The Ibex Reference

Nitrogen Release

Adam Megacz
adam@ibex.org

March 22, 2004

Contents

1 Preface

If you are reading the html version of this document and are thinking of printing it out, you might be interested in the nicely typeset pdf version produced with LaTeX.

This document is a **reference**. It is not a **specification** or a **tutorial**.

This document does not guide the user gently through examples (as a tutorial would), and it doesn't provide enough detail and formality for a third party to construct a compatible re-implementation of the Ibex Core (as a specification would).

Rather, the goal of this document is to completely describe every aspect of the environment that the Ibex Core provides to client applications, from the bottom up. If you want to be an Ibex expert, this is the right document to read. It is assumed that you are already familiar with XML and with either JavaScript or ECMAscript. If you are not familiar with ECMAscript, some reference materials are provided in

The *shoehorn sequence* (how the Ibex Core gets onto the client's computer, and how it knows where to download the initial .ibex from) is not described in this document, since it will be different for every platform that Ibex is ported to.

If you need to use or rely on some behavior you notice in the Ibex Core, but which is not clearly defined here, please post to the users mailing list.

2 Key Concepts

2.1 Definitions

Ibex itself; the native code (or Java bytecode) that runs on the client. This term does not include the *shoehorn or the UI*

a set of files (mostly XML, JavaScript, and PNG images) bundled up in a zip archive, ending with the "..ibex" extension. Together, these files specify the appearance and behavior of the application's user interface. Sometimes we'll refer to this as the ".ibex" to be clear that we're talking about the actual zip archive, rather than its visual appearance when rendered on the screen.

We will use the term "the server" to refer to any other computer which the client makes XML-RPC or SOAP calls to. Note that it is possible for the client and server to be the same machine.

this is a very small piece of code that is downloaded the first time a client uses Ibex. It downloads the Ibex core, verifies its signature, and launches it with the appropriate parameters indicating where to find the initial UI. The Shoehorn works differently on every platform, and is outside the scope of this document.

In ECMAscript, when you change the value of a property on an object, you are *putting to that property, or writing to it. For example, the ECMAscript expression "*`foo.bar = 5`*" puts the value 5 to the bar property on object foo.*

In ECMAscript, when you access the value of a property on an object, you are *getting that property, or reading from it. For example, the ECMAscript expression "*`return (3 + foo.bar)`*" gets the value of bar property on object foo and then adds 3 to it before returning the result.*

We will use the terms JavaScript and ECMAScript interchangeably in this document. The Ibex interpreter is not completely ECMA-compliant, however (see for details).

2.2 Surfaces

Each top-level window in an Ibex UI is called a *surface*. There are two kinds of surfaces: *frames*, which usually have a platform-specific titlebar and border, and *windows*, which never have any additional platform-specific decorations.

Whenever we refer to the size or position of a surface, we are referring to the size or position of the UI-accessible portion of the surface; this does not include any platform-specific decorations. This means that if you set the position of a frame to (0,0), the platform-specific titlebar will actually be off the screen on most platforms (it will be above and to the left of the top-left corner of the screen).

Surfaces are not actual JavaScript objects; you cannot obtain a reference to a surface. However, each surface is uniquely identified by its *root box*, described in the next section.

2.3 Boxes

A *box* is the fundamental unit from which all Ibex user interfaces are built. Boxes can contain other boxes (known as *children*). Each Surface has a single box associated with it called the *root box*; the root box and its children (and its children's children, and so on) form the surface's *box tree*.

There are three ways to think of a box: as a rendered visualization on the screen (the "Visual Representation"), as a JavaScript object (the "Object Representation"), and as an XML tag (the "XML Representation").

FIXME: diagram here

All three representations are equally valid, and being able to figure out what an action in one representation would mean in terms of the other two representations is crucial to a solid understanding of Ibex.

2.4 The Object Representation

Each box is a full-fledged ECMAscript object, and can store key-value pairs as properties. Some of these keys have special meaning, which will be explained later. Each box's numeric properties hold its *child boxes*.

2.5 The Visual Representation

Each box occupies a rectangular region on the surface. The visual appearance of a surface is created by rendering each box in its tree. Unless the `clip` attribute is `false`, each box will clip its childrens' visual representations to its own, so that the children appear "confined to" the parent. Children are rendered after their parents so they appear "on top of" their parents (they obscure them).

Each box has two major visual components, each with subcomponents:

FIXME: diagram

- A **path**, which consists of zero or more lines and curves. The path may be filled with a color, gradient, or texture, and may be stroked with a line of a given thickness and color. If the path is not specified, it defaults to the perimiter of the box.
  - The path has an associated **strokecolor**, which is a color
  - The path has an associated **strokewidth**, which is a number specifying the width of the stroke.
  - The path also has a **fill**, which is either a color, gradient, or texture
  - A single line of **text**, which can be rendered in different fonts, colors, and sizes.

- The text has an associated **font**, which currently can be any font supported by the FreeType2 library.
- The text also has an associated **fontsize**
- The text is drawn in an associated **textcolor**

These eight components plus the size of a box fully specify its appearance. Every single box you see in Ibex is drawn only on the basis of these components and its size.

2.6 The XML Representation

A template (discussed in the next section) is an XML file which acts as a blueprint for constructing a tree of boxes. We call this construction process *applying*, since unlike *instantiation*, you always apply a template to a pre-existing box, and you can apply multiple templates to the same box. Each XML tag corresponds to a single box, or to another template which will be applied to that box. For example, a `scrollbar` template, when applied, will construct a tree of boxes which has the visual appearance and behavior of a scrollbar.

Although it is useful to think of the XML tags as being boxes, keep in mind that the XML representation is only a blueprint for constructing a tree of JavaScript objects. Once the template has been instantiated, the XML is effectively "thrown away", and JavaScript code is free to alter the boxes.

2.7 Templates

Each template is an XML document whose root element is `<ibex>`. Any text content of the root element is ignored, and may safely be used for comments. The root element may have any of the following elements as children, each of which may appear no more than once, and which must appear in this order:

Here is a sample Ibex file:

```
<ibex>
    This is a sample Ibex file. Text up here is ignored.
    Copyright (C) 2004 Mustapha Mond.
    <static>
        // code here will be executed only once
    </static>
    <template>
        <box></box>
        <checkbox></checkbox>
        <box>
            /* This has to be commented out or else it
               will be treated as a script */
            <lib:scrollbar></scrollbar>
        </box>
    </template>
</ibex>
```

2.8 Applying an XML tag to a box

The following description of the box application is extremely detailed and precise; it is intended for UI designers who need to know the exact order in which each event happens. FIXME: easier description. While this whole process sounds very complex, it actually works pretty intuitively. The description below is given in great detail since most applications will wind up being unintentionally dependent on subtle features of this process. However, most of the time you can just pretend that the XML tags and the boxes are the same thing.

To apply an XML tag **X** to a box **B**, perform the following operations, in this order:

- Allocate a fresh scope **s** whose parent scope is **B**.
- Process each child element or text segment of **X** in the order they appear in the document: For each *text segment***t**:
  - Treat **t** a JavaScript script, and execute it with **s** as the root scope.
- For each *child element***x** of **X**:
  - Create a new box **b**.
  - If the name of tag **x** is not "`box`" (in the default XML namespace), prepend the tag's namespace identifier uri (if any) to the name of the tag, and use the result as a key to retrieve a property from the root stream (defined later). Interpret the resulting stream as a template and apply that template to **b**.
  - (recursively) apply **x** to **b**.
  - If **x** has an `id` attribute, declare a variable in **s** whose name is the value of the `id` attribute, prefixed with the `$` character, and whose value is **b**
  - Copy any `$` -variables created during the application of **x** into scope **s**.
  - Append **b** as the last child of **B**.
- Apply any attributes on **X** to **B**, except for `id`. Since XML specifies that the order of attributes cannot be significant, Ibex processes attributes in alphabetical order by attribute name. For example, if **X** has the attribute `foo=''bar"`, then the equivalent of the statement `B.foo=''bar'';` will be performed, with the following exceptions:
  - If the value portion of the attribute is the string "`true`", put the boolean `true`. If the value is "`false`", put the boolean `false`.
  - If the value is a valid ECMAscript number, put it as a number (instead of a string).
  - If the value begins with a dollar sign (`$`), retrieve the value of the corresponding variable in **s** and use that value instead.
  - If the value begins with a dot (`.`), prepend the attributes' namespace identifier uri (if any) and interpret the remainder as a property to be retrieved from the root stream (defined later).

The last two steps are referred to as the *initialization* of the node. There are two important aspects of this ordering to be aware of:

- A given box will be fully initialized before its parent is given a reference to that box. This way, parents can be certain that they will never wind up accessing a box when it is in a partially-initialized state.
- Attributes are applied *after* scripts are run so that the attributes will trigger any *traps* (defined later) placed by the script.

2.9 Life Cycle of an Ibex Application

A user begins by specifying the URL of an Ibex application run. Usually this is done by visiting a web page which uses the *shoehorn* to install the core if it is not already on the user's machine, but you can also supply the URL on the command line.

The Ibex Core downloads the .ibex for the application, loads it, applies the `main.ibex` template and renders it onto the screen, running any associated ECMAscript code.

The user interacts with the application by clicking and moving the mouse, and by pressing keys on the keyboard. These actions trigger fragments of JavaScript code which are designated to handle events. This JavaScript code can then relay important information back to the server using XML-RPC or SOAP, or it can modify the structure and properties of the user interface to change its appearance, thereby giving feedback to the user.

DIAGRAM: graphic here showing the circular feedback cycle.

The Ibex core quits when the last remaining surface has been destroyed.

3 Layout and Rendering

The size and position of every other box is determined by its properties, its childrens' sizes, and its parent's size and position. Box layout and rendering happens in four phases: *packing*, *constraining*, *placing*, and *rendering*. The Core is careful to only perform a phase on a box if the box has changed in a way that invalidates the work done the last time that phase was performed. The packing and constraining phases are performed in a single traversal of the tree (packing is preorder, constraining is postorder), and the placing and rendering phases are performed in a second traversal of the tree (first placing, then rendering, both preorder).

For brevity, the rest of this chapter deals only with width and columns. Height and rows is treated identically and independently. Also, it is important to note that the term *minimum width* is not the same thing as the property `minwidth`, although they are closely related.

3.1 The size of the root box

When the user resizes a window, Ibex changes the root box's `maxwidth` and `maxheight` to match the size chosen by the user and then determines the root box's size using the same sizing rules it uses for other boxes.

Ibex will always attempt to prevent the user from making the surface smaller than the root box's `minwidth` and `minheight`. If the `hshrink` or `vshrink` flag is set, Ibex will try to prevent the user from resizing the surface at all. However, not all platforms give Ibex enough control to do this.

3.2 The alignment point

When talking about positioning, we will often refer to the *alignment point*.

- If the `align` property is "`center`", then the alignment point is the center of the box.
- If the `align` property is "`topleft`", "`bottomleft`", "`topright`", or "`bottomright`", then the alignment point is corresponding corner of the box.
- If the `align` property is "`top`", "`bottom`", "`right`", or "`left`", then the alignment point is middle of the corresponding edge of the box.

FIXME: diagram

When positioning a child box, the alignment point is determined by the *parent's* `align` property. When positioning a visual element (a texture, path, or text string) within a box, the alignment point is determined by the *box's own* `align` property.

A simple way to think about this is that whenever there are two boxes involved in the decision, you should use the parent's alignment point.

3.3 Packing

of *cells* is created within the parent. If the parent's `cols` property is set to 0, the cell grid has an infinite number of columns. Either `cols` or `rows` must be zero, but not both.

If a child's `visible` property is `false`, it does not occupy any cells (and is not rendered). Otherwise, each child occupies a rectangular set of cells `child.colspan` cells wide and `child.rowspan` cells high.

The Core iterates over the cells in the grid in the following order: if `rows` is 0, the Core iterates across each column before proceeding to the next row; otherwise rows come before columns. At each cell, the Core attempts to place the *first remaining unplaced child's* top-left corner in that cell (with the child occupying some set of cells extending down and to the right of that cell). If the parent has a fixed number of columns and the child's `colspan` exceeds that limit, the child is placed in column zero regardless, but only occupies the available set of cells (it does not "hang off the end" of the box).

```
<box>
    <box></box>
    <box></box>
    <box></box>
    <box></box>
    <box></box>
</box>
```

Notes on the layout example:

- Box '3' doesn't fit in the gap after '2', nor in the gaps either side of '2' on the next row, hence it is pushed onto the 3rd row.
- Box '4' would fit in the gaps around '2', but must be placed *after* it's preceeding box, '3'.

3.4 Constraining

- Each box's minimum width is computed recursively as the maximum of:
  - Its `minwidth`
  - The width of the box's `text` (after applying the box's `transform`).
  - The width of the box's path (after applying the box's `transform`) *if the box is* `packed`.
  - The width of the bounding box enclosing the box's cells.
- The minimum width of each cell is computed as the minimum width of the box occupying it divided by the box's `colspan`.

- If a box's `hshrink` property is set to `true`, the box's maximum width is the same as its minimum width; otherwise it is the box's `maxwidth`.
- The maximum width of each cell is the `maxwidth` of the box occupying it divided by the box's `colspan`.

3.5 Placing

- Each column's *actual width* is set to the maximum *minimum width* of all the cells in that column. **NOTE:** although a column or row can be sized smaller than its "minimum width" or larger than its "maximum width", a box will *never* be smaller than its `minwidth` or larger than its `maxwidth`.
- Each column's maximum width is the largest maximum width of the cells in that column, but no smaller than the column's minimum width.
- The *slack* is the difference between the parent's width and the sum of its columns' actual width. The slack is divided equally among the columns. Any column which has exceeded its maximum width is set to its maximum width, and the difference is returned to the slack. This process is repeated until the slack is zero or all columns are at their maximum width.
- Next, the rows and columns are positioned within the parent box. The rows and columns are transformed according to the parent's `transform` property, and the bounding box of the resulting cells are placed such that the cells' alignment point coincides with the parent's alignment point (both alignment points are determined by the parent's `align` property). FIXME: diagram
- **Packed boxes:** Each packed box's actual position and size is then set to the aggregation of the actual sizes of the cells it spans. If this size exceeds the box's maximum width, the box is sized to its maximum width and centered horizontally within the space occupied by its cells. **Non-packed boxes**: each non-packed box is transformed according to the parent's `transform` property and then positioned so that its alignment point is `(child.x, child.y)` pixels from the parent's alignment point (both alignment points are determined by the parent's `align` property).

3.6 Rendering

Boxes are rendered in a depth-first, pre-order traversal. Note that this may cause a non-packed box to overlap its siblings.

- If the box's `transform` property is non-null, the coordinate space is transformed accordingly for the rest of this phase and for the rendering of all children.
- If the box is packed and has a non-`null` path, the path is translated such that the alignment point of the path's bounding box coincides with the box's alignment point (both alignment points are determined by the box's `align` property).
- If a box has a path, that path is filled with the color, gradient, or image specified by the `fill` property and stroked with the color and width specified by the `strokecolor` and `strokewidth` properties.

- If the box has a non-`nulltext` attribute, the text is rendered in `font` with size `fontsize` and color `textcolor`. The text is then translated such that the alignment point of the text's bounding box coincides with the box's alignment point (both alignment points are determined by the box's `align` property).
- The box's children are rendered (pre-prder traversal).

4 Box Properties

4.1 Rendering Properties

Every box has several special properties which control how it is drawn. In general, if you put an invalid value to a special property, no action will be taken – the put will be ignored.

If the value is a 5-character hex string (# `RGB), 7-character hex string (# RRGGBB), 9-character hex string (# AARRGGBB), the box's stroke color will be set to that color. If the value is one of the ICC colors (the same set of color names supported by SVG), the stroke color be set to that color. If the value is null, the stroke color will be set to clear (# 00000000).`

The width (in pixels) to stroke the path with.

This property can be set to any of the values specified for `strokecolor. Alternatively, if the value written is an object, its stream will be read and interpreted as a PNG, GIF, or JPEG image, which will become the texture for this box, and the box's minwidth and minheight properties will be automatically set to the dimensions of the image.`

The box's path. The grammar and feature set supported are identical to that specified in SVG 1.1, section 8.

The box's text; writing `null to this property sets it to ''''.`

When an object is written to this property, its stream is read using the freetype2 library, and the resulting font is used to render the box's `text.`

The size (in points) to render the text.

The color in which to render the font; accepts the same values as `strokecolor.`

4.2 Layout Properties

If set to `true, this box will shrink (horizontally/vertically/both) to the smallest size allowed by its children and the bounding box of its path.`

If the box is a root box, this is the (x/y)-coordinate of the surface; otherwise it is the distance between the parent's alignment point and this box's alignment point.

The distance between this box's (left/top) edge and the root box's (left/top) edge. A put to this property has the same effect as a put to the (`x/y) property, except` `that it is relative to the root box rather than to this box's parent. FIXME is this fakeable? How is distance measured?`

The desired minimum width and height.

The desired maximum width and height.

When read, this is the (width/height) of this box. Writing to this property is equivalent to writing to *both the minimum and maximum (width/height).*

The number of (columns/rows) in which to lay out the children of this box. If set to zero, the number of (columns/rows) is unconstrained. Either `rows or cols must be zero. If 0 is written to cols when rows is 0, the write is ignored. If a nonzero value is written to cols when rows is nonzero, rows is set to 0, and vice versa.`

The number of (columns/rows) that this box spans within its parent.

Determines the box's alignment point for positioning its text, texture, path, and children.

If set to false, this box will be rendered as if its width and height were zero. If this is a root box, the associated surface will be hidden. When reading from this property, the value `false will be returned if this box` *or any of its ancestors* `is not visible. Thus it is possible to write true to a box's visible property and then read back false.`

The layout strategy for this box.

4.3 Child Control Properties

During a box initialization, script-private references to a box's descendants with `id` attributes are placed on the box. These references allow scripts on that box to easily refer to descendant nodes created by the template in which the script appears. For example, these two blocks of code have exactly the same effect:

```
<box>                         <box>
    <box></box>               <box></box>
    $foo.color = "red";           var $foo = this[0];
                                  $foo.color = "red";
</box>                        </box>
```

The following special properties control how a box's children are laid out. If a box has a non-null redirect target, reads and writes to these properties will be forwarded to the redirect target.

The `redirect` attribute is very useful for hiding the internal structure of a widget, and for allowing widgets to act as "smart" containers for other widgets. For example, a menu widget might have an invisible child as its redirect target; this way, when boxes representing items on the menu are added as children of the menu widget, they do not appear until the menu is pulled down.

The *nth child of box* `b` *can be accessed by reading from* `b[n]`*. The nth child can be removed by writing* `null` *to* `b[n]` *(the child will become parentless). A new child can be inserted before the nth child by writing it to* `b[n]`*; if the value written is already a child of* `b`*, it will be removed from* `b` *first. It is important to note that this behavior is different from ECMAscript arrays – writing a non-*`null` *value to* `b[n]` *does not eliminate the nth child; it merely shifts it over one position.* **Note:** *Unlike most JavaScript objects, enumerating a Box's properties with the JavaScript* `for..in` *construct will enumerate only the box's children and not any other properties.*

If `true, the visual representation of this box's children will be clipped to the boundaries of this box.` **Note:** `setting this property to false imposes a substantial performance penalty.`

The number of children this box has.

Writing to this property sets the box's redirect target. This property cannot be read from, and can only be written to once.

If this box has a parent, this property returns *parent.surface*`; otherwise it returns null. This property is a simple building block that the widget library uses to implement more complex functionality such as focus control and popups.`

4.4 Other Box Properties

The shape that the cursor should take when inside this box. Valid values are: "`default'' , ''wait'', ''crosshair'', ''text'', ''hand'', and ''move'', as well as resizing cursors''east'', ''west'', ''north'', ''south'', ''northwest'', ''northeast'', ''southwest'', and ''southeast''. Note that on some platforms, resize cursors for opposite directions (such as northwest and southeast are the same). If a box's cursor is null, its parent's cursor will be used. If the root box's cursor is null, the ''default'' cursor will be used.`

The (horizontal/vertical) distance between the mouse cursor and this box's (left/top) edge. Puts to this property are ignored. This value will not be updated if the mouse is outside the root box of the surface and no button was pressed when it left.

True if the mouse is inside the rendered region of this box or any of its children. This value will be false if the mouse is inside a portion of this box which is covered up by one of this box's siblings, or one of its ancestors' descendants. Puts to this value are ignored.

Reading from this property will return the parent scope used to execute the `block of the template in which the currently-executing code resides.`

Returns a reference to the box itself. If `null is written to this property, and this box is the root box of a surface, the box will be detached and the surface destroyed. If this box has a parent, it will be detached from its parent.`

This property is actually a function; invoking `parent.indexof(child) will return the numerical index of child in parent if child` is a child of parent (or parent's redirect target), and -1 otherwise. Writing to this property has no effect.

These properties are meant to be trapped on FIXME defined later?. Placing a trap on childadded/childremoved lets a box receive notification when a child is added/removed. In either situation, the child will be passed as an argument to the trap function *after* the addition or removal has been performed.

Note that if the parent's redirect target is set to another box, these traps will only be invoked when children are manipulated by reading and writing to the parent. Reads and writes directly to the redirect target will *not* trigger the traps.

Note also that these traps are still triggered if a box's redirect target is *null*. This is useful for boxes that need to accept children and then relocate them elsewhere.

4.5 Notification Properties

The following properties are used to notify a box of changes specific to that particular box.

The value true is written to this property when the mouse enters the box.

The value true is written to this property when the mouse leaves the box.

The value true is put to this property after the size of this box changes.

4.6 Root Box Properties

The following special properties are only meaningful on the root box of a surface.

The value true is put to this property on the root box when the surface gains the input focus, and false when the surface loses the input focus. Reading from this value will return true if the surface is focused and false if it is not. Putting true to this property will *not* cause the surface to ''steal'' the input focus from other windows.

The value true is put to this property on the root box when the surface is maximized, and false when the surface is un-maximized. Reading from this value will return true if the surface is maximized and false if it is not. Putting true to this property will maximize the window, and putting false to this property will unmaximize the window. Note that not all platforms support maximization.

The value true is put to this property on the root box when the surface is minimized, and false when the surface is unminimized. Reading from this value will return true if the surface is minimized and false if it is not. Putting true to this property will minimize the window, and putting false will unminimize it.

When the user attempts to close a surface, the value true will be put to this property. Scripts may trap this property FIXME defined later? to prevent the window from closing. Putting the value true to this property on a root box has the same effect as putting null to the thisbox property.

The surface's icon. This is usually displayed on the titlebar of a window. The value should be the stream name of a PNG image. Note that not all platforms support this property.

The surface's titlebar text. Note that not all platforms support this property. Only ASCII characters 0x20-0x7F are permitted.

5 Streams

5.1 Every object has a stream...

Every object has a *stream* associated with it. A stream is a sequence of bytes that can be read or written to.

By default an object has an empty stream (zero bytes). However, some objects (returned from special methods on the `ibex` object) have streams yielding data read from an url, file, or a component of a zip archive. In a future release, the stream associated with a box will be an .ibex template which, when applied, will fully reconstitute the box's state.

5.2 ...but streams are not objects

Despite the ubiquity of streams, you cannot actually reference a stream, since it is not an object. Instead, you simply reference the object it belongs to. If you are familiar with Java, this is similar to how every Java object has a monitor associated with it, but you cannot directly manipulate the monitor (you can't pass around a reference to just the monitor).

In the rest of the section we will sometimes refer to "getting properties from a stream" or "passing a stream to a function"; this is just shorthand for saying to perform those actions on the object the stream belongs to.

5.3 Creating Streams from URLs

You can create a stream from a URL by calling

```
var r = ibex.stream.url("http://...");
```

This will return an object whose stream draws data from the specified URL. Streams are loaded lazily whenever possible.

5.4 Getting Substreams

Most stream objects let you access substreams using the usual JavaScript operators `[]` and `.`, as well as the `for..in` syntax.

```
// r1 and r2 are equivalent but not equal (!=)
var r1 = ibex.stream.url("http://www.ibex.org/foo/bar.html");
var r2 = ibex.stream.url("http://www.ibex.org/foo")["bar.html"];
```

5.5 The Root Stream

The empty-string property on the `ibex` object is called the *root stream*. You can access this object as `ibex..` or `ibex[‘’‘’]`. Additionally, any expression which starts with a dot is treated as property to be retrieved from the root stream. The following three expressions are equivalent:

```
ibex..foo
ibex[""].foo
.foo
```

5.6 Static Blocks

You can access variables within the static block of a template by appending a double period (`..`) and the variable name to the stream used to load that template:

```
        foo = 12;
...
// elsewhere
ibex.log.print(org.ibex.themes.monopoly.scrollbar..foo);   // prints "12"
```

5.7 Formatting Streams

If you attempt to send a stream as part of an XML-RPC call, the stream will be read in its entirity, Base64-encoded, and transmitted as a element.

Ibex supports two special URL protocols. The first is `data:`, which inteprets the rest of the URL as a Base64 encoded sequence of bytes to use as a source. The other is `utf8:` which interpretets the rest of the string as a Unicode character sequence to be UTF-8 encoded as a string of bytes to use as a source.

```
var r5 = ibex.stream.url("data:WFWE876WEh99sd76f");
var r6 = ibex.stream.url("utf8:this is a test");
```

You can read a UTF-8 encoded string from a stream like this:

```
var myString = ibex.stream.fromUTF(ibex.stream.url("utf8:this is a test"));
```

You can also parse XML from a stream using SAX like this:

```
ibex.stream.xml.sax(ibex.stream.url("http://foo.com/foo.xml"),
                   { beginElement : function(tagname, attributeKeyValuePairs) { ...
                     endElement   : function(tagname) { ... },
                     content      : function(contentString) { ... }
                     whitespace   : function(whitespaceString) { ... }
                   });
```

6 The Ibex object

The `ibex` object is present in the top-level scope of every script. It has the following properties:

6.1 General

reading from this property returns a new box

creates a clone of object

returns a blessed clone of stream

6.2 ECMA Library Objects

reading from this property returns a new date

this object contains the ECMA math functions

return a regexp object corresponding to string *s*

this object contains the ECMA string manipulation functions

6.3 Logging

log the debug message *m, optionally dumping object o*

log the informational message *m, optionally dumping object o*

log the warning message *m, optionally dumping object o*

log the error message *m, optionally dumping object o*

7 User Interface

opens a new browser window with URL *u*

true if the control key is depressed

true if the shift key is depressed

true if the alt key is depressed

the name of the "alt" key (usually either "alt", "meta", or "option")

the contents of the clipboard; can be read and written to

the maximum dimension of any UI element; usually 231, but may be smaller

the width of the screen, in pixels

the height of the screen, in pixels

either 0, 1, 2, or 3, indicating the mouse button currently being pressed

when a box is written to this property, it becomes the root box of a new window

when a box is written to this property, it becomes the root box of a new frame

an object whose stream is a a builtin serif font

an object whose stream is a builtin sans-serif font

an object whose stream is a a builtin fixed-width font

7.1 Networking

not yet implemented

return an XML-RPC call object with endpoint URL *u*

return a SOAP call object with endpoint URL *u, SoapAction a, and XML Namespace n*

7.2 Threads

when a function is written to this property, a new thread is forked to call it

yield the current thread

sleep for *n milliseconds*

7.3 Streams

returns a new object whose stream is drawn from URL *u*

unpacks a zip archive from *s's stream*

unpacks a cab archive from *s's stream*

valign=top¿wraps a disk-backed read cache keyed on *k around s's stream*

returns an object whose stream is drawn from *s's stream, but invokes f(n,d) as it is read from.*

Use SAX to parse the XML document on stream *s with handler h*

Same as `parse.xml(), but tries to fix broken HTML.`

treat *s's stream as a string encoded as a UTF-8 byte stream and return the string*

`ibex.stream.tempdir`

7.4 Cryptography

not implemented yet: return a stream which rsa-decrypts stream s with key k

not implemented yet: return a stream which rc4-decrypts stream s with key k

not implemented yet: immediately MD5-hash stream s

not implemented yet: immediately SHA1-hash stream s

8 Traps

8.1 Simple Traps

You can add a trap to a property by applying the `++=` operator to a function with one argument. The trap will be invoked whenever that property is written to.

```
<box>
    foo ++= function(z) {
       ibex.log.info("foo is " + z);
    }
</box>
```

If another script were to set the property “`foo`” on the box above to the value `5`, the function above would be invoked with the argument `5`. The function would then log the string “`foo is 5`“.

Within a trap, the expression `trapee` can be used to get a reference to the box on which the trap was placed.

The expression `trapname` returns the name of the trap executing the current function. This is useful when a function is applied to multiple traps. For example:

```
<box>
    func ++= function(z) {
        ibex.log.info("called trap " + trapname);
    }
    foo ++= func;
    bar ++= func;
</box>
```

8.2 Removing Traps

You can remove a trap by using the `--=` operator with the same function you added as a trap:

```
<box>
    var myfunc = function(z) { /* ... */ }
```

```
        // add the trap
        func ++= myfunc;
        // ...
        // remove the trap
        func --= myfunc;
    </box>
```

8.3 Multiple Traps on the Same Property

When the property is *written* to, each of the trap functions placed on it will be invoked in the opposite order that they were placed on the box – the most recently placed trap will execute first. This last-to-first execution of traps is called *cascading*. After the last trap is invoked, the value is stored on the box (remember, boxes are objects, so they can hold properties just like all other ECMAscript objects).

8.4 Manual Cascades

There are two additional tricks you can use when placing traps. The first is a *manual cascade*. If you want to cascade to lower traps in the middle of a function, or you want to cascade with a different value than the value passed to you (in effect "lying" to lower traps), you can use `cascade`. For example:

```
<box>
    color ++= function(c) {
        ibex.log.info("refusing to change colors!");
        cascade = "black";
    }
</box>
```

This effectively creates a box whose color cannot be changed, and which complains loudly if you try to do so.

Do *not* try to do something like this:

```
<box>
    color ++= function(z) {
        color = "black";      // INFINITE LOOP! BAD!!!
    }
</box>
```

To prevent automatic cascading, return `true` from your function:

```
<box>
    color ++= function(z) {
        return true;          // the box’s color will not change
    }
</box>
```

8.5 Read Traps

The other trick is a *read-trap*. Read traps are just like normal traps, except that you use a function that takes zero arguments instead of one. Read traps also do not automatically cascade.

```
<box>
    doublewidth <tt>++=</tt> function() { return 2 * width; }
</box>
```

If another script attempts to read from the `doublewidth` property on this box, the value it gets will be twice the actual width of the box. Note that the actual `doublewidth` property on the box never gets written to, since the trap does not cascade.

You can manually cascade on read traps as well:

```
<box>
    text <tt>++=</tt> function() { return "my text is " + cascade; }
</box>
```

Read traps are only rarely needed – most of the time a write trap should be enough.

8.6 Prohibited Traps

To prevent confusing and hard-to-debug behaviors, scripts may not place traps on any of the properties described in the sections , , or except for `childadded`, `childremoved` and `surface`. FIXME: remove?

8.7 Exceptions and Traps

If an uncaught exception is thrown from a trap, Ibex will log the exception, but will *not* propagate it to the code which triggered the trap. If the trap was a read trap, the value `null` will be returned. FIXME: is this right?

8.8 Architectural Significance of Traps

Traps are the backbone of Ibex. Since almost all UI programming is event/demand driven, traps eliminate the need for separate member/getter/setter declarations, often cutting the amount of typing you have to do to a third of what it would normally be.

8.9 Cloning

Cloning is a companion technique for traps; together they can be used to simulate any sort of environment you might need. When you call `ibex.clone(o)`, Ibex returns a new object (called the *clone*) which compares with equality (==) to the original object. Furthermore, both objects are "equal" as keys in hashtables, so:

```
var hash = {};
var theclone = ibex.clone(o);
hash[o] = 5;
ibex.log.info(hash[theclone]);    // prints "5"
```

Any writes to properties on the clone will actually write to properties on the original object, and reads from properties on the clone will read properties on the original object. In fact, the only thing that can be used to distinguish the original from the clone is traps – a trap placed on the clone is *not* placed on the original object as well.

8.10 Ibex self-emulation

When the core first starts up, it clones the `ibex` object, creates a stream for the initial .ibex, and then places a trap on the cloned `ibex` object so that its empty-string property returns the .ibex stream. The cloned Ibex object is then passed as the third (optional) argument to `ibex.apply()`, making it the default `ibex` object for the scripts that are executed as part of the template instantiation.

```
var new_ibex = ibex.clone(ibex);
var stream = ibex.bless(ibex.stream.url("http://..."));
new_ibex[""] ++= function() { return stream; }
ibex.apply(ibex.box, new_ibex..main, new_ibex);
```

Note that we called `ibex.bless()` on the stream before tacking it on to the new Ibex object. The bless function returns a clone of the object passed to it, with a few traps which are explained below. Additionally, any sub-streams retrieved by accessing properties of the blessed stream will also automatically be blessed (blessed streams are *monadic*).

Blessing a stream serves three purposes:

- Blessed clones always return the appropriate static block when their empty property is accessed; this ensures that references to the static blocks of other templates work properly.
- Blessed substreams can return their parent stream by accessing a hidden property which is reserved for internal use by Ibex. This ensures that Ibex can automatically add filename extensions where needed, according to the following rules:
  - If the stream is a template to be applied, the string "`.ibex`" is appended.
  - If the stream is an image, the string "`.png`" is appended. If no stream is found, "`.jpeg`" and "`.gif`" are tried, in that order.
  - If the stream is an font, the string "`.ttf`" is appended.
- Every call to `ibex.bless()` returns a different object (which happens to be a clone of the object passed to it) with a completely separate set of static blocks.

Ibex can self-emulate by using `ibex.clone()` on the Ibex object; this technique is very similar to the use of ClassLoaders in Java. This is useful for a number of applications, including debuggers, IDEs, sandboxing untrusted code, remote-control, and others. For example:

```
var newLoadFunction = function(url) { /* ... */ };
var new_ibex = ibex.clone(ibex);
new_ibex.load ++= function() { return newLoadFunction; }
ibex.apply(ibex.box, .main, new_ibex);
```

9 Contexts and Threading

9.1 Contexts

From the perspective of an application writer, Ibex is strictly single-threaded. Ibex is always in exactly one of the following three *contexts*:

- **Rendering Context**– (redrawing the screen)
- **Event Context** (executing javascript traps triggered by an event)
- **Thread Context** (executing a background thread spawned with `ibex.thread`)

There are two important restrictions on what can be done in particular contexts:

- The `box.mouse` property and its subproperties (`x`, `y`, and `inside`) can only be read from within the Event Context, or in a thread context *after* a the `box.mouse` property on this box or an ancestor box has been written to.
- Blocking operations (anything that accesses the network or disk) can only be performed in the Thread Context.

9.2 Background Threads

Ibex offers easy access to threads. Spawning a background thread is as simple as writing a function to the `ibex.thread` property:

```
ibex.thread = function() {
    ibex.log.info("this is happening in a background thread!");
}
```

The argument set passed to the function is currently undefined and is reserved for use in future versions of Ibex. Scripts should not depend on the number or content of these arguments.

Ibex is *cooperatively multitasked*, so threads must not process for too long. This was a deliberate choice; cooperatively multitasked environments do not require complex locking primitives like mutexes and semaphores which are difficult for novices to understand. The disadvantage of cooperative multitasking is that one thread can hog the CPU. This is unlikely to happen in Ibex for two reasons: first, all blocking I/O operations *automatically* yield the CPU, so the overall user interface never becomes unresponsive because it is waiting for a disk or network transfer. Second, since Ibex is strictly a user interface platform, Ibex scripts are unlikely to perform highly compute-intensive operations that keep the CPU busy for more than a few milliseconds.

9.3 Events

Every execution of the Event Context begins with an event, which consists of a key/value pair, and a mouse position, which consists of an x and y coordinate. The possible keys are `_Press[1-3]`, `_Release[1-3]`, `_Click[1-3]`, `_DoubleClick[1-3]`, `_Move`, `_KeyPressed`, and `_KeyReleased`.

Here are two example events:

An event is triggered by writing the key to the value on a box. This triggers any trap handlers which may be present. Once these handlers have executed, Ibex figures out which child of the current box contains the mouse (taking into account that some boxes may cover up others) and writes the key and value to that box. If none of the box's children contain the mouse position, Ibex removes the leading underscore from the key name and writes the value to *that* property. Once all the traps on that property have executed, the value is written to the box's parent.

Intuitively, Ibex delivers the underscored event to every box from the root to the target, and then delivers the non-underscored event to that same set of boxes in reverse order. So the event travels down the tree to the target, and then back up to the root. The following example prints out "first second third fourth" in that order.

```
<box>
    _Press1 ++= function(b) { ibex.log.info("first"); }
     Press1 ++= function(b) { ibex.log.info("fourth"); }
    <box>
      _Press1 ++= function(b) { ibex.log.info("second"); }
       Press1 ++= function(b) { ibex.log.info("third"); }
    </box>
</box>
```

In general, you should use the *non-underscore* names to respond to user input and use the underscored names when you want to override child boxes' behavior or route events to particular boxes (for example, when implementing a focus protocol). This is why the underscored elements are delivered to parents before children (so parents can override their childrens' behavior), but non-underscored events are delivered to children before parents (since, visually, a mouse click is usually "intended" for the leaf box underneath the cursor).

9.4 Stopping the Process

At any point in this sequence, a trap handler can choose not to cascade (by returning `true` from the trap handler function). This will immediately cease the propagation of the event. This is how you would indicate that an event has been "handled".

9.5 Re-routing events

At any point in the Event Context, you can write to the `mouse` property on any box. The value written should be an object with two properties, `x` and `y`. For example:

```
_Press1 ++= function(p) {
    mouse = { x: 32, y: 77 };
}
```

The coordinates specified are relative to the box whose `mouse` property is being written to. There is no need to supply the `inside` property; it is computed automatically. Writing to the `mouse` property causes Ibex to recompute the eventual target box, and also alter the values returned by `mouse.x`, `mouse.y`, and `mouse.inside` for any *descendants* of the current box. Writing to the `mouse` property also automatically prevents the event from returning to the box's parents – it is equivalent to not cascading on the non-underscored event. This ensures that child boxes cannot trick their parent boxes into thinking that the mouse has moved.

If you want the event to "skip over" the boxes between the trapee and the target, or if you want to re-route an event to a box which is not a descendant of the current box, simply write the value to the proper key on the target box.

```
<box>
    _KeyPressed = function(k) { ibex.log.info("first"); }
     KeyPressed = function(k) { ibex.log.info("sixth"); }
    $recipient.target = $target;
    <box>
        _KeyPressed = function(k) {
            ibex.log.info("second");
            thisbox.target.KeyPressed = k;
            // inhibit cascade to keep the event from going to $excluded
            return true;
        }
         KeyPressed = function(k) { ibex.log.info("fifth"); }
        <box>
            _KeyPressed = function(k) { ibex.log.info("this never happens"); }
        </box>
    </box>
    <box>
        _KeyPressed = function(k) { ibex.log.info("third"); }
         KeyPressed = function(k) { ibex.log.info("fourth"); }
    </box>
</box>
```

9.6 Synthesizing Your Own Events

You can create "fake events" by simply writing to the `mouse` property and then writing a value to one of the underscored properties on a box. This will have exactly the same effect as if the use had actually pressed a key, clicked a button, or moved the mouse – they are indistinguishable.

9.7 Enter and Leave

Ibex will trigger the `Enter` and `Leave` properties as it walks down the tree, based on the position of the mouse (or the faked position if the `mouse` property has been written to). However, `Enter` and `Leave` are not events since they do not implicitly cascade up or down the tree.

9.8 Detailed Description of Events

Indicates that the use has pressed a mouse button. On platforms with three mouse buttons, the *middle button is button 3 – this ensures that applications written to only use two buttons (1 and 2) will work intuitively on three button platforms.*

Indicates that the use has released a mouse button.

Indicates that the user has pressed and released the mouse button without moving the mouse much (exactly how much is platform-dependent).

Indicates that the user has clicked the mouse button twice within a short period of time (exactly how long is platform-dependent).

Indicates that the mouse has moved while within this box, or that the mouse while outside this box *if a button was pressed while within this box and has not yet been released*

A string is written to this property when a key is pressed or released If the key was any other key, a multi-character string describing the key will be put. For simplicity, we use the VK_ constants in the

Java 1.1 API java.awt.event.KeyEvent class. When a key is pressed or released, the string put will be the portion of its VK_ constant after the underscore, all in lower case. If the shift key was depressed immediately before the event took place, then the string will be capitalized. Special keynames are also capitalized; shift+home is reported as "`HOME''. Symbols are capitalized as they appear on the keyboard; for example, on an American QWERTY keyboard, shift+2 is reported as ''@''. If the alt, meta, or command key was depressed immediately before this key was pressed, then the string will be prefixed with the string ''A-''. If the control key was depressed while this key was pressed, then the string will be prefixed with the string ''C-''. If both alt and control are depressed, the string is prefixed with ''C-A-''. Ibex does not distinguish between a key press resulting from the user physically pushing down a key, and a`

’key press’ resulting from the keyboard’s typematic repeat. In the rare case that an application needs to distinguish between these two events, it should watch for KeyReleased messages and maintain an internal key-state vector.

10 Networking

10.1 XML-RPC

XML-RPC objects can be created by calling `ibex.net.rpc.xml(`*XML-RPC URL*`)`, and then invoking methods on that object. For example,

```
        Press1 += function(v) {
            ibex.thread = function() {
                color = ibex.net.rpc.xml("http://xmlrpc.ibex.org/RPC2/").color.getTodaysC
            }
        }
```

When the user clicks the first mouse button on this box, it will contact the server `xmlrpc.ibex.org`, route to the `/RPC2/` handler and invoke the `getTodaysColor()` method on the `color` object with a single string argument "`Friday`". The return value will be used to change the color of the box the user clicked on.

Note that in this example we spawned a background thread to handle the request – the `Press1` event is delivered in the foreground thread, and XML-RPC methods may only be invoked in background threads. This is to prevent the UI from "locking up" if the server takes a long time to reply.

If the XML-RPC method faults, an object will be thrown with two properties: `faultCode` and `faultString`, as defined in the XML-RPC specification. If Ibex encounters a network, transport, or session-layer error, it will throw a `String` object describing the error in a human-readable format. Scripts should not rely on the contents of this string having any special structure or significance.

If an object with an associated non-empty stream is passed as an argument to an XML-RPC method, it will be sent as a element. If a element is found in the XML-RPC reply, it will be returned as an object with a stream drawn from that byte sequence.

Each object returned by `ibex.net.rpc.xml()` represents a single HTTP connection. The connection will be held open until the object is garbage collected or the server closes the connection. If a second call is issued on the object before the first one returns (usually from a seperate thread), the two calls will be

pipelined. This can dramatically improve performance.

Ibex supports HTTP Basic and Digest authentication. To use authentication, pass `ibex.net.rpc.xml()` a URL in the form `http[s]://user:password@hostname/`. Ibex will use Digest authentication if the server supports it; otherwise it will use Basic authentication. Please be aware that many XML-RPC server implementations contain a broken implementation of Basic authentication.

10.2 SOAP

SOAP methods are invoked the same way as XML-RPC methods, but with three differences:

- `ibex.net.rpc.soap()`is used instead of `ibex.net.rpc.xml()`
- Instead of specifying just the URL of the service itself, you must specify the URL, the SOAPAction argument, and the namespace to use.
- The actual method invocation takes only one argument, which must be an object. This is necessary since SOAP arguments are specified by name, rather than ordering.

SOAP faults are handled the same way as XML-RPC faults except that the capitalization of the `faultstring` and `faultcode` members is all lower-case, to match the SOAP spec. Here is a SOAP example:

```
    Press1 ++= function(v) {
        ibex.thread = function() {
            color = ibex.net.rpc.soap("http://soap.ibex.org/SOAP",   // endpoint
                                      "GETTODAYSCOLOR",             // SOAPAction head
                                      "http://ibex.org/namespace"    // namespace for
                                 ).color.getTodaysColor( {
                                     whichday : Friday
                                 } );
        }
    }
```

As you can see, SOAP is much more verbose, yet does not offer substantially improved functionality. We recommend that XML-RPC be used whenever possible, and that SOAP be reserved for legacy applications.

The current Ibex SOAP stack does not support ’document style’ or multi-ref (`href`) data structures.

10.3 Security

Applications downloaded from the network (as opposed to those loaded from the filesystem) may only make certain kinds of connections to certain hosts. See Appendix A for a detailed description of the security policy.

11 Error Handling

If the Ibex Core encounters an error while servicing a function call originating in JavaScript, the core will throw a string consisting of an error code followed by a colon, a space, and a descriptive message. For example:

```
"ibex.net.dns.unknownhostexception: unable to resolve host foo.com"
```

The code should be used to determine how the program should respond to an error. The codes are organized in a hierarchy, so the string.startsWith() method can be used to determine if an error lies within a particular subhierarchy. The descriptive message portion of the string may be shown to the user.

an assertion failed

General I/O exceptions

Error translating between character encodings.

Attempted to access a corrupt zip archive.

End of file encountered unexpectedly

A piece of untrusted Ibex code attempted to contact a restricted host. See for details.

An attempt to resolve a hostname failed but it is not known for certain that the hostname is invalid.

An attempt to resolve a hostname failed because the hostname was invalid.

A socket was closed unexpectedly.

A connection could not be made to the remote host.

Tried to parse a malformed URL.

General SSL protocol errors.

The server's certificate was not signed by a CA trusted by Ibex.

Thrown when an HTTP error code is returned during an operation. The three characters *xyz* `will be the three-digit HTTP status code.`

The caller attempted to transmit the `null value via XML-RPC.`

The caller attempted to transmit a circular data structure via XML-RPC.

The caller attempted to transmit a "special" object via XML-RPC (for example, a Box or the Ibex object).

A JavaScript attempted to put to a property on the `null value`

A JavaScript attempted to get from a property on the `null value`

A JavaScript attempted to call the `null value`

If an exception is thrown inside a trap, the exception will propagate to the script that triggered the trap.

If an uncaught exception is thrown while applying a template, or the requested template could not be found, an error will be logged and the box to which the template was being applied will be made invisible (`visible = false`). This ensures that half-applied widgets are never shown to the user.

Security Architecture and Considerations

Due to the expense and hassle imposed by the commercial PKI code signing architecture, and the fact that it doesn't really provide any security anyways, Ibex user interfaces are distributed as unsigned, untrusted code. As such, they are handled very carefully by the Ibex Core, and assumed to be potentially malicious.

Ibex's security architecture is divided into defenses against four major classes of attacks:

.1 Malicious UI attempts to acquire or alter data on the client

Ibex user interfaces are run in an extremely restrictive sandbox. The environment does not provide primitives for accessing any data outside the Ibex core except via XML-RPC and SOAP calls. There are no facilities to allow Ibex user interfaces to access the client's operating system or to interact with other applications on the same host (unless they provide a public XML-RPC or SOAP interface). An Ibex script may only access a file on the user's hard disk if the user explicitly chooses that file from an "open file" or "save file" dialog. There is one exception to this rule: if all templates currently loaded in the Ibex core originated from the local filesystem, those templates can load additional .ibexs from the local filesystem.

The Ibex Core is written in Java, so it is not possible for scripts to perform buffer overflow attacks against the core itself.

Ibex applications may only read from the clipboard when the user middle-clicks (X11 paste), presses control-V (Windows paste), or presses alt-V (Macintosh paste; the command key is mapped to Ibex "alt"). This ensures that Ibex applications are only granted access to data that other applications have placed on the clipboard when the user specifically indicates that that information should be made available to the Ibex application.

.2 Malicious UI attempts to use client to circumvent firewalls

Ibex user interfaces may only make XML-RPC or SOAP calls and load .ibex archives via HTTP; they cannot execute arbitrary HTTP GET's or open regular TCP sockets.

Ibex will not allow a script to connect to a non-public IP address (10.0.0.0/8, 172.16.0.0/12, or 192.168.0.0/16, as specified in RFC 1918). There is one exception – if all templates currently loaded in the core originated from the same IP address, those scripts may make calls to that IP address regardless of whether or not it is firewalled. If Ibex does not have access to a DNS resolver (because it is using a proxy which performs DNS lookups), Ibex will provide the proxy with the appropriate

X-RequestOrigin header that the proxy needs in order to maintain security.

The only remaining possible attack is against a XML-RPC or SOAP service running on a firewalled host with a public address. Assigning such machines public IP addresses is a poor network security policy, and doing so squanders scarce public IPv4 addresses. As such, the onus is on the administrators of such machines to explicitly block access to clients reporting a `User-Agent:` header beginning with the three characters "`Ibex`".

.3 Malicious UI attempts to trick user into divulging secret information

All top-level windows created by Ibex are *scarred* – a stripe and a lock is drawn across the corner of the window. There is no way for a user interface to remove this scar. Ibex user interfaces may not create windows smaller than the size of the scar.

.4 Malicious network attempts to snoop or man-in-the-middle transactions

Ibex user interfaces may transmit network data using HTTPS (SSL 3.0) for encryption. Ibex will attempt 128-bit encryption, but will negotiate down to 40-bit if the server does not support strong crypto. Ibex's SSL implementation is currently provided by TinySSL and The Legion of the Bouncy Castle.

All HTTPS connections must be authenticated by the server using a certificate whose name matches the domain name of the HTTPS URL. The certificate must be signed by a trusted root CA. Ibex trusts the same 93 root CAs whose certificates are included as "trusted" in Microsoft Internet Explorer 5.5 SP2. This provides a minimal level of protection against man-in-the-middle attacks; you should not trust this connection with any data you would not normally trust an SSL-enabled web browser with.

ECMAscript compliance

Ibex's scripts are written in a modified subset of ECMA-262, revision 3 (aka JavaScript 1.5). The interpreter strays from the spec in a few ways.

.1 Omissions

The following ECMA features are not supported:

- The `undefined` value, `===`, and `!==`
- The `new` keyword (and ECMAScript object inheritance) `eval`
- `getter` and `setter`
- The ECMA `this` keyword.
- The `String`, `Number`, and `Boolean` classes. Note that `string`, `number`, and `boolean` values are supported, however.
- You may not `throw` the `null` value.

Additionally, you must declare all root-scope variables (with `var`) before using them; failure to do so will result in an exception. Box properties are pre-defined in the scope that scripts are executed in.

A Extensions

- The token `..` is equivalent to `[‘‘‘‘]`.
- Trapping
- Cloning
- Extended `catch` syntax. The following code:

  ```
  } catch(e propname "foo.bar.baz") {
     // ...
  }
  ```

  Is equivalent to:

  ```
  } catch(e) {
     if (e.propname != null && e.propname >= "foo.bar.baz" && e.propname < "foo
         // ...
     }
  }
  ```

  Multiple extended-catch blocks can appear at the end of a single try block. However, at most one non-extended catch block may appear, and if it does appear, it must be the last one.
- Since Ibex ECMAscripts are wrapped in XML, the lexical token “`lt`” is be interpreted as `<`, the lexical token “`gt`” is be interpreted as `>`, and the token “`and`” is interpreted as `& &` . Thus these tokens cannot be used as variable names.
- The identifier `static` is a reserved word in ECMAScript, but not in Ibex.
- Ibex defines an additional reserved word, “`assert`“, which will evaluate the expression which follows it, throwing a `ibex.assertion.failed` exception if the expression evaluates to `false`.
- To ensure that Ibex files appear the same in all text editors, tab characters are not allowed in Ibex files.

Some useful tutorials include:

- Netscape’s
- JavaScript 1.2 Reference. Although this document is out of date, it is arguably the best guide available for free on the Internet. The changes from JavaScript 1.2 (aka ECMA-262 r1) to 1.5 were minimal, and many of them were omitted from Ibex.
- O’Reilly’s
- JavaScript: The Definitive Guide, by David Flanagan and Paula Ferguson. The latest edition of this book covers JavaScript 1.5 (ECMA-262 r3).

- The official
- ECMA-262 specification. This is an extremely technical document.

Logging and Command Line Invocation

Very early in the loading process, Ibex begins logging messages about what it is doing. Where this output is logged to differs by platform; currently it goes to standard output when running inside a JVM, and to `$ TMPDIRbackslash ibex-log.txt` on Win32 (where `$ TMPDIR` is the value returned by `GetTempPath()`). The logs contain a lot of valuable debugging information and performance hints; if you are having trouble developing an Ibex application, be sure to check the logs.

If Ibex encounters a serious problem before it starts logging information, or if it is unable to open the log file, it will abort immediately with a critical abort, which will be displayed on the console for POSIX-native cores and in a dialog box for JVM-based and Win32-native cores.

You can invoke Ibex directly from the command line during development. When using a JVM, the invocation format is:

```
      java -jar <i>path-to-ibex-jar</i> [-sv] <i>source-location</i> [<i>initial-template
```

Where *path-to-ibex-jar* is the path to `ibex.jar`, which can be downloaded here.

On Win32, the invocation format is:

```
      ibex.exe [-v] <i>source-location</i> [<i>initial-template</i>]
```

The file `ibex.exe` is placed in Windows' ActiveX cache directory the first time Ibex is used on the machine. The ActiveX cache location depends on what version of Windows you are using; on newer versions of Windows it is `C:backslash WINDOWSbackslash DOWNLOADED PROGRAM FILESbackslash` . You can also extract `ibex.exe` from `ibex.cab`, which is available here.

The *source-location* parameter can be either the path to an .ibex archive, the http url of an .ibex archive, or the path to a directory comprising an unpacked .ibex archive.

The *initial-template* parameter is the stream name of a template to be used as the initial template. If ommitted, it defaults to `main`.

The `-v` option causes Ibex to enable verbose logging; this will cause it to log *lots* of information to the log file. This option will also substantially decrease Ibex's performance.

Copyright (C) 2004 Adam Megacz, verbatim redistribution permitted. Ibex is a trademark of Adam Megacz