Part III

Native ECMAScript Objects

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.

FIXME There may be multiple global objects.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
intrinsic function nanoAge() : double
    informative::nanoAge();
```

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

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

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

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

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

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

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

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

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

2 The Global Object

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

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

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

```
namespace __ES4__

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

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

__ES4__ type EnumerableId = …
__ES4__ type Numeric = …

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

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

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

__ES4__ const global: Object = …

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

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

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

__ES4__ function hashcode(x) …
```

2.1 Namespace for types

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

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

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

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

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

2.2 Value Properties of the Global Object

2.2.1 NaN

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

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

2.2.2 Infinity

1 The value of `Infinity` is $+\infty$ (section 8.5).

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

2.2.3 undefined

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

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

2.2.4 __ECMASCRIPT_VERSION__

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

NOTE This property is new in 4th Edition.

2.3 Function Properties of the Global Object

2.3.1 eval (s)

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

2 When the intrinsic and non-intrinsic `eval` functions are called as methods on the global objects in whose scope they are closed then they evaluate their argument as a program in the global scope that is the receiver object in the call.

3 When the intrinsic and non-intrinsic `eval` functions are called as ordinary functions under other names than `eval` then they evaluate their argument as a program in a global scope that is the scope in which the `eval` function was closed.

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

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

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

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

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

2.3.2 parseInt (string , radix)

1 The intrinsic `parseInt` function produces an integer value dictated by interpretation of the contents of the string argument `s` according to the specified radix `r`. Leading whitespace in `s` is ignored. If `r` is 0, it is assumed to be 10 except when the number begins with the character pairs 0x or 0X, in which case a radix of 16 is assumed. Any radix-16 number may also optionally begin with the character pairs 0x or 0X.

2 When the intrinsic `parseInt` function is called, the following steps are taken:

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

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

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

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

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

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

    if (s == "")
        return NaN;

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

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

3 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.

4 The non-intrinsic `parseInt` function converts its first argument to string and its second argument to int, and then calls its intrinsic counterpart:

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

2.3.2.1 isDigitForRadix

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

2.3.3 parseFloat (string)

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

2 When the intrinsic `parseFloat` function is called, the following steps are taken:

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

NOTE `parseFloat` may interpret only a leading portion of the string 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 The non-intrinsic `parseFloat` function converts its argument to string, then calls its intrinsic counterpart:

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

2.3.4 isNaN (number)

1 The intrinsic `isNaN` function takes a numeric value and returns **true** if it is **NaN**, and otherwise returns **false**.

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

2 The non-intrinsic `isNaN` function converts its argument to a number, then calls its intrinsic counterpart:

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

2.3.5 isFinite (number)

1 When the intrinsic `isFinite` function is called on a number, the following steps are taken:

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

2 The non-intrinsic `isFinite` function converts its argument to a number, then calls its intrinsic counterpart:

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

2.3.6 URI Handling Function Properties

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

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

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

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

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

uri :::
    uriCharacters$_{opt}$

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

uriCharacter :::
    uriReserved
    uriUnescaped
    uriEscaped

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

uriUnescaped :::
    uriAlpha
    DecimalDigit
    uriMark

uriEscaped :::
    % HexDigit HexDigit

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

uriMark ::: ***one of***
    `- _ . ! ~ * ' ( )`

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
            else {
                let C = string.fromCharCode(V);
                if (reservedSet.indexOf(C))
                    R = R + s.substring(start, k);
                else
                    R = R + C;
            }
        }
        return R;
    }

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

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

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

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

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

8 Where

uuuuu = vvvv + 1

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

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

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

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

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

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

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

helper const uriAlpha = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ";

helper const uriDigit = "0123456789";

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

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

2.3.6.1 decodeURI (encodedURI)

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.

2 When the intrinsic `decodeURI` function is called with one string argument `encodedURI`, the following steps are taken:

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

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

3 The non-intrinsic `decodeURI` function converts its argument to string, then calls its intrinsic counterpart:

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

2.3.6.2 decodeURIComponent (encodedURIComponent)

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.

2 When the intrinsic `decodeURIComponent` function is called with one string argument `encodedURIComponent`, the following steps are taken:

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

3 The non-intrinsic `decodeURIComponent` function converts its argument to string, then calls its intrinsic counterpart:

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

2.3.6.3 encodeURI (uri)

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

2 When the intrinsic `encodeURI` function is called with one string argument `uri`, the following steps are taken:

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

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

3 The non-intrinsic `encodeURI` function converts its argument to string, then calls its intrinsic counterpart:

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

2.3.6.4 encodeURIComponent (uriComponent)

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

2 When the intrinsic `encodeURIComponent` function is called with one string argument `uriComponent`, the following steps are taken:

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

3 The non-intrinsic `encodeURIComponent` function converts its argument to string, then calls its intrinsic counterpart:

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

2.3.7 hashcode (x)

1 The intrinsic `hashcode` function computes an unsigned integer 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.

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

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

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

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

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

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

2.3.8 Operator functions

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

2.4 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 object-object
- The `Function` class is defined in section function-object
- The `Name` class is defined in section name-object
- The `Namespace` class is defined in section namespace-object
- The `Array` class is defined in section array-object
- The `ByteArray` class is defined in section bytearray-object
- The `String` and `string` classes are defined in section string-object
- The `Boolean` and `boolean` classes are defined in section boolean-object
- The `Number`, `int`, `uint`, `double`, and `decimal` classes are defined in section number-object
- The `Date` class is defined in section date-object
- The `RegExp` class is defined in section regexp-object
- The `Map` class is defined in section map-object
- The `Error` class and its subclasses `EvalError`, `RangeError`, `ReferenceError`, `SyntaxError`, `TypeError`, and `URIError` are defined in section error-object

2.5 Type Properties on the Global Object

2.5.1 EnumerableId

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

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

2.5.2 Numeric

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

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

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

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

2.7 Other Properties of the Global Object

2.7.1 Math

1 See section math-object.

2.7.2 global

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

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

NOTE This property is new in 4th Edition.

3 Object Objects

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

2 The class `Object` has the following interface:

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

    meta static function invoke(value=undefined) …

    static const length: uint = 1
    static const prototype: Object = …

    intrinsic function toString(): string …
    intrinsic function toLocaleString(): string …
    intrinsic function toJSONString(...args): string …
    intrinsic function valueOf(): Object! …
    intrinsic function hasOwnProperty(V : EnumerableId): boolean …
    intrinsic function isPrototypeOf(V): boolean
    intrinsic function propertyIsenumerable(prop: EnumerableId,
                                           e: (undefined,boolean)=undefined): boolean …
}
```

3.1 The Object Class Called as a Function

1 When `Object` is called as a function it performs a type conversion.

3.1.1 Object (value)

1 When the `Object` class is called as a function with no arguments or with one argument value, the following steps are taken:

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

3.2 The Object Constructor

1 The `Object` constructor is invoked when `Object` is used in a `new` experssion to create an object.

3.2.1 new Object (value)

1 When the `Object` constructor is called with no arguments or with one argument value, the following steps are taken:

FIXME This needs to be elucidated. Probably, quoting the SML code is not the right thing here.

```
fun specialObjectConstructor(...) =
    MISSING SML CODE
```

3.3 Properties of the Object Class

1 The value of the internal `[[Prototype]]` property of the `Object` class is the `Object` prototype object.

2 Besides the internal properties and the `length` property (whose value is 1), the `Object` class has the following properties:

3.3.1 Object.prototype

1 The initial value of `Object.prototype` is the `Object` prototype object (obj-proto-props).

3.4 Methods on Object instances

3.4.1 toString ( )

1 When the intrinsic `toString` method is called, the following steps are taken:

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

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

3.4.2 toLocaleString ( )

1 The intrinsic `toLocaleString` method returns the result of calling `toString()`.

FIXME Precisely *which* toString is that? The point here is that it will pick up some user-overridden toString in the subclass. But for backwards compatibility that must be the toString defined on an object, or in the object's prototype. So the intrinsic `toLocaleString` can't call `intrinsic::toString` unless there are separate protocol hierarchies for the intrinsic and prototype methods, and right now there isn't.

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

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.

3.4.3 toJSONString ( )

FIXME Waiting for proposal to be cleaned up and the RI method to be implemented. This may have the same issues as `toLocaleString`: care must be taken.

3.4.4 valueOf ( )

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

FIXME This may have the same issues as `toLocaleString`: care must be taken.

3.4.5 hasOwnProperty (V)

1 When the intrinsic `hasOwnProperty` method is called with argument `V`, the following steps are taken:

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

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

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

3.4.6 isPrototypeOf (V)

1 When the intrinsic `isPrototypeOf` method is called with argument `V`, the following steps are taken:

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

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

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

3.4.7 propertyIsEnumerable (V, e)

1 When the intrinsic `propertyIsEnumerable` method is called on an object with a property name *V* and an optional second argument *e*, the following steps are taken:

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

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

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

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

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

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

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

3.5 Properties of the Object Prototype Object

1 The value of the internal `[[Prototype]]` property of the Object prototype object is null and the value of the internal `[[Class]]` property is `"Object"`.

3.5.1 Methods on the Object prototype object

1 The methods on the object prototype object all call the corresponding intrinsic methods of the Object class, as follows.

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

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

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

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

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

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

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

3.5.2 Object.prototype.constructor

1 The initial value of `Object.prototype.constructor` is the built-in `Object` constructor.

3.6 Properties of Object Instances

1 `Object` instances have no special properties beyond those inherited from the `Object` prototype object.

4 Function Objects

1 All function objects in ECMAScript are instances of the class `Function` or one of its subclasses:

```
dynamic class Function
{
    function Function(...args) …

    meta static function invoke(...args) …

    static function apply(fn: Function!, thisArg: Object, argArray: Array) …
    static function call(fn: Function!, thisArg: Object, ...args) …

    static const length: uint = 1
    static const prototype: Object = …

    meta final function invoke(...args) …

    override
    intrinsic function toString(): string …

    intrinsic function apply(thisArg:Object, argArray) …
    intrinsic function call(thisArg:Object, ...args) …
    intrinsic function HasInstance(V) …

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

FIXME Need to write about how the final function `invoke` implements `[[Call]]` and `[[Construct]]` semantics, and how its API is not really what's written there but something that is customized to the function instance created. This also ties in to how `length` is handled: every function object has a separate `length` property. That may be best modelled as a getter or setter accessing some internal property, in the same way invoke accesses internal properties (for compiled code). Whether to put that description here or in the last section below or in some chapter on execution semantics... hard to say.

FIXME It may be worth noting that there are callable objects that are not function objects...

FIXME Also need to describe how subclasses work, and note that the non-static `invoke` function is `final`, therefore subclasses can add properties and methods but can't override the function calling behavior.

4.1 The Function Class Called as a Function

1 When the `Function` class 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.

4.1.1 Function (p1, p2, … , pn, body)

1 When the `Function` class is called with some arguments *p1*, *p2*, … , *pn*, *body* (where *n* might be 0, that is, there are no "*p*" arguments, and where *body* might also not be provided), the following steps are taken:

2 Create and return a new Function object as if the function constructor had been called with the same arguments (section).

4.2 The Function Constructor

1 When `Function` is called as part of a `new` expression, it is a constructor: it initialises the newly created object.

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

1 The last argument specifies the body (executable code) of a function; any preceding arguments specify formal parameters.

2 When the Function constructor is called with some arguments *p1*, *p2*, … , *pn*, *body* (where *n* might be 0, that is, there are no "*p*" arguments, and where body might also not be provided), the following steps are taken:

```
function Function(...args) {
    let parameters = "";
    let body = "";
    if (args.length > 0) {
        body = args[args.length-1];
```

```
            args.length = args.length-1;
            parameters = args.join(",");
        }
        body = string(body);
        magic::initializeFunction(this, intrinsic::global, parameters, body);
    }
```

3 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.

4 If the list of parameters is not parsable as a *FormalParameterList*$_{opt}$, or if the body is not parsable as a *FunctionBody*, then throw a **SyntaxError** exception.

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

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

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

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

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

4.3 Properties of the Function Constructor

1 The value of the internal `[[Prototype]]` property of the Function constructor is the Function prototype object (section Function.prototype).

2 Besides the internal properties and the `length` property (whose value is 1), the Function constructor has the following properties:

4.3.1 Function.prototype

1 The initial value of `Function.prototype` is the Function prototype object (section Function.prototype).

4.3.2 Function.apply

1 The static `apply` method takes two arguments, *thisArg* and *argArray*, and performs a function call using the `[[Call]]` property of the object. If the object does not have a `[[Call]]` property, a **TypeError** exception is thrown.

```
static function apply(fn : Function!, thisArg, argArray) {
    if (thisArg === undefined || thisArg === null)
        thisArg = global;
    if (argArray === undefined || argArray === null)
        argArray = [];
    else if (!(argArray is Array))
        throw new TypeError("argument array to 'apply' must be Array");
    return magic::apply(fn, thisArg, argArray);
}
```

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

4.3.3 Function.call

1 The static `call` method takes one or more arguments, *thisArg* and (optionally) *arg1*, *arg2* etc, and performs a function call using the `[[Call]]` property of the object. If the object does not have a `[[Call]]` property, a TypeError exception is thrown. The called function is passed *arg1*, *arg2*, etc. as the arguments.

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

4.4 Methods on Function instances

4.4.1 toString ( )

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

FIXME It doesn't make sense for (function () {}).toString() to return something that looks like a *FunctionDeclaration*, since the function has no name.

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

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

4.4.2 apply ( )

1 The intrinsic `apply` method calls the static `Function.apply` method with the `this` object as the first argument:

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

4.4.3 call ( )

1 The intrinsic `call` method calls the static `Function.apply` method with the `this` object as the first argument and the rest arguments as the arguments array:

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

4.4.4 [[HasInstance]] (V)

1 When the `[[HasInstance]]` method of a Function object is called with value V, the following steps are taken:

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

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

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

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

4.5 Properties of the Function Prototype Object

1 The `Function` prototype object is itself a `Function` object (its `[[Class]]` is `"Function"`) that, when invoked, accepts any arguments and returns **undefined**:

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

2 The value of the internal `[[Prototype]]` property of the Function prototype object is the Object prototype object (section Object.prototype).

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

4.5.1 Function.prototype.constructor

1 The initial value of `Function.prototype.constructor` is the built-in `Function` class object.

4.5.2 Methods on the Function Prototype Object

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

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

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

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

4.6 Value Properties of Function Instances

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

4.6.1 length

1 The value of the `length` property is usually 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.

4.6.2 prototype

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

5 Array Objects

1 Array objects give special treatment to a certain class of property names. A property name *P* (in the form of a string value) is an array index if and only if `ToString(ToUint32(P))` is equal to *P* and `ToUint32(P)` is not equal to 2^{32}-1.

FIXME For ES4 we need a better definition of a property name than something based on a string, but for arrays it will do for now.

2 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.

3 The set of *array elements* held by an object are those properties of the object that are named by nonnegative unsigned integers 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.)

4 The Array class has the following interface:

```
dynamic class Array extends Object
{
    function Array(...args) …

    meta static function invoke(...items) …

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

    static const length: uint = 1
    static const prototype: Object = …

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

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

5.1 The Array Constructor Called as a Function

1 When `Array` 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.

5.1.1 Array (...items)

1 When the `Array` class is invoked as a function the following steps are taken:

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

5.2 The Array Constructor

When `Array` is called as part of a new expression, it is a constructor: it initialises the newly created object.

5.2.1 new Array (...items)

1 The `[[Prototype]]` property of the newly constructed object is set to the original Array prototype object, the one that is the initial value of `Array.prototype` (section Array.prototype).

2 The `[[Class]]` property of the newly constructed object is set to `"Array"`.

3 The newly constructed object is initialized with a `length` property and array elements in the following manner:

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

5.3 Methods on the Array Class

1 The Array class provides a number of methods for manipulating array elements. These 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 methods on the Array class are all new in 4th edition.

5.3.1 Array.concat (object, ...items)

1 When the `concat` method is called with an *object* and zero or more additional arguments *items*, it returns a new Array containing the array elements of *object* followed by the array elements of each *item* in order.

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

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

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

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

    return out;
}
```

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

5.3.2 Array.every (object, checker, thisObj=null)

1 When the `every` method is called with an *object*, a function *checker*, and optionally an object *thisObj*, it calls *checker* on every array element in *object* in increasing numerical index order, returning **false** as soon as *checker* returns a false value, otherwise returning **true** if all the calls return true values.

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.

```
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;
}
```

5.3.3 Array.filter (object, checker, thisObj=null)

1 When the `filter` method is called with an *object*, a function *checker*, and optionally an object *thisObj*, it calls *checker* on every array element in *object* in increasing numerical index order, collecting all the 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.

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

```
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;
}
```

5.3.4 Array.forEach (object, eacher, thisObj=null)

1 When the `forEach` method is called with an *object*, a function *eacher*, and optionally an object *thisObj*, it calls *eacher* on every array element in *object* in increasing numerical index order. The return value of *eacher*, if any, is discarded.

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.

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

```
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);
}
```

5.3.5 Array.indexOf (object, value, from=0)

1 When the `indexOf` method is called with an *object*, a *value*, and optionally a starting index *from*, it compares the *value* to each array element in *object* in increasing numerical index order, returning the array index the first time *value* is equal to an element.

2 *From* provides the starting index value; it is rounded toward zero before use. If *from* is negative, it is treated as *`object`*`.length+`*`from`*.

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

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

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

5.3.6 Array.join (object, separator=",")

1 The array elements of *object* are converted to strings, and these strings are then concatenated, separated by occurrences of *separator*.

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

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

    return out;
}
```

5.3.7 Array.lastIndexOf (object, value, from=NaN)

1 When the `lastIndexOf` method is called with an *object*, a *value*, and optionally a starting index *from*, it compares the *value* to each array element in *object* in decreasing numerical index order, returning the array index the first time *value* is equal to an element.

2 *From* provides the starting index value; it is rounded toward zero before use. If *from* is negative, it is treated as *`object`*`.length+`*`from`*.

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

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

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

5.3.8 Array.map (object, mapper [, thisObj])

1 When the `map` method is called with an *object*, a function *mapper*, and optionally an object *thisObj*, it calls *mapper* on every array element in *object* in increasing numerical index order.

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 (defaulting to the global object if it is **null**).

3 The `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]*.

```
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;
}
```

5.3.9 Array.pop (object)

1 The last array element of *object* is removed from *object* and returned.

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

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

5.3.10 Array.push (object, ...items)

1 The *items* are appended to the end of the array elements of *object*, in the order in which they appear. The new `length` propety of the object is returned as the result of the call.

2 When the `push` method is called with an *object* and zero or more arguments *item1*, *item2*, etc., the following steps are taken:

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

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

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

    object.length = len;
```

```
        return len;
    }
```

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

5.3.11 Array.reverse (object)

1 The array elements of *object* are rearranged so as to reverse their order. *Object* is returned as the result of the call.

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

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

        return object;
    }
```

5.3.12 Array.shift (object)

1 The array element called `0` in *object* is removed from *object* and returned.

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

        let x = object[0];

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

5.3.13 Array.slice (object, start, end)

1 The `slice` method takes three arguments, an *object*, *start*, and *end*, and returns an array containing the array elements of *object* from element *start* up to, but not including, element *end* (or through the end of the array elements if *end* is undefined). If *start* is negative, it is treated as `object.length+start`. If *end* is negative, it is treated as `object.length+end`. The following steps are taken:

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

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

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

        return out;
    }

    helper static function clamp(intValue:double, len:uint):uint
    {
        return (intValue < 0.0)
             ? (intValue + len < 0.0) ? 0 : uint(intValue + len)
            : (intValue > len) ? len : uint(intValue);
    }
```

5.3.14 Array.some (object, checker [, thisObj])

1 When the `some` method is called with an *object*, a function *checker*, and optionally an object *thisObj*, it calls *checker* on every array element in *object* in increasing numerical index order, returning **true** as soon as *checker* returns a true value, otherwise returning **false** if all the calls return false values.

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.

```
    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;
    }
```

5.3.15 Array.sort (object, comparefn)

1 The array elements of *object* are sorted. 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 $x > y$.

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

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

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

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

3 Otherwise the following steps are taken.

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

4 The returned object must have the following two properties.

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

5 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.

6 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.

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

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

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

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

    if (comparefn === undefined) {
        x = x.toString();
        y = y.toString();
        if (x < y) return -1;
        if (x > y) return 1;
        return 0;
    }
    return comparefn(x, y);
}
```

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

5.3.16 Array.splice (object, start, deleteCount, ...items)

1 When the `splice` method is called with three or more arguments *object*, *start*, *deleteCount* and (optionally) some *items*, the *deleteCount* array elements of the object starting at array index *start* are replaced by the *items*. The following steps are taken:

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

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

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

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

    let end = start + deleteCount;
```

```

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

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

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

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

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

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

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

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

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

5.3.17 Array.unshift (object, ...items)

1 The *items* are inserted as new array elements at the start of the *object*, such that their order within the array elements of *object* is the same as the order in which they appear in the argument list.

2 When the unshift method is called with an object and zero or more *items*, the following steps are taken:

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

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

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

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

        object.length = len+numitems;

        return len+numitems;
    }
```

5.4 Properties of the Array Class

1 The value of the internal `[[Prototype]]` property of the Array constructor is the Function prototype object (section Function.prototype).

2 Besides the internal properties and the `length` property (whose value is 1), the Array constructor has the following properties:

5.4.1 Array.prototype

1 The initial value of `Array.prototype` is the Array prototype object (section Array.prototype).

5.5 Properties of the Array Prototype Object

1 The value of the internal `[[Prototype]]` property of the Array prototype object is the Object prototype object (section Object.prototype).

2 The Array prototype object is itself an array; its `[[Class]]` is `"Array"`, and it has a `length` property (whose initial value is +0) and the special internal `[[Put]]` method described in section Array.`[[Put]]`.

> **NOTE** The Array prototype object does not have a `valueOf` property of its own; however, it inherits the `valueOf` property from the Object prototype Object.

5.5.1 Array.prototype.constructor

1 The initial value of `Array.prototype.constructor` is the built-in Array constructor.

5.5.2 Array.prototype.toString ( )

1 The result of calling this function is the same as if the built-in `join` method were invoked for this object with no argument.

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

5.5.3 Array.prototype.toLocaleString ( )

1 The elements of this Array are converted to strings using their `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.

2 The result is calculated as follows:

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

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

5.5.4 Generic Methods on the Array Prototype Object

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

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

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

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

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

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

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

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

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

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

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

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

```

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

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

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

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

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

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

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

5.6 Properties of Array Instances

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

5.6.1 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.

5.7 Method Properties of Array Instances

5.7.1 Intrinsic methods

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

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

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

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

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

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

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

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

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

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

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

```

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

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

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

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

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

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

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

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

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

5.7.2 [[Put]] (P, V)

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

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

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

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

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