return **undefined** because the negative lookahead must fail for the pattern to succeed. For example,

```
    /(.*?)a(?!(a+)b\2c)\2(.*)/.exec("baaabaac")
```

looks for an `a` not immediately followed by some positive number *n* of `a`'s, a `b`, another *n* `a`'s (specified by the first `\2`) and a `c`. The second `\2` is outside the negative lookahead, so it matches against undefined and therefore always succeeds. The whole expression returns the array:

```
    ["baaabaac", "ba", undefined, "abaac"]
```

21.4.7 CharacterSet

1 A `CharacterSet` matches the input at the current location if the canonical representation of the character at the current location is a member of the character set.

```
function CharacterSet(cs)
    : cs=cs {}

const cs: CharsetMatcher;

public function match(ctx: Context, x: State, c: Continuation) /*: MatchResult */ {
    let e = x.endIndex;
    let cap = x.captures;
    if (e == ctx.inputLength)
        return failure;
    let cc = Canonicalize(ctx, ctx.input[e]);
    let res = cs.match(ctx, cc);
    if (!res)
        return failure;
    return c(ctx, new State(e+1, cap));
}
```

2 The helper function `Canonicalize` converts a character to its canonical form. In case-significant matches the canonical form is the character itself. In case-insignificant matches all characters are converted to upper case immediately before they are compared.

```
function Canonicalize(ctx, ch) {
    if (!ctx.ignoreCase)
        return ch;
    let u = ch.toUpperCase();
    if (u.length != 1)
        return ch;
    if (ch.charCodeAt(0) >= 128 && u.charCodeAt(0) < 128)
        return ch;
    return u;
}
```

NOTE If converting a character to upper case would expand that character into more than one character (such as converting "ß" (\u00DF) into "SS"), then the character is left as-is instead. The character is also left as-is if it is not an ASCII character but converting it to upper case would make it into an ASCII character. This prevents Unicode characters such as \u0131 and \u017F from matching regular expressions such as `/[a-z]/i`, which are only intended to match ASCII letters. Furthermore, if these conversions were allowed, then `/[^\W]/i` would match each of `a`, `b`, …, `h`, but not `i` or `s`.

21.4.7.1 Character sets

1 A character set as passed to the `CharacterSet` constructor is a mathematical set of characters. However, in this Standard the sets are provided with a concrete representation in order to present their semantics operationally.

2 A character set is represented as a tree of objects that represent unions, intersections, complements, ranges, and primitive sets containing some number of single characters. These data types are presented in the following sections.

21.4.7.2 CharsetEmpty

1 The empty character set contains no characters, so matching always fails.

```
public function match(ctx: Context, c: string): boolean {
    return false;
}
```

21.4.7.3 CharsetUnion

1 The union of two sets contains a character if either set contains it.

```
function CharsetUnion(m1,m2) : m1=m1, m2=m2 {}

const m1: Charset,
      m2: Charset

public function match(ctx: Context, c: string): boolean {
    return m1.match(ctx, c) || m2.match(ctx, c);
}
```

21.4.7.4 CharsetIntersection

FIXME It has been agreed that character set intersection (which is used to represent set intersection and difference on the source level) be removed.

1 The intersection of two sets contains a character if both sets contain it.

```
function CharsetIntersection(m1,m2) : m1=m1, m2=m2 {}

const m1: Charset,
      m2: Charset

public function match(ctx: Context, c: string): boolean {
    return m1.match(ctx, c) && m2.match(ctx, c);
}
```

21.4.7.5 CharsetComplement

1 A complemented character set contains a character if the contained set does not contain the character.

```
function CharsetComplement(m) : m=m {}

const m: Charset

public function match(ctx: Context, c: string): boolean {
    return !m.match(ctx, c);
}
```

21.4.7.6 CharsetRange

1 A set representing a range contains a character if the character matches the canonicalized value of one of the characters in the range.

```
function CharsetRange(lo,hi) : lo=lo, hi=hi {}

const lo: string,
      hi: string

public function match(ctx: Context, c: string): boolean {
    let lo_code = lo.charCodeAt(0);
    let hi_code = hi.charCodeAt(0);
    for ( let i=lo_code ; i <= hi_code ; i++ )
        if (Canonicalize(ctx, string.intrinsic::fromCharCode(i)) == c)
            return true;
    return false;
}
```

NOTE The strings `lo` and `hi` both contain a single character.

21.4.7.7 CharsetAdhoc

1 An ad-hoc character set contains a character *c* if the *c* is equal to the canonicalized value of one of the characters in the set.

```
function CharsetAdhoc(cs) : cs=explodeString(cs) {}

const cs: [string]

public function match(ctx: Context, c: string): boolean {
    for ( let i=0 ; i < cs.length ; i++ ) {
        if (Canonicalize(ctx, cs[i]) == c)
            return true;
    }
    return false;
}
```

2 The helper function `explodeString` converts a string into an array containing the individual characters in the string, each represented as a one-character string.

```
function explodeString(s : string) : [string] {
    let cs = [] : [string];
    for ( let i=0 ; i < s.length ; i++ )
        cs[i] = s[i];
    return cs;
}
```

21.4.8 Capturing

1 A capturing matcher succeeds if its contained matcher *m* matches the input at the current location. The string that is matched by the contained matcher is saved in the captures array at index *parenIndex*.

```
function Capturing(m, parenIndex) : m=m, parenIndex=parenIndex {}

const m: Matcher,
      parenIndex: uint

public function match(ctx: Context, x: State, c: Continuation): MatchResult {

    let function d( ctx: Context, y: State ): MatchResult {
        let cap = copyCapArray( y.captures, 0, 0 );
        let xe = x.endIndex;
        let ye = y.endIndex;
        cap[parenIndex+1] = ctx.input.substring(xe, ye);
        return c(ctx, new State(ye, cap));
    }

    return m.match(ctx, x, d);
}
```

21.4.9 Backref

1 A back-referencing matcher succeeds if the input at the current location exactly matches the value in the captures array at index *capno*.

```
function Backref(capno) : capno=capno {}

const capno: uint

public function match(ctx: Context, x: State, c: Continuation): MatchResult {
    let cap = x.captures;
    let s = cap[capno];
    if (s == null)
        return c(ctx, x);
    let e = x.endIndex;
    let len = s.length;
    let f = e+len;
    if (f > ctx.inputLength)
        return failure;
    for ( let i=0 ; i < len ; i++ )
        if (Canonicalize(ctx, s[i]) != Canonicalize(ctx, ctx.input[e+i]))
```

```
            return failure;
    return c(ctx, new State(f, cap));
}
```

NOTE An escape sequence of the form \ followed by a nonzero decimal number *n* matches the result of the *n*th set of capturing parentheses (see 15.10.2.11). If the *n*th entry in the captures array is undefined because it hasn't captured anything, then the backreference always succeeds.

21.4.10 Empty

1 An empty match succeeds without consuming input.

```
public function match(ctx: Context, x: State, c: Continuation): MatchResult {
    return c(ctx, x);
}
```

21.5 Methods on the class `RegExp`

21.5.1 new RegExp(pattern, flags)

Description

1 The `RegExp` constructor creates a new regular expression. *Pattern* can be an existing regular expression, in which case the source and flags for the new object is taken from *pattern*.

Implementation

```
function RegExp( pattern, flags )
    : { matcher: matcher,
        names: names,
        source: source,
        multiline: multiline,
        ignoreCase: ignoreCase,
        global: global,
        extended: extended,
        sticky: sticky } = analyzePatternAndFlags(pattern, flags)
    , private::lastIndex = 0
{
}
```

NOTE If the characters of *src* do not have the form *Pattern*, then a **SyntaxError** exception will be thrown.

2 The class `RegExp` has two internal properties `matcher` and and `names`:

```
private const matcher: Matcher,
              names:   [string?]
```

3 The `matcher` property holds the regular expression matcher object (the result of compiling the regular expression). The `names` property holds a string in position *i* if capturing submatch *i* in the pattern string was given a name; the string is that name.

NOTE If pattern is a *StringLiteral*, the usual escape sequence substitutions are performed before the string is processed by `RegExp`. If *pattern* must contain an escape sequence to be recognised by `RegExp`, the `"\"` character must be escaped within the *StringLiteral* to prevent its being removed when the contents of the *StringLiteral* are formed.

NOTE The `source` property of the newly constructed object is set to an implementation-defined string value in the form of a *Pattern* based on *src*.

21.5.2 RegExp(pattern, flags)

Description

1 The `RegExp` class object invoked as a function converts *pattern* to `RegExp`.

2 If *pattern* is a `RegExp` object and *flags* is undefined, then return *pattern* unchanged. Otherwise construct a new regular expression from *pattern* and *flags* and return that.

Returns

3 The `RegExp` function returns a `RegExp` object.

Implementation

```
static meta function invoke( pattern, flags ) {
    if (pattern is RegExp && flags === undefined)
        return pattern;
    else
```

```
        return new RegExp(pattern, flags);
}
```

21.6 Methods on `RegExp` instances

21.6.1 intrinsic::exec (s)

Description

1 The intrinsic `exec` method performs a regular expression match of the string *s* against the regular expression.

Returns

2 The intrinsic `exec` method returns an `Array` object containing the results of the match, or **null** if the string did not match.

Implementation

```
intrinsic function exec(s : string) : Array {
    let length = s.length;
    let i = lastIndex;
    if (!global)
        i = 0;
    let res = failure;
    while (true) {
        if (i < 0 || i > length) {
            lastIndex = 0;
            return null;
        }
        res = matcher.match(s, i, multiline, ignoreCase);
        if (res !== failure)
            break;
        ++i;
    }
    if (global)
        lastIndex = res.endIndex;
    let a = new Array(res.captures.length);
    a.index = i;
    a.input = s;
    a.length = res.captures.length;
    a[0] = s.substring(i,res.endIndex);
    for ( let j=1 ; j < res.captures.length ; j++ )
        a[j] = res.captures[j];
    for ( let j=1 ; j < names.length ; j++ )
        if (names[j] !== null)
            a[names[j]] = res.captures[j];
    return a;
}
```

21.6.2 intrinsic::test (s)

Description

1 The intrinsic `test` method tests whether the string *s* can be successfully matched against the regular expression.

Returns

2 The intrinsic `test` method returns **true** if the string can be matched, and otherwise **false**.

Implementation

```
intrinsic function test(s : string) : boolean
    exec(s) !== null;
```

21.6.3 intrinsic::toString()

Description

1 The intrinsic `toString` method converts the regular expression to a string.

2 Let *src* be a string in the form of a *Pattern* representing the current regular expression. *src* may or may not be identical to the `source` property or to the source code supplied to the `RegExp` constructor; however, if *src* were supplied to the `RegExp` constructor along with the current regular expression's flags, the resulting regular expression must behave identically to the current regular expression.

3 The intrinsic `toString` method produces a string value formed by concatenating the strings "/", *src*, and "/"; plus "g" if the `global` property is true, "i" if the `ignoreCase` property is true, "m" if the `multiline` property is true, "x" if the `extended` property is true, and "y" if the `sticky` property is true.

NOTE An implementation may choose to take advantage of *src* being allowed to be different from the source passed to the `RegExp` constructor to escape special characters in *src*. For example, in the regular expression obtained from `new RegExp("/")`, *src* could be, among other possibilities, "/" or "\/". The latter would permit the entire result ("/\//") of the `toString` call to have the form *RegularExpressionLiteral*.

Returns

4 The intrinsic `toString` method returns a string.

Implementation

5 The intrinsic `toString` method is implementation-defined.

21.6.4 meta::invoke (s)

Returns

1 When a `RegExp` object is called as a function, it invokes the `exec` method on its argument and returns what `exec` returns.

Implementation

```
meta function invoke(s : string) : Array
    exec(s);
```

21.7 Value properties on `RegExp` instances

21.7.1 source

1 The value of the `source` property is `string` in the form of a Pattern representing the current regular expression.

21.7.2 global

1 The value of the `global` property is a `boolean` value indicating whether the flags contained the character `"g"`.

21.7.3 ignoreCase

1 The value of the `ignoreCase` property is a `boolean` value indicating whether the flags contained the character `"i"`.

21.7.4 multiline

1 The value of the `multiline` property is a `boolean` value indicating whether the flags contained the character `"m"`.

21.7.5 extended

1 The value of the `extended` property is a `boolean` value indicating whether the flags contained the character `"x"`.

21.7.6 sticky

1 The value of the `sticky` property is a `boolean` value indicating whether the flags contained the character "`y`".

21.7.7 lastIndex

1 The value of the `lastIndex` property is an integer that specifies the string position at which to start the next match.

2 The value is converted to integer on setting.

Implementation final function get lastIndex() ... final function set lastIndex(x) ...

21.8 Methods on the `RegExp` prototype object

Description

1 The methods on the `RegExp` prototype object call their intrinsic counterparts.

Returns

2 The methods on the `RegExp` prototype object return what their intrinsic counterparts return.

Implementation

```
prototype function exec(this:RegExp, s)
    this.exec(string(s));
```

```
prototype function test(this:RegExp, s)
    this.test(string(s));

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

22 The class `Vector`

1 The class `Vector` is a parameterized, dynamic, direct subclass of `Object`. It represents dense, typed, 0-based, one-dimensional arrays with bounds checking and optionally fixed length.

2 The class `Vector` provides two benefits. One is optimization: the restrictions placed on the class---denseness and a predefined iteration order---make it possible for ECMAScript implementations to implement it particularly efficiently. The other is error checking: `Vector` provides stronger type checking and bounds checking than `Array`.

COMPATIBILITY NOTE The class `Vector` is new in the 4th Edition of this Standard.

3 The class `Vector` provides a method suite that is largely compatible with the class `Array`.

NOTE It is likely that many current uses of `Array` can be switched over to `Vector` without much work, and programs that can be switched will receive the benefits of stronger type and bounds checking.

4 The type parameter of the `Vector` is called its *base type*.

5 As the `Vector` class is dynamic, new properties can be added to its instances but any property whose name is a number (an instance of any class in the union type `AnyNumber`) is handled specially. These properties are called *indexed properties*.

6 Only indexed properties named by nonnegative integers less than the value of the property `length` are defined, and only indexed properties named by nonnegative integers less than 2^{32}-1 can be defined.

7 Any attempt to read an undefined indexed property results in a **RangeError** exception being thrown.

8 Any attempt to write an undefined indexed property results in a **RangeError** being thrown unless the index is equal to the current value of `length`, the current value of `length` is not 2^{32}-1, and the value of the property `fixed` is not **true**.

9 The property `fixed` is a flag that determines whether the vector has fixed length or not. Any attempt to update the value of `length` fails if the `fixed` property has the value **true**.

NOTE If `v` is a `Vector` then reading and writing `v[3.14]` or `v[-3]` will always fail, though reading and writing `v["3.14"]` or `v["-3"]` will succeed.

This behavior deviates from the 3rd Edition, where strings and numbers are interchangeable as property names. But that's no longer quite true in 4th Edition anyway, which has have namespaces and `Name` objects.

Most attempts to set or get properties that are named by numbers that are not valid array indices are probably errors, especially if the object is an Array. Most attempts to read beyond the end of an Array are probably errors. And in a number of cases, attempts to write beyond the end of an Array are probably errors too. The `Vector` class makes it possible to discover these errors.

10 All indexed properties named by nonnegative integers less than `length` are always defined.

NOTE As a consequence, a `Vector` does not have "holes" in its index range in the way an `Array` does.

22.1 Synopsis

1 The class `Vector` provides the following interface:

```
__ES4__ dynamic class Vector.<T> extends Object
{
    public function Vector(length: double=0, fixed: boolean=false) …

    static const length = 2;

    override intrinsic function toString() …
    override intrinsic function toLocaleString() …
    intrinsic function concat(...items): Vector.<T> …
    intrinsic function every(checker: Checker, thisObj: Object=null): boolean …
    intrinsic function filter(checker: Checker, thisObj: Object=null): Vector.<T> …
    intrinsic function forEach(eacher: Eacher, thisObj: Object=null): void …
    intrinsic function indexOf(value: T, from: AnyNumber=0): AnyNumber …
    intrinsic function join(separator: string=","): string …
    intrinsic function lastIndexOf(value: T, from: AnyNumber=Infinity): AnyNumber …
    intrinsic function map(mapper:Mapper, thisObj:Object=null) …
    intrinsic function pop(): T …
    intrinsic function push(...items): double …
    intrinsic function reduce(reducer/*: function*/, initialValue:(T|None)=NONE ): T …
    intrinsic function reduceRight(reducer/*: function*/, initialValue:(T|None)=NONE ): T
…
    intrinsic function reverse(): Vector.<T> …
    intrinsic function shift(): T …
    intrinsic function slice(start: AnyNumber=0, end: AnyNumber=Infinity, step:
AnyNumber=1): Vector.<T> …
    intrinsic function some(checker: Checker, thisObj: Object=null): boolean …
    intrinsic function sort(comparefn: function(T, T): AnyNumber): Vector.<T> …
    intrinsic function splice(start: AnyNumber, deleteCount: AnyNumber, ...items):
Vector.<T> …
    intrinsic function unshift(...items): double …

    iterator function get(deep: boolean = false) : iterator::Iterator.<double> …
    iterator function getKeys(deep: boolean = false) : iterator::Iterator.<double> …
    iterator function getValues(deep: boolean = false) : iterator::Iterator.<T> …
    iterator function getItems(deep: boolean = false) : iterator::Iterator.<[double,T]> …

    public var fixed: boolean …

    public final function get length() …
    public final function set length(len: AnyNumber) …

    meta final function get(name): T …
    meta final function set(name, v): void …
    meta final function has(name) …
    meta final function delete(name) …
}
```

2 The types `Checker`, `Eacher`, and `Mapper` are as for the `Array` class (see class Array).

3 The `Vector` prototype object provides these direct properties:

```
    toString:       function () …
    toLocaleString: function () …
    concat:         function (...items) …
    every:          function (checker, thisObj) …
    filter:         function (checker, thisObj) …
    forEach:        function (eacher, thisObj) …
    indexOf:        function (value, from) …
    join:           function (separator) …
    lastIndexOf:    function (value, from) …
    map:            function (mapper, thisObj) …
    pop:            function () …
    push:           function (...items) …
    reduce:         function (callback, initialValue) …
    reduceRight:    function (callback, initialValue) …
    reverse:        function () …
    shift:          function () …
    slice:          function (start, end) …
    some:           function (checker, thisObj) …
    sort:           function (comparefn) …
    splice:         function (start, deleteCount, ...items) …
    unshift:        function (...items) …
```

22.2 Methods on the `Vector` class object

22.2.1 new Vector.<T> (length=…, fixed=…)

Description

1 The `Vector` constructor initializes a new `Vector` object.

2 *Length* is the inital value of the `length` property. Its default value is zero. Every indexed element of the new vector below *length* is initialized to a default value that is appropriate to the base type `T`.

3 *Fixed* is the initial value of the `fixed` property. Its default value is **false**.

22.3 Methods on `Vector` instances

22.3.1 intrinsic::toString ( )

Description

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

Returns

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

Implementation

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

22.3.2 intrinsic::toLocaleString ( )

Description

1 The intrinsic `toLocaleString` method converts the `Vector` to a string in the following manner.

2 Elements of this `Vector` 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

3 The `toLocaleString` method returns a `string`.

Implementation

```
override intrinsic function toLocaleString() {
    let limit = length;
    let separator = localeSpecificSeparatorString();
    let s = "";
    let i = 0;

    while (true) {
        let x = this[i];
        if (x !== undefined && x !== null)
            s += x.toLocaleString();
        if (++i == limit)
            break;
        s += separator;
    }
    return s;
}
```

NOTE The first parameter to this method 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.

22.3.3 intrinsic::concat (...items)

Description

1 The intrinsic `concat` method collects the vector elements from `this` followed by the vector elements from the additional *items*, in order, into a new `Vector` object. All the *items* must be `Vector` instances whose base types are subtypes of the base type of `this`.

Returns

2 The `concat` method returns a new `Vector` object with the same base type as `this`.

Implementation

```
intrinsic function concat(...items): Vector.<T>
    concat(items);

helper function concat(items) {
    let v = new Vector.<T>;
    let k = 0;

    for ( let i=0, limit=length ; i < limit ; i++ )
        v[k++] = this[i];

    for ( let j=0 ; j < items.length ; j++ ) {
        let item = items[j];
        for ( let i=0, limit=item.length ; i < limit ; i++ )
            v[k++] = item[i];
    }

    return v;
}
```

FIXME Need to check a detail of the type system, namely whether `Vector.<T>` is a subtype of Vector.<U> if T is a subtype of U and U is not `*`.

22.3.4 intrinsic::every (checker, thisObj=…)

Description

1 The intrinsic `every` method calls *checker* on every vector element of `this` in increasing index order, stopping as soon as any call returns **false**.

2 *Checker* is called with three arguments: the vector element value, the vector element index, and `this`. *ThisObj* is used as the `this` object in the call.

Returns

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

Implementation

```
intrinsic function every(checker: Checker, thisObj: Object=null): boolean {
    for ( let i=0, limit=length ; i < limit ; i++ )
        if (!checker.call(thisObj, this[i], i, this))
            return false;
    return true;
}
```

22.3.5 intrinsic::filter (checker, thisObj=…)

Description

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

2 *Checker* is called with three arguments: the vector element value, the vector element index, and `this`. *ThisObj* is used as the `this` object in the call.

Returns

3 The `filter` method returns a new `Vector` object with the same base type as `this`, containing the elements that were collected, in the order they were collected. The length of the new `Vector` is equal to the number of values that were collected.

Implementation

```
intrinsic function filter(checker: Checker, thisObj: Object=null): Vector.<T> {
    var result = new Vector.<T>;
    for ( let i=0, limit=length ; i < limit ; i++ ) {
        let item = this[i];
        if (checker.call(thisObj, item, i, this))
            result[result.length] = item;
    }
    return result;
}
```

22.3.6 intrinsic::forEach (eacher, thisObj=…)

Description

1 The intrinsic `forEach` method calls *eacher* on every vector element of `this` in increasing index order, discarding any return value of *eacher*.

2 *Eacher* is called with three arguments: the vector element value, the vector element index, and `this`. *ThisObj* is used as the `this` object in the call.

Returns

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

Implementation

```
intrinsic function forEach(eacher: Eacher, thisObj: Object=null): void {
    for ( let i=0, limit=length ; i < limit ; i++ )
        eacher.call(thisObj, this[i], i, this);
}
```

22.3.7 intrinsic::indexOf (value, from=…)

Description

1 The intrinsic `indexOf` method compares *value* with every vector element of `this` in increasing index order, starting at the index *from*, stopping when a vector element is equal to *value* by the `===` operator.

2 If *from* is negative, it is treated as `this.length+from`.

Returns

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

Implementation

```
intrinsic function indexOf(value: T, from: AnyNumber=0): AnyNumber {
    let start = clamp( from, length );
    for ( let i=start, limit=length ; i < limit ; i++ ) {
        let item = this[i];
        if (item === value)
            return i;
    }
    return -1;
}
```

4 The helper function `clamp` performs clamping of `from` to the length of this `Vector`.

```
helper function clamp(val: AnyNumber, len: double): double {
    val = toInteger(val);
    if (val < 0)
        val += len;
    return intrinsic::toUint( Math.min( Math.max( val, 0 ), len ) );
}
```

NOTE The helper function `toInteger`, used by `clamp`, is described elsewhere; it performs the `ToInteger` operation of the 3rd Edition.

22.3.8 intrinsic::join (separator=…)

Description

1 The intrinsic `join` method concatenates the string representations of the vector elements of `this` in increasing index order, separating the individual strings by occurrences of *separator*.

Returns

2 The `join` method returns the concatenated string.

Implementation

```
intrinsic function join(separator: string=","): string {
    let limit = length;
    let s = "";
    let i = 0;

    for (let i = 0; i < limit; i++) {
        let item = this[i];
        if (i != 0)
            s += separator;
        if (item is Object)
            s += string(x);
    }
    return s;
}
```

22.3.9 intrinsic::lastIndexOf (value, from=…)

Description

1 The intrinsic `lastIndexOf` method compares *value* with every vector element of `this` in decreasing numerical index order, starting at the index *from*, stopping when a vector element is equal to *value* by the === operator.

2 If *from* is negative, it is treated as `this.length+from`.

Returns

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

Implementation

```
intrinsic function lastIndexOf(value: T, from: AnyNumber=Infinity): AnyNumber {
    let start = clamp( from, length );
    for ( let i=start ; i >= 0 ; i-- ) {
        let item = this[i];
        if (item === value)
            return i;
    }
    return -1;
}
```

22.3.10 intrinsic::map (mapper, thisObj=…)

Description

1 The intrinsic `map` method calls *mapper* on each vector element of `this` in increasing numerical index order, collecting the return values from *mapper*.

2 *Mapper* is called with three arguments: the vector element value, the vector element index, and `this`. *ThisObj* is used as the `this` object in the call.

Returns

3 The `map` method returns a new `Vector` object of the same base type and length as this `Vector`. The element at index *i* in the new vector is the value collected from the call to *mapper* on `this[i]`.

Implementation

```
intrinsic function map(mapper:Mapper, thisObj:Object=null) {
    var result = new Vector.<T>(length);
    for ( let i=0, limit=length ; i < limit ; i++ ) {
        let item = this[i];
        result[i] = mapper.call(thisObj, item, i, this);
    }
    return result;
}
```

22.3.11 intrinsic::pop ()

Description

1 The intrinsic `pop` method extracts the last vector element from `this` and removes it by decreasing the value of the `length` property of `this` by 1.

Returns

2 The `pop` method returns the removed element, or the appropriate default value for the base type of `this` if there are no elements.

Implementation

```
intrinsic function pop(): T {
    if (length == 0)
        return undefined;

    let v = this[length-1];
    length--;
    return v;
}
```

22.3.12 intrinsic::push (...items)

Description

1 The intrinsic `push` method appends the values in *items* to this `Vector`, in the order in which they appear in *items.* The `length` property of this `Vector` will be incremented by the length of *items.*

Returns

2 The `push` method returns the new value of the `length` property of this `Vector`.

Implementation

```
intrinsic function push(...items): double
    push(items);

helper function push(items) {
    for ( let i=0, limit=items.length ; i < limit ; i++ )
        this[length] = items[i];
    return length;
}
```

22.3.13 intrinsic::reverse ()

Description

1 The intrinsic `reverse` method rearranges the vector elements of `this` so as to reverse their order. The `length` property of `this` remains unchanged.

Returns

2 The `reverse` method returns `this`.

Implementation

```
intrinsic function reverse(): Vector.<T> {
    for ( let i=0, j=length-1 ; i < j ; i++, j-- )
        [this[i], this[j]] = [this[j], this[i]];
    return this;
}
```

22.3.14 intrinsic::shift ()

Description

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

Returns

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

Implementation

```
intrinsic function shift(): T {
    if (length == 0)
        return undefined;
    let v = this[0];
    for ( let i=1, limit=length ; i < limit ; i++ )
        this[i-1] = this[i];
    length--;
    return v;
}
```

22.3.15 intrinsic::slice (start=…, end=…, step=…)

Description

1 The intrinsic `slice` method extracts the subrange of array elements from `this` between *start* (inclusive) and *end* (exclusive) into a new Array. Each *step* element is taken.

2 The default value of *start* is 0. If it is negative, it is treated as *`object`*`.length+`*`start`*.

3 The default value of *end* is Infinity. If it is negative, it is treated as *`object`*`.length+`*`end`*.

4 The default value of *step* is 1. If it is 0, it is set to 1.

Returns

5 The `slice` method returns a new `Vector` object with the same base type as `this`, containing the extracted vector elements.

Implementation

```
intrinsic function slice(start: AnyNumber=0, end: AnyNumber=Infinity, step: AnyNumber=1):
Vector.<T> {
    step = toInteger(step);
    if (step == 0)
        step = 1;

    start = clamp(start, length);
    end = clamp(end, length);

    let result = new Vector.<T>;
    if (step > 0)
        for (let i=start; i < end ; i += step)
            result[result.length] = this[i];
    else
        for (let i=start; i > end ; i += step)
            result[result.length] = this[i];

    return result;
}
```

22.3.16 intrinsic::some (checker, thisObj=…)

Description

1 The intrinsic `some` method calls *checker* on every vector element in `this` in increasing index order, stopping as soon as *checker* returns a true value.

2 *Checker* is called with three arguments: the vector element value, the vector element index, and `this`. *ThisObj* is used as the `this` object in the call.

Returns

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

Implementation

```
intrinsic function some(checker: Checker, thisObj: Object=null): boolean {
    for ( let i=0, limit=length ; i < limit ; i++ ) {
        let item = this[i];
        if (checker.call(thisObj, item, i, this))
            return true;
    }
    return false;
}
```

22.3.17 intrinsic::sort (comparefn)

Description

1 The intrinsic::`sort` method sorts the vector elements of `this` according to the ordering defined by `comparefn`.

2 The sort is not necessarily stable (that is, elements that compare equal do not necessarily remain in their original order). *Comparefn* must be a consistent (see sorting-logic) function that accepts two arguments *x* and *y* of the base type of `this` and returns a negative value if $x < y$, zero if $x = y$, or a positive value if $x > y$.

COMPATIBILITY NOTE Unlike the case for `Array`, the *comparefn* is a required argument.

FIXME (Ticket #197.) Should we provide a default comparator?

Returns

3 The `sort` method returns `this`.

Implementation

4 The `sort` method calls on the generic sorting engine, passing a function to compare elements of *this*.

```
intrinsic function sort(comparefn: function(T, T): AnyNumber): Vector.<T> {
    let object = this;
    return sortEngine(object,
                                  0,
                                  length-1,
                                  (function (j, k) comparefn(object[j], object[k])));
}
```

NOTE For a description of the informative `sortEngine` method, see sorting-logic.

FIXME The signature of `comparefn` is probably too constraining, it will require the client to pass a strongly-typed function.

22.3.18 intrinsic::splice (start, deleteCount, ...items)

Description

1 The intrinsic `splice` method replaces the *deleteCount* vector elements of `this` starting at index *start* with values from the *items*.

Returns

2 The `splice` method returns a new `Vector` object of the same base type as `this`, containing the vector elements that were removed from `this`, in order.

Implementation

```
intrinsic function splice(start: AnyNumber, deleteCount: AnyNumber, ...items): Vector.<T>
    splice(start, deleteCount, items);

helper function splice(start, deleteCount, items) {
    let out = new Vector.<T>;
    let len = intrinsic::toUint(length);

    start = clamp( start, len );
    deleteCount = clamp( deleteCount, len - start );
    let end = start + deleteCount;

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

    let insertCount = items.length;
    let shiftAmount = insertCount - deleteCount;

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

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

    for (let i = 0; i < insertCount; i++)
        this[start+i] = items[i];

    length = len + shiftAmount;
    return out;
}
```

22.3.19 intrinsic::unshift (...items)

Description

1 The instrinsic `unshift` method inserts the values in *items* as new vector elements at the start of `this`, such that their order within the vector elements of `this` is the same as the order in which they appear in *items*. Existing vector elements in `this` are shifted upward in the index range, and the `length` property of `this` is updated.

Returns

2 The `unshift` method returns the new value of the `length` property of `this`.

Implementation

```
intrinsic function unshift(...items): double
    unshift(items);

helper function unshift(items) {
    let numitems = items.length;
    let oldlimit = length;
    let newlimit = oldlimit + numitems;

    length = newlimit;
    for ( let i=0 ; i < length ; i++ )
        this[newlimit-i] = this[oldlimit-i];
    for ( let i=0 ; i < numitems ; i++ )
        this[i] = items[i];
    return newlimit;
}
```

22.4 Iteration protocol on `Vector` instances

1 Iterators are defined on the `Vector` such that `for-in` and `for each-in` loops always iterate across the vector from low indices toward high indices. Only indexed properties defined directly on the vector object are visited.

Implementation

```
iterator function get(deep: boolean = false) : iterator::Iterator.<double>
    getKeys(deep);

iterator function getKeys(deep: boolean = false) : iterator::Iterator.<double> {
    let i = 0;
    let a = this;
    return {
        const next:
            function () : double {
                if (i < a.length)
                    return i++;
                throw iterator::StopIteration;
            }
    } : iterator::Iterator.<double>;
}

iterator function getValues(deep: boolean = false) : iterator::Iterator.<T> {
    let i = 0;
    let a = this;
    return {
        const next:
            function () : T {
                if (i < a.length)
                    return a[i++];
                throw iterator::StopIteration;
        }
    } : iterator::Iterator.<T>;
}

iterator function getItems(deep: boolean = false) : iterator::Iterator.<[double,T]> {
    let i = 0;
    let a = this;
    return {
        const next:
        function () : T {
            if (i === a.length)
                return [i,a[i++]];
            throw iterator::StopIteration;
        }
    } : iterator::Iterator.<[double,T]>;
}
```

22.5 Value properties of `Vector` instances

22.5.1 length

1 The property `length` determines the range of valid indices into the `Vector`. Indices up to but not including `length` are always defined.

2 When `length` is given a new value that is smaller than its old value then the elements in the vector at the new length and beyond are removed from the vector.

3 When `length` is given a new value that is greater than its old value then the elements in the vector at the old length and beyond are given a default value that is appropriate to the base type `T`.

4 If an attempt is made to set `length` when the `fixed` property is **true** then a **RangeError** is thrown.

5 If an attempt is made to set `length` to any value that is not a nonnegative integer less than 2^{32} then a **RangeError** is thrown.

22.5.2 fixed

1 The boolean property `fixed` determines whether the `Vector` has fixed length.

2 If `fixed` has the value **true** then any attempt to change `length` will result in in a **RangeError** being thrown.

3 The value of `fixed` is not constant, so vectors can be of fixed length and variable length at different times.

22.5.3 Numerically named properties

1 A `Vector` contains all properties whose names are nonnegative integers below the value of the `Vector`'s `length` property.

2 If an attempt is made to read a property whose name is a number that is not a nonnegative integer below `length` then a **RangeError** is thrown.

3 If an attempt is made to write a property whose name is a number that is not a nonnegative integer below `length` then one of two things happen:

- If the `fixed` property has the value `true`, or if the number is not a nonnegative integer, or if the number is nonnegative but not the same value as the value of `length`, or if `length` is already 2^{32}-1, then a **RangeError** is thrown.
- Otherwise, the property is defined on the vector and the `length` property is incremented by 1.

22.6 Methods on the `Vector` prototype object

Description

1 The methods on the `Vector` prototype object perform a small amount of type conversion and delegate to the corresponding intrinsic methods.

Returns

2 The methods on the `Vector` prototype object return what their corresponding intrinsic methods return.

Implementation

```
prototype function toString(this:Vector.<*>)
    this.intrinsic::toString();

prototype function toLocaleString(this:Vector.<*>)
    this.intrinsic::toLocaleString();

prototype function concat(this:Vector.<*>, ...items)
    this.concat(items);

prototype function every(this:Vector.<*>, checker, thisObj=undefined)
    this.intrinsic::every(checker, thisObj is Object ? thisObj : null);

prototype function filter(this:Vector.<*>, checker, thisObj=undefined)
    this.intrinsic::filter(checker, thisObj is Object ? thisObj : null);

prototype function forEach(this:Vector.<*>, eacher, thisObj=undefined)
    this.intrinsic::forEach(checker, thisObj is Object ? thisObj : null);

prototype function indexOf(this:Vector.<*>, value, from=undefined)
    this.intrinsic::indexOf(value, Number(from));

prototype function join(this:Vector.<*>, separator=undefined)
    this.intrinsic::join(separator == undefined ? "," : string(separator));

prototype function lastIndexOf(this:Vector.<*>, value, from=undefined)
    this.intrinsic::indexOf(value, from == undefined ? Infinity : Number(from));

prototype function map(this:Vector.<*>, mapper, thisObj=undefined)
    this.intrinsic::map(mapper, thisObj is Object ? thisObj : null);

prototype function pop(this:Vector.<*>)
    this.intrinsic::pop();

prototype function push(this:Vector.<*>, ...items)
    this.push(items);

prototype function reverse(this:Vector.<*>)
    this.intrinsic::reverse();

prototype function shift(this:Vector.<*>)
    this.intrinsic::shift();

prototype function slice(this:Vector.<*>, start, end, step)
    this.intrinsic::slice(Number(start), Number(end), Number(step));

prototype function some(this:Vector.<*>, checker, thisObj=undefined)
    this.intrinsic::some(checker, thisObj is Object ? thisObj : null);

prototype function sort(this:Vector.<*>, comparefn)
    this.intrinsic::sort(comparefn);

prototype function splice(this:Vector.<*>, start, deleteCount, ...items)
    this.splice(Number(start), Number(deleteCount), items);

prototype function unshift(this:Vector.<*>, ...items)
    this.unshift(items);
```

23 The class `Map`

1 The class `Map` is a parameterized, dynamic, non-final, direct subclass of `Object` that provides a reliable, efficient, mutable, and iterable map from keys to values. Keys and values may be of arbitrary types.

COMPATIBILITY NOTE The class `Map` is new in the 4th Edition of this Standard.

2 A `Map` is realized as a hash table. When the `Map` is constructed the caller may provide specialized functions that compare keys and compute hash values for keys.

23.1 Synopsis

1 The class `Map` provides the following interface:

```
__ES4__ dynamic class Map.<K,V> extends Object
{
    function Map(equals:   Callable = (function(a,b) a === b),
                 hashcode: Callable = intrinsic::hashcode) …

    static const length = 2;

    intrinsic function size() : double …
    intrinsic function get(key: K, notfound: (V|undefined)=undefined) : (V|undefined) …
    intrinsic function put(key: K, value: V, notfound: (V|undefined)=undefined) : (V|
undefined) …
    intrinsic function has(key:K) : boolean …
    intrinsic function remove(key:K) : boolean …
    intrinsic function clear() : void …

    iterator function get(deep: boolean = false) : iterator::Iterator.<K> …
    iterator function getKeys(deep: boolean = false) : iterator::Iterator.<K> …
    iterator function getValues(deep: boolean = false) : iterator::Iterator.<V> …
    iterator function getItems(deep: boolean = false) : iterator::Iterator.<[K,V]> …

    private const equals   : Callable = …
    private const hashcode : Callable = …
    private var population : uint = …
}
```

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

```
    size:   function () …
    get:    function (key, notfound) …
    put:    function (key, value, notfound) …
    has:    function (key) …
    remove: function (key) …
    clear:  function () …
```

23.2 Methods on the `Map` class object

23.2.1 new Map.<K,V>(equals=…, hashcode=…)

Description

1 The `Map` constructor creates a new map for key type *K* and value type *V*.

2 The optional *equals* argument is a function that compares two keys and returns **true** if they are equal and **false** if they are not. This function must implement a reflexive, transitive, and symmetric relation, and *equals(k1,k2)* must be constant for any two actual keys *k1* and *k2*. The default value for *equals* is a function that compares the two keys using the `===` operator.

3 The optional *hashcode* argument is a function that takes a key and returns a numeric value for it; this key is converted to a `uint` *hash value* for the key. The hash value may be used to find associations more quickly in the map. Two calls to *hashcode* on the same key value must always result in the same hash value, and a call to *hashcode* must always result in the same hash value for two key values that compare equal by the *equals* function. The default value for *hashcode* is the intrinsic global function `hashcode`.

NOTE The constraint that *equals* and *hashcode* return constant values does not apply to key values that are not in a `Map` nor referenced from an activation of any method on `Map`.

NOTE There is no requirement that the values returned from *hashcode* for two unequal keys must be different.

NOTE The operator `==` is not a valid comparator for the global intrinsic function `hashcode` because `==` will consider some values to be equal for which `hashcode` returns different hash values.

Implementation

4 The `Map` constructor initializes the `Map` object by saving its parameters in private storage and initializing the count of the number of associations in the table to zero.

```
function Map(equals   : Callable = (function (x,y) x === y),
ashcode : Callable = intrinsic::hashcode)
quals = equals
ashcode = function (k) intrinsic::toUint(hashcode(k) cast AnyNumber)
```

```
opulation = 0
    {
    }
```

23.3 Methods on `Map` instances

23.3.1 intrinsic::size ( )

Returns

1 The intrinsic method `size` returns the number of associations in the map.

Implementation

```
intrinsic function size() : double
ulation;
```

23.3.2 intrinsic::get (key, notfound=…)

Returns

1 The intrinsic method `get` returns the value associated with *key*, or *notfound* if there is no such association.

Implementation

```
intrinsic function get(key: K, notfound: (V|undefined)=undefined) : (V|undefined) {
 probe = find(key);
urn probe ? probe.value : notfound;
}
```

2 The informative function `find` searches for *key* in the `Map` and returns an object containing at least the properties `key` and `value` if the association was found, or otherwise **null**. (The returned object is part of the `Map` data structure, and writing to it updates the association in the `Map`.)

```
informative function find(key: K): like { key: K, value: V } …
```

23.3.3 intrinsic::put (key, value, notfound=…)

Description

1 The intrinsic method `put` creates an association between *key* and *value*, or overwrites an existing association if there is one.

Returns

2 The `put` method returns the old value of the association if there was one, otherwise it returns *notfound*.

Implementation

```
intrinsic function put(key: K, value: V, notfound: (V|undefined)=undefined) : (V|
undefined) {
 oldvalue = notfound;
 probe = find(key);
(probe) {
 oldvalue = probe.value;
 probe.value = value;

e {
 ++population;
 insert(key, value);

urn oldvalue;
}
```

3 The informative function `insert` adds a new association between *key* and *value* to the `Map`.

```
informative function insert(key: K, value: V): void …
```

23.3.4 intrinsic::has (key)

Returns

1 The intrinsic method `has` returns **true** if there exists an association for *key*, or **false** otherwise.

Implementation

```
intrinsic function has(key:K) : boolean {
 probe = find(key);
urn probe ? true : false;
}
```

23.3.5 intrinsic::remove (key)

Description

1 The intrinsic method `remove` removes any association for *key*.

Returns

2 The `remove` method returns **true** if there was an association for *key*, or **false** otherwise.

Implementation

```
intrinsic function remove(key:K) : boolean {
 probe = find(key);
(probe) {
 --population;
 eject(probe);
 return true;

urn false;
}
```

3 The informative function `eject` removes the association for *key* from the `Map`.

```
informative function eject(box: like { key: K, value: V }): void …
```

23.3.6 intrinsic::clear ( )

Description

1 The intrinsic method `clear` removes all associations from the map.

Returns

2 The `clear` method returns nothing.

Implementation

```
intrinsic function clear() : void {
 (let k in this)
 intrinsic::remove(k);
}
```

23.4 Iteration protocol on `Map` instances

1 The iterator protocol makes use of a helper method `iterate` which first collects the values that will be returned by the iterator methods and then returns an object that provides the correct `next` method:

```
helper function iterate.<T>(f) {
 a = [];
ormative::allItems(function (k,v) { f(a,k,v) });
urn {
 const next:
t (i=0, limit=a.length)
  function () : T {
      if (i < limit)
   return a[i++];
hrow iterator::StopIteration;
        }
 iterator::Iterator.<T>;
}
```

2 The informative function `allItems` calls its function argument on every key/value pair in the `Map`:

```
informative function allItems(fn: function): void …
```

3 The iterator methods `getKeys`, `getValues`, and `getItems` return iterator objects that iterate over keys, values, and key/value pairs, respectively. The iterator method `get` iterates over keys (like `getKeys`).

Implementation

```
iterator function getKeys(deep: boolean = false) : iterator::Iterator.<K>
per::iterate.<K>(function (a,k,v) { a.push(k) });
```

```
iterator function getValues(deep: boolean = false) : iterator::Iterator.<V>
per::iterate.<V>(function (a,k,v) { a.push(v) });

iterator function getItems(deep: boolean = false) : iterator::Iterator.<[K,V]>
per::iterate.<[K,V]>(function (a,k,v) { a.push([k,v]) });

iterator function get(deep: boolean = false) : iterator::Iterator.<K>
rator::getKeys(deep);
```

23.5 Methods on the `Map` prototype object

1 The methods on the `Map` prototype object are constrained to being called on instances of `Map`. They all delegate to the corresponding intrinsic method on their `this` object.

```
prototype function size(this: Map.<*,*>)
s.intrinsic::size();

prototype function get(this: Map.<*,*>, key, notfound)
s.intrinsic::get(key, notfound);

prototype function put(this: Map.<*,*>, key, value, notfound)
s.intrinsic::put(key, value, notfound);

prototype function has(this: Map.<*,*>, key)
s.intrinsic::has(key);

prototype function remove(this: Map.<*,*>, key)
s.intrinsic::remove(key);

prototype function clear(this: Map.<*,*>)
s.intrinsic::clear();
```

24 The meta-object classes

1 The intrinsic meta-object interfaces `Type`, `NominalType`, `ClassType`, `InterfaceType`, `AnyType`, `UndefinedType`, `NullType`, `UnionType`, `RecordType`, `ArrayType`, `FunctionType`, `Field`, and `FieldValue`, along with the intrinsic helper types `FieldIterator`, `NominalTypeIterator`, `TypeIterator`, `ValueIterator`, and `FieldValueIterator`, provide a simple reflection capability.

2 The standard meta-objects described by the interface types may be immutable.

FIXME (Ticket #199.) Right now the class `Class` is not defined in ES4. (Nor is there an `Interface`, nor are there described classes for other type objects.) If these are defined, then we must decide whether they implement the meta-object interfaces or not. Otherwise we have problems of nonportability due to name shadowing in some systems but not in others, or subclassability in some systems but not in others.

3 ECMAScript implementations may choose to provide extensions to these interfaces, in order to provide richer reflective capabilities. Clients wishing to use extended meta-object interfaces can perform runtime downcasts on the meta-objects described by this Standard.

FIXME (Ticket #200.) The meta-objects system does not currently deal with parameterized types. It will almost certainly need to. First, classes can contain type definitions and those definitions can be public, and it would be strange if they could not be iterated by the `publicMembers` and `publicStaticMembes` methods of `NominalType`. But type definitions can bind type parameters, so there would need to be a representation of that. Second, method definitions can bind type parameters (this facility is used in eg the `Map` class for internal helper methods), so even if type definitions were not iterated we'd have to deal with the problem for methods.

There is also the question of what the meaning of `typeOf(Map)` means, when `Map` is a paremeterized type. Presumably that too needs to reveal parameterization in some way.

The original meta-objects proposal had some facilities for parameterization in the context of instantiation: the `construct` methods would take a type iterator that would provide values for type parameters. But that's not good enough, and the following spec does not provide for these. Something like that will come back in, though.

There will probably be a new interface `ParemeterizedType` that acts as a binder for type parameters:

```
interface ParameterizedType
{
    function numberOfParameters()
    function construct(typeVals: TypeIterator): Type
}
```

We must decide whether only instantiated types can be manipulated or not. If so, then we could then arrange for a "dummy type" factory to allow easy instantiation for inspection purposes:

```
intrinsic function dummyTypes(t: Type): TypeIterator
```

That may or may not run afoul of type checking, if any type checking happens at instantiation time. The alternative would be to provide an interface for `TypeParameter`, and for `ParameterizedType` to allow inspection of its contained type without instantiation, with the proviso that `TypeParameter` values will pop up during inspection and will have to be handled.

NOTE In the following sections all interfaces, types, and methods are implicitly defined in the `intrinsic` namespace. The methods defined on the interfaces are not defined on the prototypes of the classes that implement those interfaces.

NOTE Type annotations that denote nullable types are revealed as a union of `NullType` and another type.

24.1 Retrieving the type of an object

24.1.1 typeOf (v)

Description

1 The global intrinsic function `typeOf` delivers the run-time type of its argument *v*, which may be a value of any type.

Returns

2 The function `typeOf` returns an object that implements `Type` and possibly one of the interfaces extending `Type`.

Implementation

3 The function `typeOf` is implementation-dependent.

24.2 The interface `Type`

1 The intrinsic interface `Type` describes a type in the system in basic terms.

24.2.1 Synopsis

```
interface Type
{
    function canConvertTo(t: Type): boolean
    function isSubtypeOf(t: Type): boolean
}
```

24.2.2 Methods

24.2.2.1 canConvertTo (t)

Returns

1 The `canConvertTo` method returns **true** if this type can be converted to the type *t*, otherwise it returns **false**.

24.2.2.2 isSubtypeOf (t)

Returns

1 The `isSubtypeOf` method returns **true** if this type is a subtype of the type *t*, otherwise it returns **false**.

24.3 The interface `Field`

1 The intrinsic interface `Field` describes a field (property) of a class, record, or array type by the field name and field type.

24.3.1 Synopsis

```
interface Field
{
    function name(): Name
    function type(): Type
}
```

24.3.2 Methods

24.3.2.1 name ( )

Returns

1 The `name` method returns the field name as a `Name` object.

24.3.2.2 type ( )

Returns

1 The `type` method returns the field type as an object that implements `Type` or one of the interfaces extending `Type`.

24.4 The interface `FieldValue`

1 The intrinsic interface `FieldValue` describes a field (property) of a record or array by the field name and field value. It is used for constructing new record and array instances.

24.4.1 Synopsis

```
interface FieldValue
{
    function name(): Name
    function value(): *
}
```

24.4.2 Methods

24.4.2.1 name ( )

Returns

1 The `name` method returns the field name as a `Name` object.

24.4.2.2 value ( )

Returns

1 The `value` method returns the actual field value as an ECMAScript value.

24.5 The interface `NominalType`

1 The intrinsic interface `NominalType` is a base interface for `InterfaceType` and `ClassType`. It provides accessors for aspects common to those two types.

24.5.1 Synopsis

```
interface NominalType extends Type
{
    function name(): Name
    function superTypes(): NominalTypeIterator
    function publicMembers(): FieldIterator
    function publicStaticMembers(): FieldIterator
}
```

24.5.2 Methods

24.5.2.1 name ( )

Returns

1 The `name` method returns the name of the nominal type as a `Name` object.

24.5.2.2 superTypes ( )

Returns

1 The `superTypes` method returns an iterator that iterates over the superclasses and implemented interfaces of this nominal type.

24.5.2.3 publicMembers ( )

Returns

1 The `publicMembers` method returns an iterator that iterates over the field definitions of all public instance fields (both method properties and value properties).

24.5.2.4 publicStaticMembers ( )

Returns

1 The `publicMembers` method returns an iterator that iterates over the field definitions of all public class fields (both method properties and value properties).

2 The constructor method is included in the set of static members, provided that it is public.

FIXME (Ticket #200.) Likely, this will also iterate type definitions (see comment at the start of this chapter).

24.6 The interface `InterfaceType`

1 The intrinsic interface `InterfaceType` describes an interface.

24.6.1 Synopsis

```
interface InterfaceType extends NominalType
{
    function implementedBy(): ClassTypeIterator
}
```

24.6.2 Methods

24.6.2.1 implementedBy ( )

Returns

1 The `implementedBy` method returns an interator that iterates over all the class types that implement this interface.

24.7 The interface `ClassType`

1 The intrinsic interface `ClassType` describes a class and provides a means of creating new instances of the class.

24.7.1 Synopsis

```
interface ClassType extends NominalType
{
    function construct(valArgs: ValueIterator): Object
}
```

24.7.2 Methods

24.7.2.1 construct (valArgs)

Description

1 The `construct` method creates a new instance of the class represented by this `ClassType`, provided the class's constructor is public.

2 The iterator *valArgs* provides any value arguments required by the constructor. Only as many values as necessary for calling the constructor will be consumed from the iterator. If the constructor takes no arguments then *valArgs* may be null.

Returns

3 The `construct` method returns a new object of the type represented by this `ClassType`.

24.8 The interface `AnyType`

1 The intrinsic interface `AnyType` describes the type `*`.

24.8.1 Synopsis

```
interface AnyType extends Type
{
}
```

24.9 The interface `NullType`

1 The intrinsic interface `NullType` describes the type `null`.

24.9.1 Synopsis

```
interface NullType extends Type
{
}
```

24.10 The interface `UndefinedType`

1 The intrinsic interface `UndefinedType` describes the type `undefined`.

24.10.1 Synopsis

```
interface UndefinedType extends Type
{
}
```

24.11 The interface `UnionType`

1 The intrinsic interface `UnionType` describes a union of other types. No object has a union type for its manifest type. Union types are only used for annotating parameters or fields, and cannot be instantiated.

24.11.1 Synopsis

```
interface UnionType extends Type
{
    function members(): TypeIterator
}
```

24.11.2 Methods

24.11.2.1 members ( )

Returns

1 The `members` method returns an iterator that iterates over the member types of the union.

24.12 The interface `RecordType`

1 The intrinsic interface `RecordType` describes a structural object type.

24.12.1 Synopsis

```
interface RecordType extends Type
{
    function fields(): FieldIterator
    function construct(valArgs: FieldValueIterator): Object
}
```

24.12.2 Methods

24.12.2.1 fields ( )

Returns

1 The `fields` method returns an iterator that iterates over the fields of the record type.

24.12.2.2 construct (valArgs)

Description

1 The `construct` method creates a new instance of the structural object type represented by this `RecordType`.

2 The iterator *valArgs* provides any field names and values required to initialize the object. All values will be consumed from the iterator; the iterator may provide more field names and values than are required by the type. If the iterator does not provide a value for a field required by the type, the field will be initialized to **undefined** cast to the type of the field; this may cause a **TypeError** exception to be thrown at run-time.

3 *ValArgs* may not be null.

Returns

4 The `construct` method returns a new object of the type represented by this `RecordType`.

24.13 The interface `ArrayType`

1 The intrinsic interface `ArrayType` describes a structural array type.

24.13.1 Synopsis

```
interface ArrayType extends Type
{
    function fields(): FieldIterator
    function construct(length: uint, valArgs: FieldValueIterator): Object
}
```

24.13.2 Methods

24.13.2.1 fields ( )

Returns

1 The `fields` method returns an iterator that iterates over the fields of the array type. The fields are iterated from low to high indices, and only fields that are present are iterated. The name of the field provides the field index as the `identifier`.

24.13.2.2 construct (length, valArgs)

Description

1 The `construct` method creates a new instance of the structural array type represented by this `ArrayType`.

2 The value *length* provides the value for the length of the array; it is set after all fields have been initialized.

3 The iterator *valArgs* provides any field names and values required to initialize the object. All values will be consumed from the iterator; the iterator may provide more field names and values than are required by the type. The field name must encode the correct array index of the field in the `identifier`. If the iterator does not provide a value for a field required by the type, the field will be initialized to **undefined** cast to the type of the field; this may cause a **TypeError** exception to be thrown at run-time.

4 *ValArgs* may not be null.

Returns

5 The `construct` method returns a new object of the type represented by this `ArrayType`.

24.14 The interface `FunctionType`

1 The intrinsic interface `FunctionType` describes a structural function type. Function types cannot be instantiated.

24.14.1 Synopsis

```
interface FunctionType extends Type
{
    function boundThis(): Type
    function parameterTypes(): TypeIterator
    function defaultValues(): ValueIterator
    function hasRestType(): boolean
    function returnType(): Type
}
```

24.14.2 Methods

24.14.2.1 boundThis ( )

Returns

1 The `boundThis` method returns a type if the function has a bound `this` value, otherwise it returns an `AnyType` object.

24.14.2.2 parameterTypes ( )

Returns

1 The `parameterTypes` method returns an iterator that iterates over the types of the formal parameters of the function, starting with the first argument and iterating in order, including all optional and rest arguments.

2 Arguments that do not have annotations will be revealed as type `AnyType`.

24.14.2.3 defaultValues ( )

Returns

1 The `defaultValues` method returns an iterator that iterates over the default values of the optional arguments, starting with the first default value and iterating in order.

24.14.2.4 hasRestTypes()

Returns

1 The `hasRestTypes` method returns **true** if the function has a rest argument, **false** otherwise.

24.14.2.5 returnType ( )

Returns

1 The `returnType` method returns the return type annotation for this function, or an `AnyType` object if there was no type annotation.

24.15 Iterator types

1 The following iterator type definitions are used as annotations on parameters and methods in the interface hierarchy described previously.

```
type FieldIterator = iterator::IteratorType.<Field>
type ClassTypeIterator = iterator::IteratorType.<ClassType>
type NominalTypeIterator = iterator::IteratorType.<NominalType>
type TypeIterator = iterator::IteratorType.<Type>
type FieldValueIterator = iterator::IteratorType.<FieldValue>
type ValueIterator = iterator::IteratorType.<*>
```

25 Error classes

1 ECMAScript provides a hierarchy of standard pre-defined error classes rooted at the class `Error` (see class Error).

2 The ECMAScript implementation throws a new instance of one of the pre-defined 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 pre-defined error classes 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 pre-defined subclasses are non-final and dynamic and may be subclassed by user-defined exception classes.

5 All the pre-defined subclasses of `Error` share the same structure.

26 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.

26.1 Synopsis

1 The class `Error` provides the following interface:

```
dynamic class Error extends Object
{
    public function Error(message) …
    static meta function invoke(message) …

    static public const length = 1

    override intrinsic function toString() …
    override helper function getClassName() …
}
```

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

```
    toString: function () …
    name:     "Error"
    message:  …
```

26.2 Methods on the `Error` class

26.2.1 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

```
public function Error(message) {
    if (message !== undefined)
        this.message = string(message);
}
```

26.2.2 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

```
static meta function invoke(message)
    new Error(message);
```

26.3 Methods on `Error` instances

26.3.1 intrinsic::toString ( )

Description

1 The intrinsic `toString` method converts the `Error` object to an implementation-defined string.

Returns

2 A `string` object.

Implementation

```
override intrinsic function toString()
    private::toString();
```

3 The private function `toString` is implementation-defined.

26.3.2 helper::getClassName ( )

Description

1 The helper method `getClassName` overrides the method defined in `Object` and makes the pre-defined subclasses of `Error` appear to have the `[[Class]]` value `"Error"`.

NOTE The helper method `getClassName` is a specification artifact. The protocol it defines for overriding `[[Class]]` is not available to user code.

Returns

2 The helper method `getClassName` returns a string.

Implementation

```
override helper function getClassName() {
    if (isExactlyType(this, EvalError) ||
        isExactlyType(this, RangeError) ||
        isExactlyType(this, ReferenceError) ||
        isExactlyType(this, SyntaxError) ||
        isExactlyType(this, TypeError) ||
        isExactlyType(this, URIError))
        return "Error";
    return super.getClassName();
}

helper function isExactlyType(obj, cls)
    let (objtype = reflect::typeOf(obj))
        cls.reflect::isSubtypeOf(objtype) && objtype.reflect::isSubtypeOf(cls);
```

26.4 Methods on the `Error` prototype object

26.4.1 toString ( )

Description

1 The prototype `toString` method calls the private `toString` method.

Returns

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

Implementation

```
public prototype function toString()
    this.private::toString();
```

26.5 Value properties on the `Error` prototype object

26.5.1 name

1 The initial value of the `name` prototype property is the string "`Error`".

26.5.2 message

1 The initial value of the `message` prototype property is an implementation-defined string.

27 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 sections XXX.

FIXME Clean up the section references when we reach final draft.

27.1 Synopsis

1 The `EvalError` class provides this interface:

```
dynamic class EvalError extends Error
{
    public function EvalError(message) …
    static meta function invoke(message) …

    static public const length = 1
}
```

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

```
    name:    "EvalError"
    message: …
```

27.2 Methods on the `EvalError` class

27.2.1 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

```
public function EvalError(message)
    : super(message)
{
}
```

27.2.2 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

```
static meta function invoke(message)
    new EvalError(message);
```

27.3 Value properties on the `EvalError` prototype object

27.3.1 name

1 The initial value of the `name` prototype property is the string "`EvalError`".

27.3.2 message

1 The initial value of the `message` prototype property is an implementation-defined string.

28 The class `RangeError`

1 The implementation throws a new `RangeError` instance when it detects that a numeric value has exceeded the allowable range. See sections XXX.

FIXME Clean up the section references when we reach final draft.

28.1 Synopsis

1 The `RangeError` class provides this interface:

```
dynamic class RangeError extends Error
{
    public function RangeError(message) …
    static meta function invoke(message) …

    static public const length = 1
}
```

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

```
    name:    "RangeError"
    message: …
```

28.2 Methods on the `RangeError` class

28.2.1 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

```
public function RangeError(message)
    : super(message)
{
}
```

28.2.2 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

```
static meta function invoke(message)
    new RangeError(message);
```

28.3 Value properties on the `RangeError` prototype object

28.3.1 name

1 The initial value of the `name` prototype property is the string "`RangeError`".

28.3.2 message

1 The initial value of the `message` prototype property is an implementation-defined string.

29 The class `ReferenceError`

1 The implementation throws a new `ReferenceError` instance when it detects an invalid reference value. See sections XXX.

FIXME Clean up the section references when we reach final draft.

29.1 Synopsis

```
dynamic class ReferenceError extends Error
{
    public function ReferenceError(message) …
    static meta function invoke(message) …

    static public const length = 1
}
```

1 The `ReferenceError` prototype object provides these direct properties:

```
    name:    "ReferenceError"
    message: …
```

29.2 Methods on the `ReferenceError` class

29.2.1 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

```
public function ReferenceError(message)
    : super(message)
{
}
```

29.2.2 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

```
static meta function invoke(message)
    new ReferenceError(message);
```

29.3 Value properties on the `ReferenceError` prototype object

29.3.1 name

1 The initial value of the `name` prototype property is the string "`ReferenceError`".

29.3.2 message

1 The initial value of the `message` prototype property is an implementation-defined string.

30 The class `SyntaxError`

1 The implementation throws a new `SyntaxError` instance when a parsing error has occurred. See sections XXX.

FIXME Clean up the section references when we reach final draft.

30.1 Synopsis

```
dynamic class SyntaxError extends Error
{
    public function SyntaxError(message) …
    static meta function invoke(message) …

    static public const length = 1
}
```

1 The `SyntaxError` prototype object provides these direct properties:

```
    name:    "SyntaxError"
    message: …
```

30.2 Methods on the `SyntaxError` class

30.2.1 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

```
public function SyntaxError(message)
    : super(message)
{
}
```

30.2.2 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

```
static meta function invoke(message)
    new SyntaxError(message);
```

30.3 Value properties on the `SyntaxError` prototype object

30.3.1 name

1 The initial value of the `name` prototype property is the string "`SyntaxError`".

30.3.2 message

1 The initial value of the `message` prototype property is an implementation-defined string.

31 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 sections XXX.

FIXME Clean up the section references when we reach final draft.

31.1 Synopsis

```
dynamic class TypeError extends Error
{
    public function TypeError(message) …
    static meta function invoke(message) …

    static public const length = 1
}
```

1 The `TypeError` prototype object provides these direct properties:

```
    name:    "TypeError"
    message: …
```

31.2 Methods on the `TypeError` class

31.2.1 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

```
public function TypeError(message)
    : super(message)
{
}
```

31.2.2 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

```
static meta function invoke(message)
    new TypeError(message);
```

31.3 Value properties on the `TypeError` prototype object

31.3.1 name

1 The initial value of the `name` prototype property is the string "`TypeError`".

31.3.2 message

1 The initial value of the `message` prototype property is an implementation-defined string.

32 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 sections XXX.

FIXME Clean up the section references when we reach final draft.

32.1 Synopsis

```
dynamic class URIError extends Error
{
    public function URIError(message) …
    static meta function invoke(message) …

    static public const length = 1
}
```

1 The `URIError` prototype object provides these direct properties:

```
    name:    "URIError"
    message: …
```

32.2 Methods on the `URIError` class

32.2.1 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

```
public function URIError(message)
    : super(message)
{
}
```

32.2.2 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

```
static meta function invoke(message)
    new URIError(message);
```

32.3 Value properties on the `URIError` prototype object

32.3.1 name

1 The initial value of the `name` prototype property is the string "`URIError`".

32.3.2 message

1 The initial value of the `message` prototype property is an implementation-defined string.