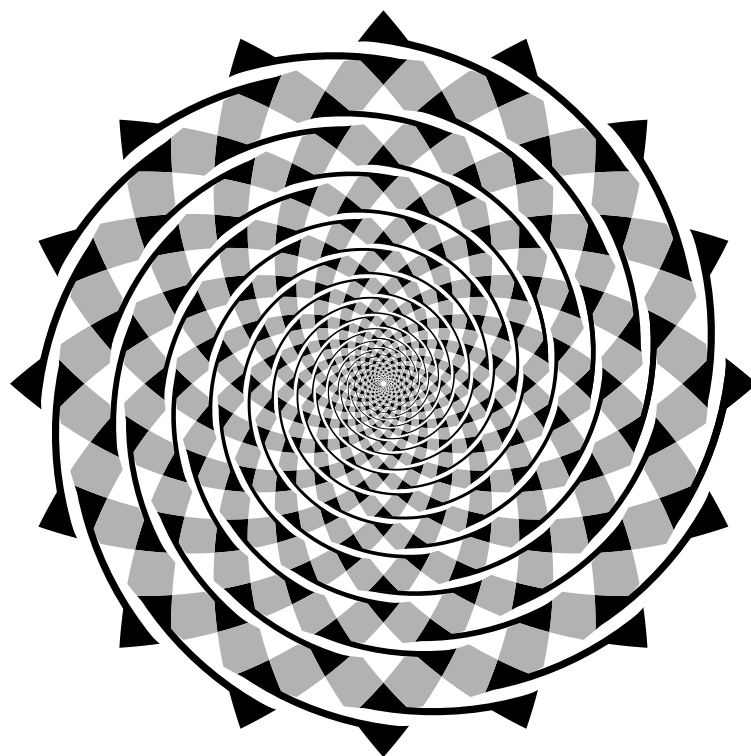


SVG ではじめる Web Graphics 2017 年度版

(情報メディア専門ユニット I)

平成 29 年 3 月 2 日改訂



フレイザーの渦巻き錯視

この文書についてお問い合わせは下記のメールまでご連絡ください。

©2006 – 2017 平野照比古

e-mail: hilano@ic.kanagawa-it.ac.jp

URL: <http://www.hilano.org/hilano-lab>

目次

第 1 章 SVG について	1
1.1 SVG とはなにか	1
1.2 XML とは	2
1.3 SVG の画像を表示する方法	4
1.3.1 SVG が取り扱えるソフトウェア	5
1.4 HTML5 について	5
1.4.1 HTML5 までの道程	5
1.4.2 HTML5 における新機能	5
第 2 章 SVG 入門	7
2.1 SVG ファイルを作成する方法と作成上の注意	7
2.2 SVG の基礎	11
2.2.1 座標系について	11
2.2.2 SVG 文書におけるコメントの記入方法	11
2.2.3 色について	11
2.3 直線	12
2.4 長方形	18
2.5 円と楕円	23
2.6 グラデーション	26
2.6.1 線形グラデーション	26
2.6.2 放射グラデーション	34
2.7 不透明度	37
付録 A SVG の色名	付録 1
付録 B 参考文献について	付録 5
付録 C JavaScript 入門	付録 9
C.1 JavaScript とは	付録 9
C.2 データの型	付録 9
C.2.1 プリミティブデータ型	付録 9
C.2.2 配列	付録 11
C.3 演算子	付録 11
C.3.1 代入、四則演算	付録 11

C.3.2	比較演算子	付録 12
C.4	制御構造	付録 13
C.4.1	if 文	付録 13
C.4.2	switch 文	付録 13
C.4.3	for 文と while 文	付録 14
C.5	関数	付録 15
C.5.1	関数の定義と呼び出し	付録 15
C.5.2	仮引数への代入	付録 16
C.5.3	arguments について	付録 17
C.6	変数のスコープと簡単な例	付録 18
C.6.1	JavaScript における関数の特徴	付録 21
C.6.2	クロージャ	付録 23
C.7	配列のメソッド	付録 26
C.8	オブジェクト	付録 29
C.8.1	配列とオブジェクト	付録 29
C.8.2	コンストラクタ関数	付録 31
C.8.3	オブジェクトリテラルと JSON	付録 35
C.8.4	ECMAScript5 のオブジェクト属性	付録 36
C.8.5	エラーオブジェクトについて	付録 41
付録 D	CSS について	付録 43

第1章 SVG について

1.1 SVG とはなにか

Web 上での各種規格は World Wide Web Consortium(W3C) と呼ばれる組織が中心になって制定しています。その Web Design and Applications¹ (図 1.1 参照) には次のように書かれています。

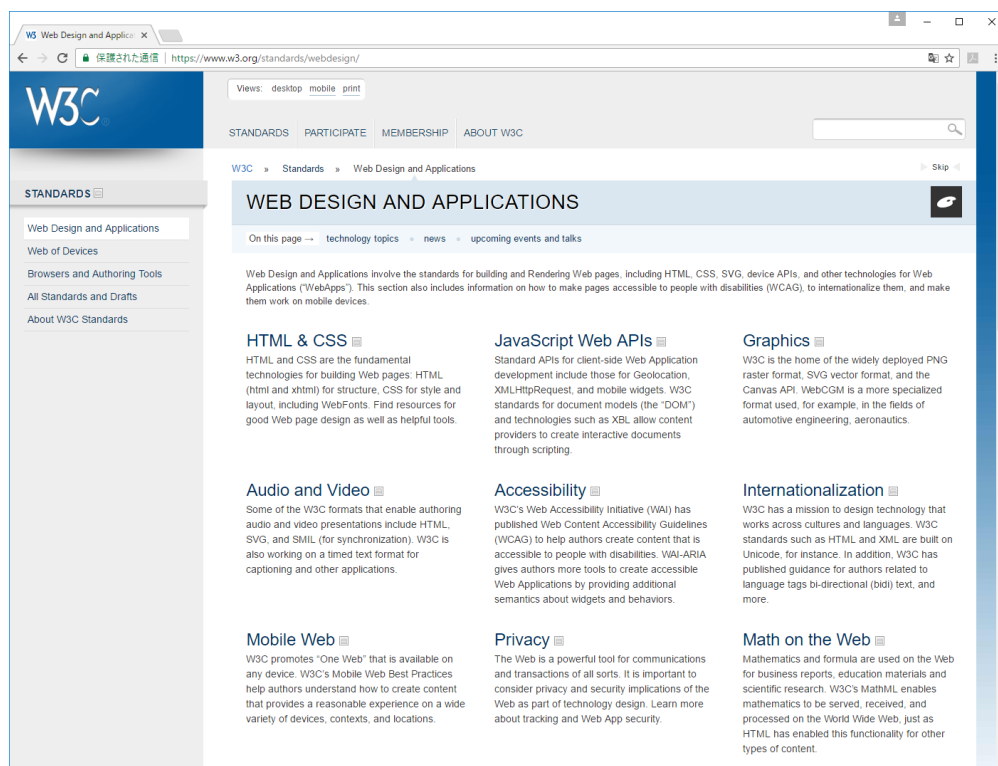


図 1.1: World Wide Web Consortium の Web Design and Application のページ (2017/2/27 参照)

Web Design and Applications involve the standards for building and Rendering Web pages, including HTML, CSS, SVG, device APIs, and other technologies for Web Applications (“ WebApps ”). This section also includes information on how to make pages accessible to people with disabilities (WCAG), to internationalize them, and make them work on mobile devices.

¹<http://www.w3.org/standards/webdesign/>

このページの Graphics をクリックすると W3C のグラフィックス関係の規格の解説のページへ移動します (図 1.2)。

このページの下に SVG の簡単な説明があります。

Scalable Vector Graphics (SVG) is like HTML for graphics. It is a markup language for describing all aspects of an image or Web application, from the geometry of shapes, to the styling of text and shapes, to animation, to multimedia presentations including video and audio. It is fully interactive, and includes a scriptable DOM as well as declarative animation (via the SMIL specification). It supports a wide range of visual features such as gradients, opacity, filters, clipping, and masking.

The use of SVG allows fully scalable, smooth, reusable graphics, from simple graphics to enhance HTML pages, to fully interactive chart and data visualization, to games, to standalone high-quality static images. SVG is natively supported by most modern browsers (with plugins to allow its use on all browsers), and is widely available on mobile devices and set-top boxes. All major vector graphics drawing tools import and export SVG, and they can also be generated through client-side or server-side scripting languages.

SVG の正式な規格書は図 1.2 のページの右側にある CURRENT STATUS の部分の SVG をクリックするとみることができます (図 1.3)。

画像を作成し、保存する形式を大きく分けるとビットマップ方式とベクター方式があります。

ビットマップ方式は画像を画素 (ピクセル) という単位に分け、その色の情報で画像を表します。したがってはじめに決めた大きさで画像の解像度が決まります。Adobe 社の Photoshop や Windows に標準でついてくるペイントはビットマップ方式の画像を作成するツールです。

これに対し、ベクター方式では線の開始位置と終了位置 (または長さや方向)、線の幅、色の情報などをそのまま持ちます。したがって画像をいくら拡大しても画像が汚くなることはありません。ベクター方式のソフトウェアはドロー系のソフトウェアとも呼ばれます。代表的なものとしては Adobe 社の Illustrator があります。

SVG はその名称からもわかるようにベクター形式の画像を定義するフォーマットのひとつです。

予習問題 1.1 次の事柄について調べなさい。

1. 画像の保存形式を調べ、それがビットマップ方式かベクター方式か調べなさい。
2. ビットマップ方式とベクター方式の画像形式の利点と難点をそれぞれ述べなさい。

1.2 XML とは

XML は eXtensible Markup Language の略です。“Markup Language” の部分を説明します。

ワープロなどで文書を作成するとき、ある部分は見出しなので字体をゴシックに変えるということを行います。このように文書には内容の記述の部分とそれを表示するための情報を含んだ部分があることになります。このように本来の記述以外の内容を含んだ文書をハイパーテキストと呼びます。Microsoft Word のようなワープロソフトが作成する文書もハイパーテキストです。このよ

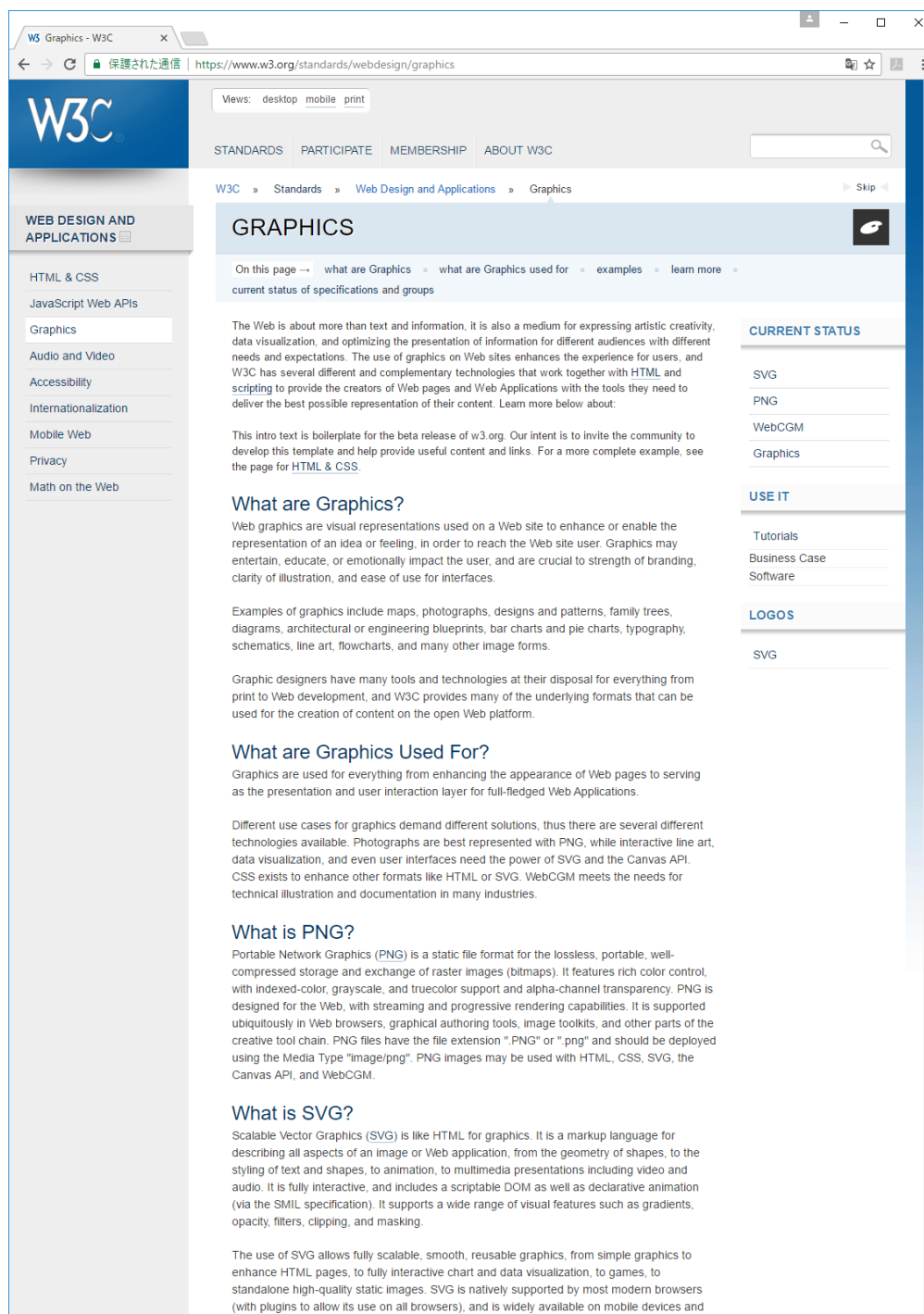


図 1.2: W3C の SVG に関するトップページ (2017/2/27 参照)

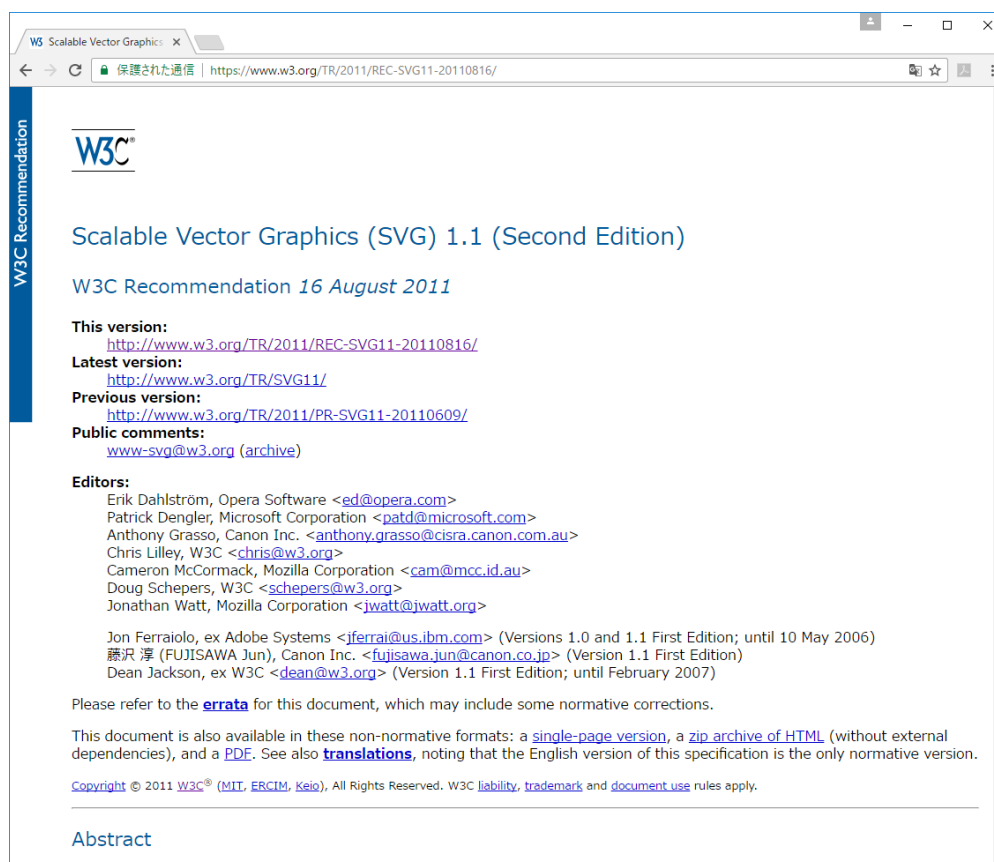


図 1.3: SVG の規格のページ (2017/2/27 参照)

うなハイパーテキストを表現するために文書の中にタグと呼ばれる記号を挿入したものが Markup Language です。

Web のページは普通 HTML(HyperText Markup Language) と呼ばれる形式で記述されています。HTML 文書は先頭が `<HTML>` 要素で最後が `</HTML>` で終わっています。ここで現れた `<HTML>` 要素がタグと呼ばれるものです。HTML 文書ではこのタグが仕様で決まっていてユーザが勝手にタグを定義することができません。一方、XML では extensible という用語が示すようにタグは自由に定義できます。したがって、XML を用いることでいろいろな情報を構造化して記述することが可能になっています。XML についてもっと知りたい場合には文献 [15]などを参照してください。

1.3 SVG の画像を表示する方法

SVG に基づいた画像を作成する方法については次章で解説をします。SVG は XML の構造を持ったベクター形式の画像を定義するフォーマットです。したがって、SVG に基づいたファイルだけでは単なる文字の羅列にしか見えず、これだけでは画像を表示することはできません。画像を表

示するためにはソフトウェアが必要になります。このテキストの題名である Web Graphics の観点からブラウザにより表示することを主に考えます。最近のブラウザは SVG の画像の表示をサポートしていますが、SVG の機能の一つであるアニメーション (第??章で説明します) のうち色のアニメーションについてはサポートしていません。色のアニメーションは CSS で実現できるので 2017 年度からはこの方法で説明をします。

1.3.1 SVG が取り扱えるソフトウェア

ブラウザ以外のいくつかのソフトウェアでも SVG ファイルを取り扱うことができます。

Adobe 社のドローソフト Illustrator は 簡単な SVG のファイルを表示したり、結果を SVG ファイルとして保存することが可能です。

1.4 HTML5 について

2015 年 3 月現在、代表的なブラウザは最新の HTML5 機能をインプリメントしています。

1.4.1 HTML5 までの道程

W3C は文法上あいまいであった HTML4 の規格を厳密な XML の形式に合う形にするために XHTML[28] を制定しました。これとは別にベンダー企業は 2004 年に WHATWG (Web Hypertext Application Technology Working Group) を立ち上げ、XHTML とは別の規格を検討し始めました。2007 年になって W3C は WHATWG と和解をし、WHATWG の提案を取り入れる形で HTML5 の仕様の検討が始まり、2014 年 10 月に規格が正式なもの (Recommendation) になりました。

1.4.2 HTML5 における新機能

HTML5 では HTML 文書で定義される要素だけではなく、それに付随する概念も含めます。HTML5 における新機能としては次のようなものがあります。

- 文書の構成をはっきりさせる要素が導入されています。文書のヘッダー部やフッター部を直接記述できます。
- フォントの体裁を記述する要素が非推奨となっています。これらの事項は CSS で指定することが推奨されています。
- SVG 画像をインラインで含むことができます。
- WebStorage

ローカルのコンピュータにその Web ページに関する情報を保存して、あとで利用できる手段を与えます。機能としては Cookie と同じ機能になりますが、Cookie がローカルに保存され

たデータを一回サーバーに送り、サーバーがそのデータを見て Web ページを作成するのに対し、WebStorage ではそのデータがローカルの範囲で処理されている点が一番の違いです。

Web ページの中には 2 度目に訪れた時に以前の情報を表示してくれるところがあります。これは Cookie に情報を保存しておき、再訪したとき、ブラウザが保存されたデータを送り、そのデータを用いてサーバーがページを構成するという手法で実現しています。WebStorage を用いると保存されたデータをサーバーに送ることなく同様のことが実現可能となるのでクライアントとサーバーの間での情報の交換する量が減少します。また、WebStorage では保存できるデータの量が Cookie と比べて大幅に増加しています。

- <canvas> 要素

インラインでの画像表示の方法として導入されました。画像のフォーマットはビットマップ方式で、JavaScript を用いて描きます。描かれた図形はオブジェクト化されないため、マウスのクリック位置にある図形はどれかなどの判断は自分でプログラムする必要があります。

また、アニメーションをするためには途中の図形の形や位置などを自分で計算する必要があり、複数の図形のアニメーションの同期も自分で管理する必要があります。

このテキストでははじめは単独で SVG による画像を描きますが、途中からは HTML 文書内で SVG の画像を表示し、それに対して構成要素を変えるプログラミングについて解説します。

第2章 SVG 入門

2.1 SVG ファイルを作成する方法と作成上の注意

SVG はテキストベースの画像表現フォーマットです。これを作成するためにはテキストエディタと呼ばれるソフトウェアを使います。Windows で標準についてくるメモ帳はテキストエディタの例です。

問題 2.1 エディタソフトウェアにはどのようなものがあるか調べなさい。特に *Unix* ではどのようなものが有名であるか調べる。また、XML 文書を編集するためのテキストエディタにどのようなものがあるか調べなさい。

テキストエディタで SVG ファイルを作成するときの注意を次にあげておきます。

- SVG ファイルの標準の文字コードは UTF-8 です。Windows のメモ帳では文字コードを UTF-8 で保存するとファイルの先頭に BOM と呼ばれるコードが付き、XML 形式のファイルとしては正しくないものになってしまいます。エディタによっては保存する文字コードにこの BOM なしで保存を選択できるものがあります。
- SVG ファイルの内容は HTML ファイルのようにタグをつけた形で記述します。タグで定義されるものを要素と呼びます。
- HTML ファイルのタグでは次のようなことが可能です。
 - 要素名は大文字小文字の区別がなく、要素の開始を表す部分と終了を表す部分で大文字小文字が違っていても問題はおきません。
 - 改行を示す `
` 要素のように終了部分がない要素もあります。
- SVG ファイルは厳密な XML の構文に従うので要素の開始部分に対応する対応する終了部分を必ず記述する必要があります (簡略な形式もあります)。また、大文字小文字も完全に一致している必要があります。うまく動かないときにはこの点を確認しましょう。¹
- 要素などに記述する単語は英語です。複数形を示す `s` がついているのを忘れたりするだけで動かなくなるのでよくチェックしましょう。
- ファイルの拡張子は `svg` が標準です。

¹ 厳密な XML の構文に従う HTML の文書としては XHTML の形式があります。この形式では `
` 要素の代わりに `
` と記述することで終了タグが不要になります。

このテキストではいろいろな SVG ファイルのリストが出てきます。リストの入力に対しては次のことに注意してください。

- リストの各行の先頭には行番号が書いてありますが、これは説明のためにつけたものなので、エディタで入力するときは必要ありません。
- 要素の間の空白は原則的にはひとつ以上あれば十分です。見やすく読みやすくなるようにきれいに記述しましょう。

Google Chrome における XML 文書としてのエラーがあった場合の表示例を解説します。

SVG リスト 2.1: XML 文書としてエラーがあった場合

```
1 <?xml version="1.0" encoding="UTF-8" ?>
2 <svg xmlns="http://www.w3.org/2000/svg"
3     xmlns:xlink="http://www.w3.org/1999/xlink"
4     height="100%" width="100%">
5   <title>XML 文書としてエラーがある場合</title>
6   <rect x="50"y="50" width="100" height="100" fill="gray" />
7 </svg>
```

リスト 2.1 では 6 行目で `x="0" y="0"` と記すべきところを `x="0"y="0"` と空白を入れなかった場合のエラーの表示です。

Google Chrome ではエラーがあった位置を行数と桁で指摘してくれます。

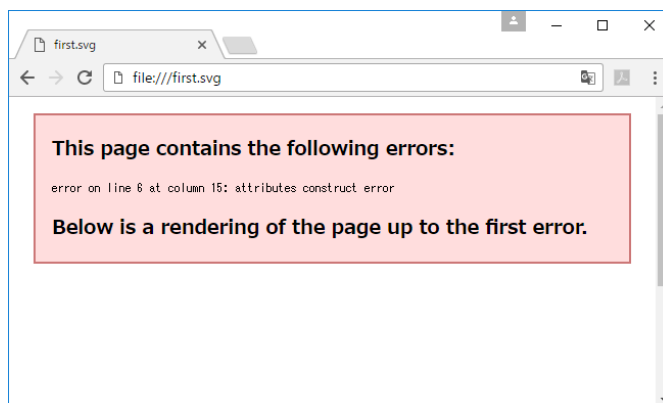


図 2.1: Google Chrome におけるエラーメッセージ

これでよくわからない場合には <http://w3c.github.io/developers/tools/> を利用する方法があります (図 2.2)。

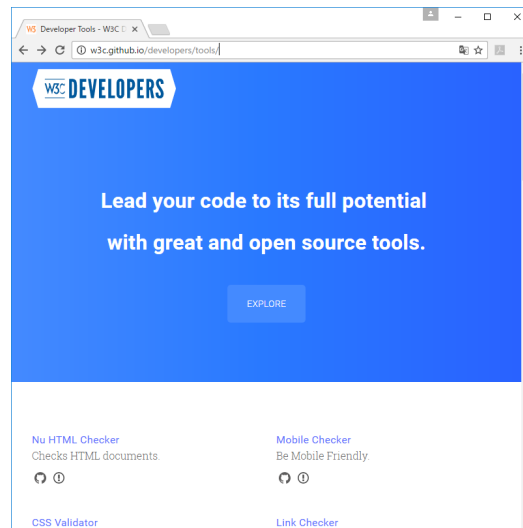


図 2.2: Validator のトップ画面

1. 図 2.2 で Nu Html Checker をクリックすると図 2.3 が表示されます。

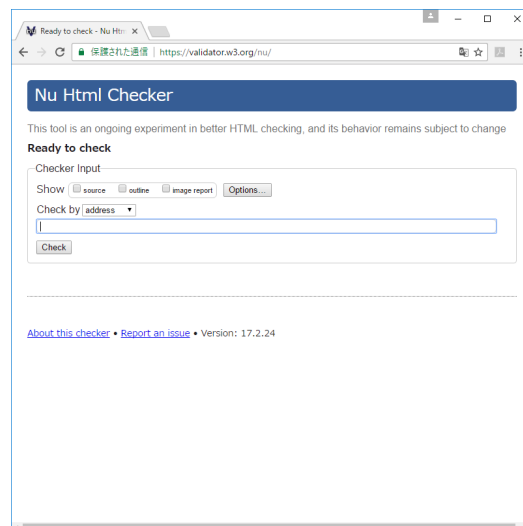


図 2.3: Nu Html Checker の画面

2. ここで「Check by」のプルダウンメニューで「file upload」を設定し、「ファイル選択」でチェックしたいファイルを指定します(図 2.4)。

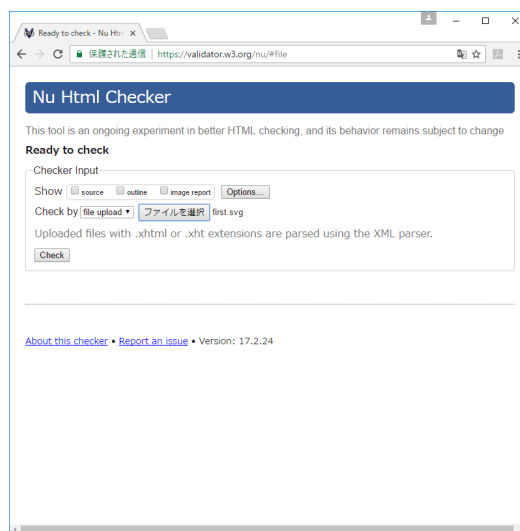


図 2.4: ファイルのアップロード

3. ここで「check」のボタンを押すとデータが送られてデータの検証が行われます(図 2.5)。

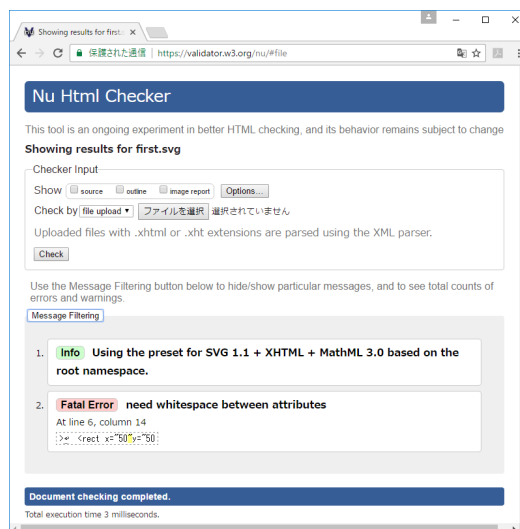


図 2.5: 検証結果の表示画面

このページを下のほうに間違いの理由と場所がより具体的に指摘されています。

2.2 SVG の基礎

2.2.1 座標系について

ものの位置を示すためには座標系とその単位が必要です。SVG の場合は初期の段階では左上隅が原点 (0,0) になり、単位はピクセル (px) です。水平方向は右のほうへ行くにつれて、垂直方向は下に行くほどそれぞれ値が増大します (図 2.6 参照)。

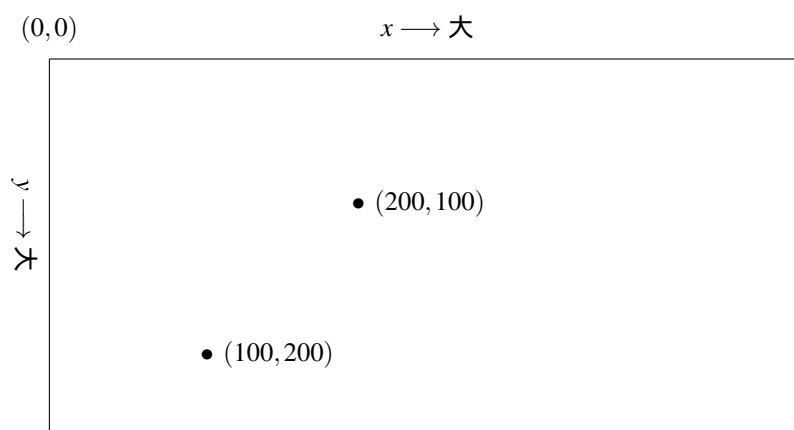


図 2.6: SVG の座標系

後で述べるように SVG ではいくつかの図形をまとめて平行移動したり、回転させることが可能です。また、水平方向や垂直方向に拡大することも可能です (座標系がすでに回転していればその方向に拡大が行われます)。

2.2.2 SVG 文書におけるコメントの記入方法

SVG 文書でコメントを入れる方法は HTML 文書と同様にコメントの部分を `<!--` と `-->` ではさみます。この本の中ではコメントをほとんど利用していません。

2.2.3 色について

SVG で色を指定する方法は次の 3 種類です。

1. 名称による指定

英語名で色を指定します。どのような名称があり、どのような色になるかは付録 A を見てください。

2. rgb 関数による色の指定

色の 3 原色、赤、緑、青のそれぞれに対し 0～255 の値を割り当てて指定します。なお、rgb

関数では 0 ~ 255 の値ではなく % で色の割合を指定することも可能です。たとえば red は `rgb(100%,0%,0%)` と表されます。

3. 16 進数による色の指定

0 ~ 255 までの数は 16 進数 2 桁で表すことができます。これを用いて `rgb` で定めた色を 16 進数 6 桁で表示ができます。

このほかに各色の構成を 16 進数 1 桁で表し、16 進数 3 桁で指定することもできます。

SVG で利用できる色名と色のサンプルは付録 A を参考にしてください。なお、特別な色として、塗らないことを指示する `none` があります。

2.3 直線

図形のうちで一番簡単な直線を描いてみましょう。次の例を見てください。

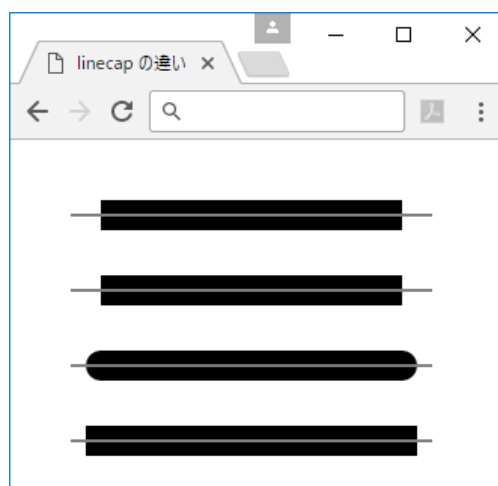


図 2.7: 直線と端の指定 `stroke-linecap` の違い

これを描いた SVG ファイルの内容は次のとおりです。

SVG リスト 2.2: 直線の例

```

1 <?xml version="1.0" encoding="UTF-8" ?>
2 <svg xmlns="http://www.w3.org/2000/svg"
3     xmlns:xlink="http://www.w3.org/1999/xlink"
4     height="100%" width="100%">
5 <title>linecap の違い</title>
6 <g transform="translate(60,50)">
7   <line x1="0" y1="0" x2="200" y2="0" stroke-width="20" stroke="black"/>
8   <line x1="0" y1="50" x2="200" y2="50" stroke-width="20" stroke="black"
9       stroke-linecap="butt"/>
10  <line x1="0" y1="100" x2="200" y2="100" stroke-width="20" stroke="black"

```



```

11     stroke-linecap="round"/>
12 <line x1="0" y1="150" x2="200" y2="150" stroke-width="20" stroke="black"
13     stroke-linecap="square"/>
14 <line x1="-20" y1="0" x2="220" y2="0" stroke-width="2" stroke="gray" />
15 <line x1="-20" y1="50" x2="220" y2="50" stroke-width="2" stroke="gray" />
16 <line x1="-20" y1="100" x2="220" y2="100" stroke-width="2" stroke="gray" />
17 <line x1="-20" y1="150" x2="220" y2="150" stroke-width="2" stroke="gray" />
18 </g>
19 </svg>

```

- 1 行目はこのファイルが XML の規格に基づいて書かれていることを宣言しています。この行の先頭に空白があってははいけません。また<?xml の部分も空白があってははいけません。

それに続く version や encoding の部分は属性、=の右側はその属性値とそれぞれよばれます。属性値は必ず"で囲む必要があります。

- version は XML の規格のバージョンを表します。
- encoding はこのファイルが使用している文字の表し方 (エンコーディングとよばれます。)を表します。SVG で日本語を含む文字列を扱うためには UTF-8 と呼ばれるエンコーディングにする必要があります。XML 文書は通常、UTF-8 でエンコーディングするのが標準となっています。

- 2 行目から 4 行目の部分は<svg> 要素を定義しています。このように一番初めに現れる要素をルート要素とよびます。SVG のファイルではルート要素は必ず<svg> 要素となります。ルート要素は XML 文書では 1 回しか現れてはいけません。

- <svg> 要素の属性値として xmlns が指定されています。これは XML 名前空間とよばれていて、要素やそれに対する属性が定義されている場所を示しています。ここでは<svg> 要素が定義されている場所を指定しています²。
- 次の属性 xmlns:xlink とは xlink が定義する名前空間の参照場所を指定しています。このファイルではこの名前空間を使用していないので省略してもかまいません。
- width と height はこの SVG の画像を表示する大きさを指定しています。ここでは両者とも 100% なのでブラウザの画面全体という意味になります。

- 6 行目はこれから描く図形をひとまとめにするために<g> 要素を用いています。transform はこのグループ化された図形を移動することを指定しています。この属性値は表 2.1 のようなものがあります。

ここでは translate(60,50) となっているので原点の位置が左から 60、上から 50 の位置に移動します。

- 7 行目から 13 行目で stroke-linecap で指定される直線の両端の形が異なるものを 4 つ定義しています。

²XML 文書にはこのほかに<!DOCTYPE で始まる DTD(Document Type Definition) への参照が通常あります。なくても動作するのでこのテキストでは省略しました。

表 2.1: transform の属性値

属性値の関数名	機能	例
translate	平行移動	translate(20,40)
rotate	回転 (単位は度, 向きは時計回り)	rotate(30)
scale	拡大・縮小	scale(0.5), scale(1,-1)

- <line> 要素は直線を定義します。この要素は<g> 要素の内側に<line> 要素は<g> 要素の子要素になっているといえます。また、<g> 要素は<line> 要素の親要素であるとも言います。
- 属性としては表 2.2 を参考にしてください。

表 2.2: <line> 要素の属性

属性名	意味	属性値
x1	開始位置の x 座標	数値
y1	開始位置の y 座標	数値
x2	終了位置の x 座標	数値
y2	終了位置の y 座標	数値
stroke	直線を塗る色	色名または rgb 値で与える
stroke-width	直線の幅	数値
stroke-linecap	直線の両端の形を指定	butt (default) 何もつけ加えない round 半円形にする square 長方形にする

- はじめの直線は開始位置が (0,0) で終了点が (200,0) となっています。
 - 線の幅 (stroke-width) が 20 でその色 (stroke) を黒 (black) に指定しています。
 - はじめの二つが同じ形をしているので linecap のデフォルト値が butt であることがわかります。
 - 最後の文字列 /> はこの要素が子要素を含まないことを示すための簡略的な記述方法です。
- 14 行目から 17 行目でははじめの直線 4 本より幅が狭いものを灰色 (gray) で描いています。これは描かれる線分の位置がどこに来るかを確かめるためです。
この図から直線の幅は与えられた点の位置から両側に同じ幅で描かれることがわかります。
 - この灰色の線がすべて見えていることから後から書かれた図形のほうが優先して表示されることがわかります。
 - 18 行目では<g> 要素の内容が終了したことを示す</g>があります。
 - 19 行目では<svg> 要素の内容が終了したことを示す</svg>があります。

フィックの錯視 直線を組み合わせてできる一番簡単な錯視図形がフィックの錯視 [32, 61 ページ] です (図 2.8)。

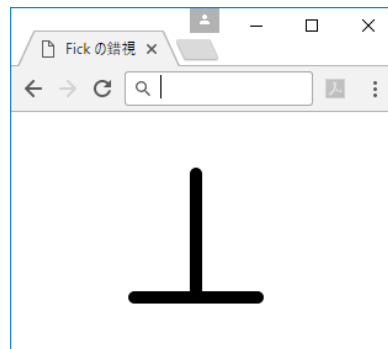


図 2.8: フィックの錯視

水平の線分と垂直な線分の長さが同じなのにその位置が中央にあると長く見えるというとして知られているものです。簡単な図形ながら錯視量が大きいことで有名な図形です。これを描く SVG 文書のリストがリスト 2.3 です。

SVG リスト 2.3: フィックの錯視

```
1 <?xml version="1.0" encoding="UTF-8" ?>
2 <svg xmlns="http://www.w3.org/2000/svg"
3     xmlns:xlink="http://www.w3.org/1999/xlink"
4     height="100%" width="100%">
5 <title>Fick の錯視</title>
6 <g transform="translate(100,150)">
7   <line x1="0" y1="0" x2="100" y2="0"
8       stroke-width="10" stroke="black" stroke-linecap="round" />
9   <line x1="50" y1="0" x2="50" y2="-100"
10       stroke-width="10" stroke="black" stroke-linecap="round" />
11 </g>
12 </svg>
```

- 7 行目から 8 行目で水平の直線を描いています。
- 9 行目から 10 行目で垂直の直線を描いています。

問題 2.2 フィックの錯視に関連して次のことを行いなさい。

1. 90° 回転した図形でも同じように見えることを確認しなさい。
2. 垂直な直線の位置を水平方向に移動すると見え方はどのように変わるか調べなさい。
3. 中央の線分の長さをどれだけ縮小したら同じ長さに見えるか調べなさい。

ミュラー・ライヤーの錯視 図 2.9 は中央の線分が同じ長さなのに両端の矢印の向きが異なることで大きさが違って見えるというミュラー・ライヤーの錯視 [32, 57 ページ] として知られる図形です。

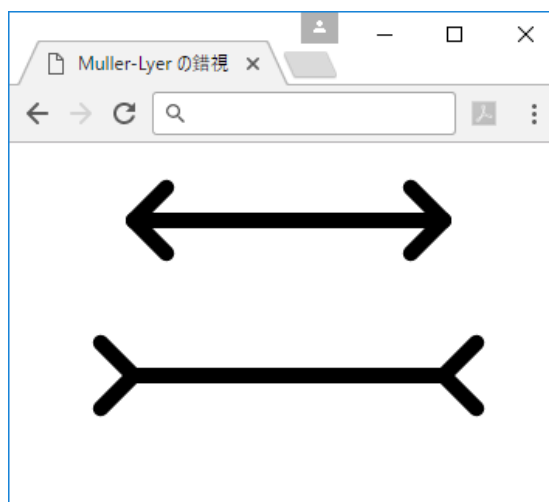


図 2.9: ミュラー・ライヤー錯視

この図形のリストでは両端の矢印の長さや角度を最小限の手間で変えられるようにしています。

SVG リスト 2.4: ミュラー・ライヤーの錯視

```

1  <?xml version="1.0" encoding="UTF-8" ?>
2  <svg xmlns="http://www.w3.org/2000/svg"
3      xmlns:xlink="http://www.w3.org/1999/xlink"
4      height="100%" width="100%">
5  <title>Muller-Lyer の錯視</title>
6  <defs>
7      <line class="Line" id="Line" x1="0" y1="0" x2="30"
8          stroke-width="10" stroke-linecap="round" stroke="black" />
9      <g id="edge">
10         <g transform="rotate(45)" >
11             <use xlink:href="#Line" />
12         </g>
13         <g transform="rotate(-45)" >
14             <use xlink:href="#Line" />
15         </g>
16     </g>
17 </defs>
18 <g transform="translate(80,50)" >
19     <line x1="0" y1="0" x2="200" y2="0"
20         stroke-width="10" stroke-linecap="round" stroke="black" />
21     <use xlink:href="#edge"/>
22     <g transform="translate(200,0)" >
23         <g transform="scale(-1,1)" >
24             <use xlink:href="#edge"/>
25         </g>

```

```

26     </g>
27 </g>
28 <g transform="translate(80,150)" >
29     <g>
30         <line x1="0" y1="0" x2="200" y2="0"
31             stroke-width="10" stroke-linecap="round" stroke="black"/>
32         <g transform="scale(-1,1)">
33             <use xlink:href="#edge"/>
34         </g>
35         <g transform="translate(200,0)">
36             <use xlink:href="#edge"/>
37         </g>
38     </g>
39 </g>
40 </svg>

```

- 両端にある矢印をすべて同じ長さにするために雛形を定義することにします。雛形は<defs>要素の中に記述します (6 行目)。
- 矢印を構成する直線を7 行目から8 行目で定義します。この要素は後で参照するために id で名前を付けておきます。ここでは Line という名称になっています。
- 次にこの直線を使って片側の矢印を構成します。二つの直線からなるので後でまとめて一つのものとして参照するために二つの矢印をグループ化します。それに利用するのが9 行目にある<g> 要素です。後で参照するために edge という名称を与えています。
- この中に二つの直線を描きますが、それぞれを $\pm 45^\circ$ 傾けるためにさらに<g> 要素を用います (10 行目と13 行目)。10 行目の<g> 要素には transform で rotate(45) という値を指定しているので原点 (0,0) を中心として 45° 時計回りに傾くことになります。
- この<g> 要素のなかに先ほど定義した直線を参照する記述をします。これが <use> 要素 です (11 行目と14 行目)。参照先を指定するために xlink:href を用います。参照先は id で定義された名称の前に#を付けます。
- 18 行目から27 行目の間が上の矢印を記述している部分です。

```

18 <g transform="translate(80,50)" >
19     <line x1="0" y1="0" x2="200" y2="0"
20         stroke-width="10" stroke-linecap="round" stroke="black" />
21     <use xlink:href="#edge"/>
22     <g transform="translate(200,0)" >
23         <g transform="scale(-1,1)" >
24             <use xlink:href="#edge"/>
25         </g>
26     </g>
27 </g>

```

- 19 行目から20 行目で横の直線を定義しています。
- 21 行目で左の矢印の部分定義しています。
- 22 行目から26 行目で右の矢印を描いています。
- 右の矢印は全体を横の直線の左端に平行移動させます (`transform="translate(200,0)"`)。
- 矢印の向きを反転させるために23 行目で `transform="scale(-1,1)"` をしています。 x 座標の値を -1 倍し、 y 座標の値をそのまま (1 倍) に指定しています。

- 28 行目から39 行目で下の部分の図形を定義しています。

```

28 <g transform="translate(80,150)" >
29   <g>
30     <line x1="0" y1="0" x2="200" y2="0"
31       stroke-width="10" stroke-linecap="round" stroke="black"/>
32     <g transform="scale(-1,1)">
33       <use xlink:href="#edge"/>
34     </g>
35     <g transform="translate(200,0)">
36       <use xlink:href="#edge"/>
37     </g>
38   </g>
39 </g>

```

- 矢印の向きが上と逆になっているので左の矢印は属性 `transform="scale(-1,1)"` を持つ `<g>` 要素の中に入れて左右の向きの入れ替えを行っています。
- 右の矢印は属性 `transform="translate(200,0)"` を持つ `<g>` 要素の中に入れて平行移動しています。

問題 2.3 ミューラー・ライヤーの錯視図形で次のことを調べなさい。

1. 両端の矢印の長さを変える
2. 両端の矢印の色を変えることとそのときの見え方の違い
3. 両端の矢印の角度を変えることとそのときの見え方の変化
4. `<defs>` 要素を用いないでこの図形を定義して、矢印の長さを変えたときの手間の違い

2.4 長方形

長方形を表すのは `<rect>` 要素です。長方形の属性を表 2.3 に掲げます。

表 2.3: <rect> 要素の属性

属性名	意味	属性値
x	長方形の左上の位置の x 座標	数値
y	長方形の左上の位置の y 座標	数値
width	長方形の幅	負でない数値
height	長方形の高さ	負でない数値
stroke-width	長方形の縁取りの幅	負でない数値
stroke	長方形の縁取りの色	色
fill	長方形の内部の塗りつぶしの色	色
rx	長方形の角に丸みを付けはじめる水平方向の位置	負でない数値
ry	長方形の角に丸みを付けはじめる垂直方向の位置	負でない数値

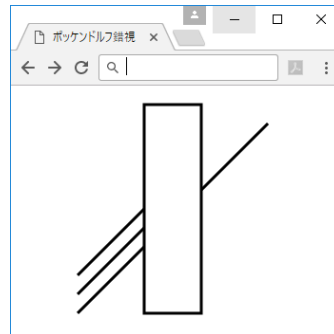


図 2.10: ポッケンドルフ錯視

ポッケンドルフの錯視 図 2.10 は中央部が隠されているために直線の対応が見誤るといいうポッケンドルフの錯視 [32, 58 ページ] として知られる図形です。長方形と直線を使って描いています。

SVG リスト 2.5: ポッケンドルフ錯視

```

1  <?xml version="1.0" encoding="UTF-8" ?>
2  <svg xmlns="http://www.w3.org/2000/svg"
3      xmlns:xlink="http://www.w3.org/1999/xlink"
4      height="100%" width="100%">
5  <title>ポッケンドルフ錯視</title>
6  <g transform="translate(170,140)">
7      <line x1="100" y1="-100" x2="-100" y2="100"
8          stroke-width="3" stroke="black"/>
9      <line x1="0" y1="-20" x2="-100" y2="80"
10         stroke-width="3" stroke="black"/>
11     <line x1="0" y1="-40" x2="-100" y2="60"
12         stroke-width="3" stroke="black" />
13     <rect x="-30" y="-120" width="60" height="220"
14         stroke-width="3" stroke="black" fill="white" />
15 </g>

```

16 </svg>

- 3本の直線を下から描いています。
- 11行目から12行目で一番下にある直線を描いています。これが右側に飛び出しているものです。
- 7行目から8行目、9行目から10行目に一番下の直線より半分の長さのものを縦方向に20ずつ移動した形で描いています。
- 最後に、中央部を隠すように長方形を描いています(13行目から14行目)。ここでは内部を白で塗りつぶしているの为先に描かれた直線の部分でこれと重なる部分は見えなくなります。

ここでは属性を別々に指定していますがこれらをまとめて指定できる `style` という属性もあります。この場合は次のように指定できます。

```
style="fill:black;stroke:red;stroke-width:4"
```

`name="value"`の代わりに `name:value` と表し、属性同士の間は ;(セミコロン) で区切ります。

色の対比 図 2.11 は正方形の内部が同じ色なのに周りの色で見え方が異なるという錯視図形です。

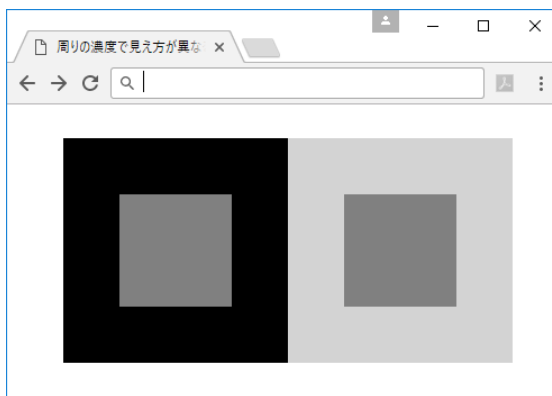


図 2.11: 周りの濃度で見え方が異なる

SVG リスト 2.6: 周りの濃度で見え方が異なる

```
1 <?xml version="1.0" encoding="UTF-8" ?>
2 <svg xmlns="http://www.w3.org/2000/svg"
3     xmlns:xlink="http://www.w3.org/1999/xlink"
4     height="100%" width="100%">
5   <title>周りの濃度で見え方が異なる</title>
6   <g transform="translate(50,30)" >
```



```

7      <rect x="0" y="0" width="200" height="200" fill="black" />
8      <rect x="50" y="50" width="100" height="100" fill="gray" />
9  </g>
10     <g transform="translate(250,30)" >
11         <rect x="25" y="25" width="150" height="150" fill="gray"
12             stroke-width="50" stroke="lightgray" />
13     </g>
14 </svg>

```

この図形は左の部分は二つの正方形を重ねて描いています。右の部分は縁取りの幅を大きくして同じような大きさの図形になる用の表示位置や長方形の大きさを調整しています。同じ図形を描くのもいろいろな方法があることを確認してください。

- 左の部分は6行目から9行目の部分で描かれています。大きな正方形(7行目)の上に小さな正方形(8行目)を描いています。
- 右の部分は11行目から12行目のひとつの正方形で描かれています。線の幅(stroke-width)の半分だけ元来の図形の外側にはみ出しますので左上の位置をその分だけ移動させています。また、線の幅だけ width と height の値が左の正方形の値より小さくなっています。

問題 2.4 図 2.12 の 6 個の長方形は一番左が赤(#FF0000)で一番右がオレンジ(#FFA500)で塗られています。間にある長方形の色はこの 2 つの色を補間しています [37, カラー図版 3]。

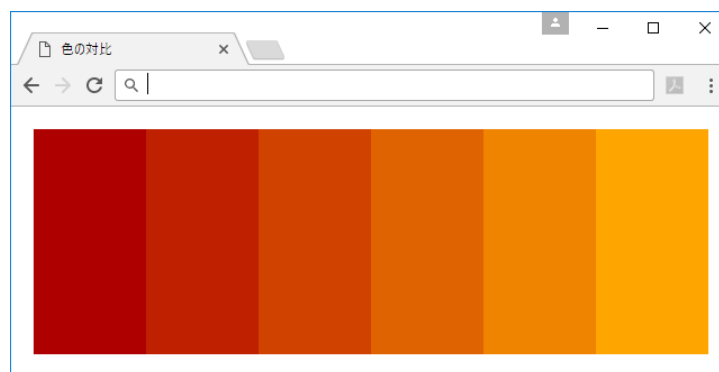


図 2.12: 色の対比

境界での色の差が強調されて見えますが、境界を指で隠すと両者の色の区別がなくなります。この図を作成しなさい。また、ほかの色の組み合わせでも調べなさい。

ヘルマン格子 図 2.13 は白い線の交差している位置に灰色のちらつきが表れるというヘルマン格子 [32, 180 ページ] と呼ばれるものです。

SVG リスト 2.7: ヘルマン格子

```

1 <?xml version="1.0" encoding="UTF-8" ?>
2 <svg xmlns="http://www.w3.org/2000/svg"

```

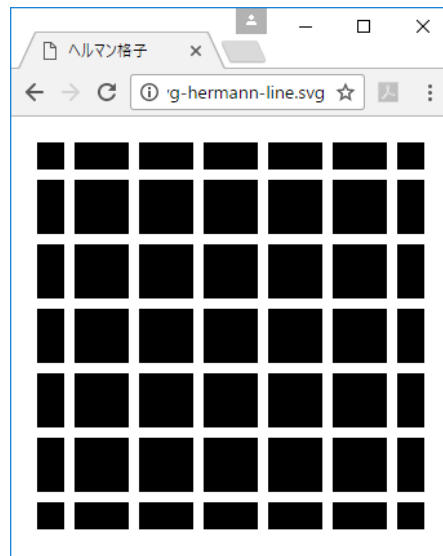


図 2.13: ヘルマン格子

```

3      xmlns:xlink="http://www.w3.org/1999/xlink"
4      height="330" width="330">
5  <title>ヘルマン格子</title>
6  <defs>
7      <line id="line" x1="0" y1="0" x2="0" y2="300" stroke-width="8"/>
8      <use id="vertical" x="0" y="0" xlink:href="#line" stroke="white"/>
9      <g id="horizontal" transform="rotate(-90)">
10         <use x="0" y="0" xlink:href="#line" stroke="white"/>
11     </g>
12 </defs>
13 <g transform="translate(20,20)">
14     <rect x="0" y="0" width="300" height="300" fill="black" />
15     <use x="25" y="0" xlink:href="#vertical" />
16     <use x="75" y="0" xlink:href="#vertical" />
17     <use x="125" y="0" xlink:href="#vertical" />
18     <use x="175" y="0" xlink:href="#vertical" />
19     <use x="225" y="0" xlink:href="#vertical" />
20     <use x="275" y="0" xlink:href="#vertical" />
21
22     <use x="0" y="25" xlink:href="#horizontal" />
23     <use x="0" y="75" xlink:href="#horizontal" />
24     <use x="0" y="125" xlink:href="#horizontal" />
25     <use x="0" y="175" xlink:href="#horizontal" />
26     <use x="0" y="225" xlink:href="#horizontal" />
27     <use x="0" y="275" xlink:href="#horizontal" />
28 </g>
29 </svg>

```

- 7 行目で水平垂直に引く直線の雛形を決めています。ここでは長さや幅だけ定めています。

- 8 行目で垂直方向の直線の属性 `stroke` を `white` に設定しています。
- 9 行目から 11 行目で垂直方向の直線を回転させて水平方向の直線にしています。
- 10 行目で色を `white` に設定しています。
- 14 行目で背景を黒にするための正方形を定義しています。
- 15 行目から 20 行目で垂直方向の直線を描いています。<use> 要素 のなかで属性 `x` や `y` を指定することで参照している図形の表示位置を移動できます。
- 22 行目から 27 行目で水平方向の直線を描いています。

問題 2.5 リスト 2.7 について次の事柄を検討しなさい。

1. 成分の間隔や幅、または色をいろいろ変えてどれが一番良く錯視が見えるか
2. 垂直線をひとつのグループにして、それを参照することで水平線の記述を簡略化すること
3. 正方形を並べて同じような図形を描くこと

問題 2.6 図 2.14 はカフェウォール錯視 [32, 62 ページ] と呼ばれる錯視図形です³。水平線はすべて平行なのですが黒い正方形がずれているために平行に見えません。この図形を描きなさい。

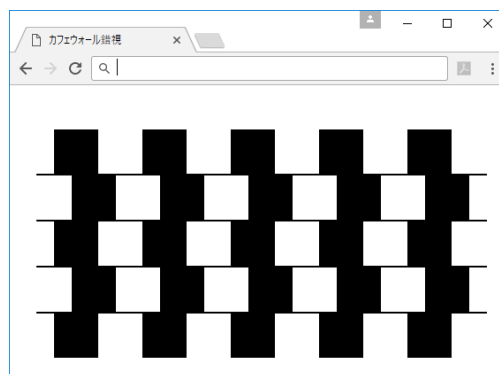


図 2.14: カフェウォール錯視

2.5 円と楕円

<circle> 要素は円の属性を表します。また、<ellipse> 要素は楕円を表します。表 2.4 にこの二つの要素の代表的な属性を掲げます。

楕円の属性は円の場合の半径を表す `r` の代わりに `x` 軸、`y` 軸の方向の軸の長さをそれぞれ表す `rx`、`ry` 属性があります。この値を同じにすれば円となります。

³水平線が黒の場合にはミュンスターベルグ錯視と呼ばれます。

表 2.4: 円と楕円の属性

属性名	説明	値
cx	円、楕円の中心の x 座標	数値
cy	円、楕円の中心の y 座標	数値
r	円の半径	数値
rx	楕円の x 軸方向の長さ	数値
ry	楕円の y 軸方向の長さ	数値
stroke	縁取りを塗る色	色名または rgb 値で与える
stroke-width	縁取りの幅	数値
fill	内部を塗る色	色名または rgb 値で与える

周りの大きさで見え方が変わる 例 2.15 は同じ大きさの円の周りに大きさの違う円を並べたものです。中央の円は、周りの円が小さいと大きく、周りの円が大きいと小さく見えます。

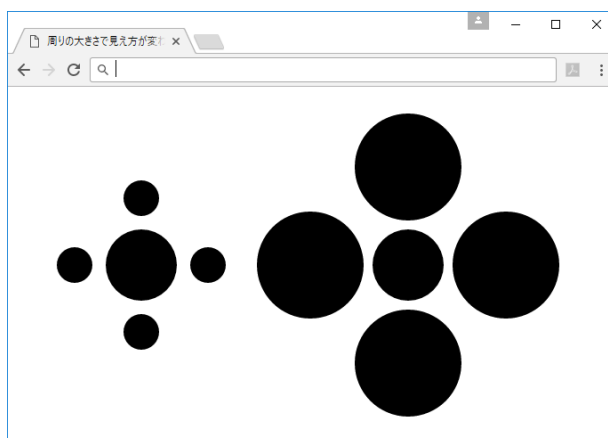


図 2.15: 周りの大きさで見え方が変わる

SVG リスト 2.8: 周りの大きさで見え方が変わる

```

1  <?xml version="1.0" encoding="UTF-8" ?>
2  <svg xmlns="http://www.w3.org/2000/svg"
3      xmlns:xlink="http://www.w3.org/1999/xlink"
4      height="100%" width="100%">
5      <title>周りの大きさで見え方が変わる</title>
6      <defs>
7          <circle cx="0" cy="0" id="CCircle" r="40" fill="black" />
8          <circle cx="0" cy="0" id="LCircle" r="60" fill="black" />
9          <circle cx="0" cy="0" id="SCircle" r="20" fill="black" />
10     </defs>
11     <g transform="translate(150,200)">
12         <use xlink:href="#CCircle"/>

```

```

13     <g transform="translate(0,75)">
14         <use xlink:href="#SCircle"/>
15     </g>
16     <g transform="rotate(90),translate(0,75)">
17         <use xlink:href="#SCircle"/>
18     </g>
19     <g transform="rotate(180),translate(0,75)">
20         <use xlink:href="#SCircle"/>
21     </g>
22     <g transform="rotate(270),translate(0,75)">
23         <use xlink:href="#SCircle"/>
24     </g>
25 </g>
26 <g transform="translate(450,200)">
27     <use xlink:href="#CCircle"/>
28     <g transform="translate(110,0)">
29         <use xlink:href="#LCircle"/>
30     </g>
31     <g transform="rotate(90),translate(110,0)">
32         <use xlink:href="#LCircle"/>
33     </g>
34     <g transform="rotate(180),translate(110,0)">
35         <use xlink:href="#LCircle"/>
36     </g>
37     <g transform="rotate(270),translate(110,0)">
38         <use xlink:href="#LCircle"/>
39     </g>
40 </g>
41 </svg>

```

- 7行目で中央にある円を定義し、それに CCircle という id を付けています。
- 8行目で右側の周りに置く円のひとつを定義し、それに LCircle という id を付けています。
- 9行目で左側に描く円を定義しています。この円には LCircle という id を付けています。
- 11行目から25行目で左側の図形を定義しています。(0,0)の位置に中央の円を描き、その周りに SCircle で参照する小さな円を4つ付けています。小さな円を描く方法として16行目で transform の値を rotate(90),translate(0,75) と二つ指定しています。これは次のように記述したものと同じです。

```

    <g transform="rotate(90)">
        <g transform="translate(0,75)">
            <use xlink:href="#SCircle"/>
        </g>
    </g>

```

- 右側の図形も26行目から40行目で同様に描いています。

問題 2.7 リスト 2.15 において次のことをしなさい。

1. `<defs>` 要素内で定義された、周りにある円の属性値を変えて、全体の記述を簡単にしなさい。
2. 中心の円や周りの円の色を変えたときに見え方が変わるかどうか調べなさい。
3. 円の代わりに正方形で同様の図形を作成しなさい。

問題 2.8 図 2.16 はデルブーフの錯視 [32, 59 ページ] と呼ばれています。この図は [37, 131 ページ 図 12.7] からとりました。左右の単独にある円が中央で重なっています。小さいほうの円は左より大きく見え、大きいほうの円は右より小さく見えます。

この現象は円を正方形に変えても起こります。正方形による同様の図を描いて確認しなさい。

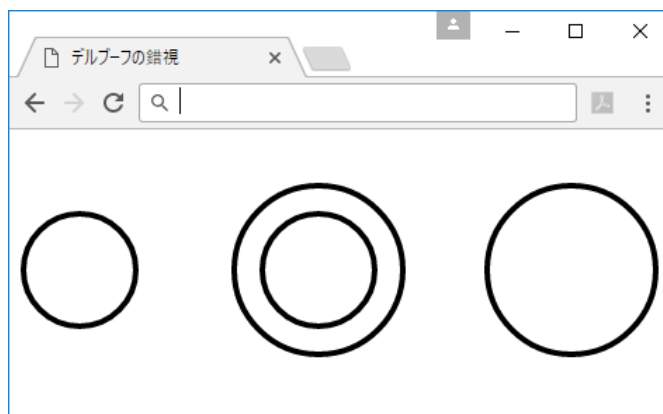


図 2.16: デルブーフの錯視

2.6 グラデーション

今までは長方形の内部を単一の色で塗りつぶしました。これ以外に色が順に変化するグラデーションという塗り方もあります。SVG では 2 種類のグラデーションが利用できます。ここでは簡単なグラデーションの定義と使い方だけを紹介します。

グラデーションは `<defs>` 要素の中で定義し、後から参照するための属性 `id` をつけます。

2.6.1 線形グラデーション

線形グラデーションの基礎 開始位置と終了位置の色を指定することで途中の色が中間の色に変わっていくものです。次の例を見てください。

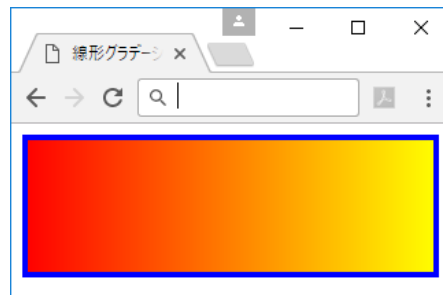


図 2.17: 線形グラデーション

SVG リスト 2.9: 線形グラデーション

```

1 <?xml version="1.0" encoding="UTF-8" ?>
2 <svg xmlns="http://www.w3.org/2000/svg"
3     xmlns:xlink="http://www.w3.org/1999/xlink"
4     height="100%" width="100%">
5   <title>線形グラデーション</title>
6   <defs>
7     <linearGradient id="Gradient1"
8       gradientUnits="objectBoundingBox">
9       <stop stop-color="red" offset="0%" />
10      <stop stop-color="yellow" offset="100%" />
11    </linearGradient>
12  </defs>
13  <g transform="translate(10,10)">
14    <rect x="0" y="0" width="300" height="100"
15      stroke-width="4" stroke="blue"
16      fill="url(#Gradient1)" />
17  </g>
18 </svg>

```

- グラデーションのパターンは<defs> 要素のなかで定義します。
- 7 行目が線形の定義の開始を示す<linearGradient> 要素です。この要素には次のような属性が与えられています。
 - id 後で参照するための名称を定義するものです。16 行目で参照されています。id の属性値は他のものと重複してはいけません。
 - gradientUnits はグラデーションをどの座標系で指定するかを表します。ここではオブジェクトの座標系で決めることを示す objectBoundingBox を指定しています。このほかには userSpaceOnUse があります。グラデーションを適用するオブジェクトをとりまく座標系を基準にします。この違いについては 30 ページ以降で説明します。
- 9 行目と10 行目にグラデーションの特定の位置に対する場所(offset)とそこでの色(stop-color)を<stop> 要素で定義しています。ここでは開始位置 (offset="0%") の色を赤に、終了位置 (offset="100%") の色を黄色に指定しています。

したがって、このグラデーションは左の赤から右に行くにしたがい黄色へと変化することになります。なお、`<stop>` 要素は途中の値も指定できるので2つより多くてもかまいません。

- `<linearGradient>` 要素のなかで別の要素を宣言しているのでこの要素に対する終了を示す `</linearGradient>` が必要です (11 行目)。
- 12 行目で `</defs>` の宣言を終了しています。
- 14 行目から16 行目で長方形のオブジェクトを宣言しています。 `fill` の値はグラデーションの定義を参照するために `url(#Gradient1)` と表しています。#の後に id で定義したラベルを用いていることに注意してください。

問題 2.9 図 2.18 はマッハバンド錯視 [32, 99 ページ] と呼ばれています。この図では左 1/3 の位置から右 1/3 の間だけにグラデーションを付けているだけです。グラデーションの開始の位置では少し明るく色が強調されて見えています。

この図を SVG で作成しなさい。また、色やグラデーションの位置をいろいろ変えてどのように見えるか確認しなさい。

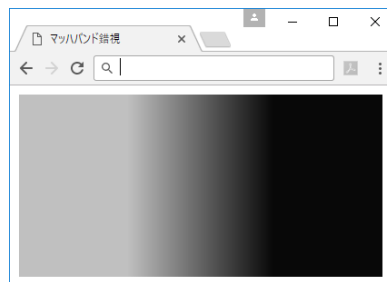


図 2.18: マッハバンド錯視

問題 2.10 図 2.19 はザバーニョの錯視と呼ばれています。グラデーションがついた長方形を 90° ずつ回転したものを並べただけですが中央部がより明るく見えます。

この図を SVG で作成しなさい。また、色やグラデーションをいろいろ変えてどのように見えるか確認しなさい。

線形グラデーションの向きを変える

図 2.20 はグラデーションが左上から右下に変化しています。

リスト 2.10 はこの図を描くものです。

SVG リスト 2.10: 傾いた方向の線形グラデーション

```

1 <?xml version="1.0" encoding="UTF-8" ?>
2 <svg xmlns="http://www.w3.org/2000/svg"
3     xmlns:xlink="http://www.w3.org/1999/xlink"

```

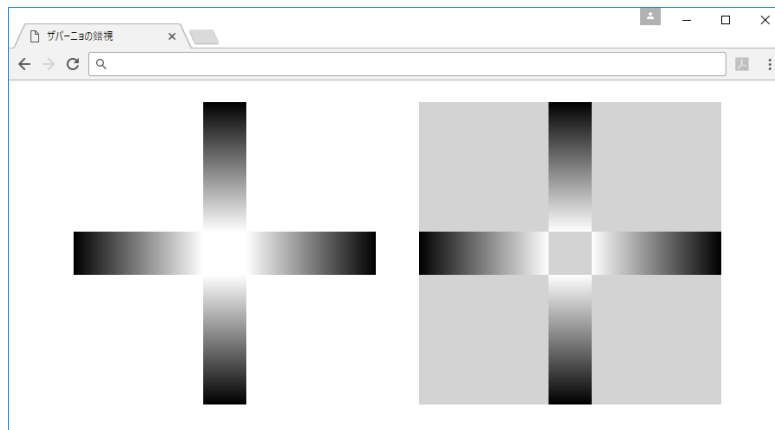



図 2.19: ザバーニョの錯視

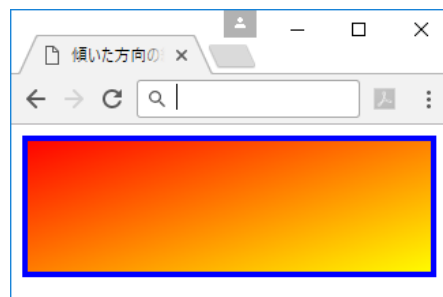


図 2.20: 傾いた方向の線形グラデーション

```

4      height="100%" width="100%">
5  <title>傾いた方向の線形グラデーション</title>
6  <defs>
7      <linearGradient id="Gradient1" gradientUnits="objectBoundingBox"
8          x1="0%" y1="0%" x2="100%" y2="100%">
9          <stop stop-color="red" offset="0%" />
10         <stop stop-color="yellow" offset="100%" />
11     </linearGradient>
12 </defs>
13 <g transform="translate(10,10)">
14     <rect x="0" y="0" width="300" height="100"
15         stroke-width="4" stroke="blue" fill="url(#Gradient1)" />
16 </g>
17 </svg>

```

このリストとリスト 2.9 の違いは8 行目の記述が追加してあるだけです。x1 と y1 でグラデーションの開始位置を指定できます。ここではともに 0% なので左上が指定されることになります。x2 と y2 でグラデーションの開始位置を指定できます。ここではともに 100% なので右下が指定されるこ

とになります。したがって、グラデーションの方向が左上から右下に向かうことになるわけです。

問題 2.11 リスト 2.10 について次のことを調べなさい。

1. x1、y1、x2 と y2 の値をいろいろ変えたときグラデーションの向きがどのように変わるか
2. 長方形の縦横比が異なったものにたいしてグラデーションがどのように変化するか
3. グラデーションの方向が左上から右下 45° の方向になるようにすること

問題 2.12 図 2.21 はコフカリング [32, 90 ページ] と呼ばれています。背景の明度に差がある部分を中央部のグラデーションの部分が隠れています。同じ色の円の縁取りが明るさの違う色に見えます。グラデーションでなくても縁取りと同じ色でなければ明度に差があるように見えることも確認しなさい。

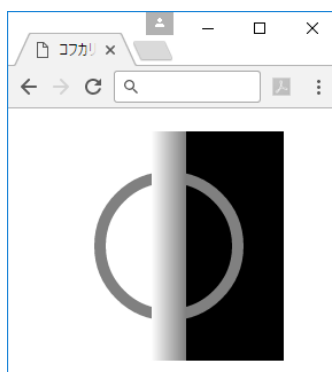


図 2.21: コフカリング

問題 2.13 図 2.22 はひし形を用いたクレイク・オブライエン効果と呼ばれています ([37, 58 ページ 図 6.4])。ひし形を塗っているグラデーションはどこも同じですが下の方が明るく見えます。

gradientUnits の値の違い gradientUnits の値として objectBoundingBox と userSpaceOnUse があることはすでに説明しました。図 2.23 はこの違いを説明するためのものです。色の変化を見やすくするために色の変化が完全に別な色になるようなグラデーションを付けています。

SVG リスト 2.11: gradientUnits の値の違い

```

1 <?xml version="1.0" encoding="UTF-8" ?>
2 <svg xmlns="http://www.w3.org/2000/svg"
3     xmlns:xlink="http://www.w3.org/1999/xlink"
4     height="100%" width="100%">
5   <title>線形グラデーションの gradientUnits の違い</title>
6   <defs>
7     <linearGradient id="GradBB" gradientUnits="objectBoundingBox">
8       <stop stop-color="red" offset="0%" />
9       <stop stop-color="red" offset="20%" />

```

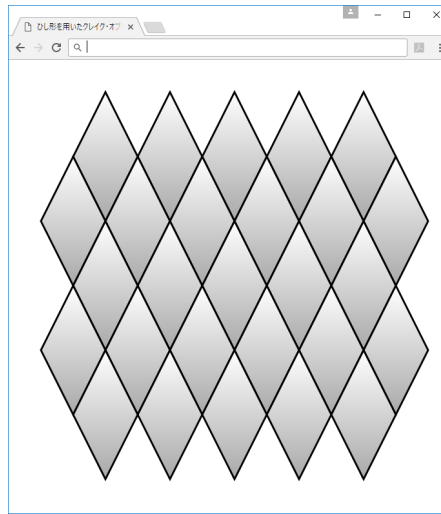


図 2.22: ひし形を用いたクレイク・オブライエン効果

```

10     <stop stop-color="yellow" offset="20%" />
11     <stop stop-color="yellow" offset="80%" />
12     <stop stop-color="red"    offset="80%" />
13     <stop stop-color="red"    offset="100%" />
14 </linearGradient>
15 <linearGradient id="GradUS1" gradientUnits="userSpaceOnUse"
16   x1="0%" y1="0%" x2="100%" y2="0%">
17   <stop stop-color="red"    offset="0%" />
18   <stop stop-color="red"    offset="20%" />
19   <stop stop-color="yellow" offset="20%" />
20   <stop stop-color="yellow" offset="80%" />
21   <stop stop-color="red"    offset="80%" />
22   <stop stop-color="red"    offset="100%" />
23 </linearGradient>
24 <linearGradient id="GradUS2" gradientUnits="userSpaceOnUse"
25   x1="0" y1="0" x2="300" y2="0" >
26   <stop stop-color="red"    offset="0%" />
27   <stop stop-color="red"    offset="20%" />
28   <stop stop-color="yellow" offset="20%" />
29   <stop stop-color="yellow" offset="80%" />
30   <stop stop-color="red"    offset="80%" />
31   <stop stop-color="red"    offset="100%" />
32 </linearGradient>
33 <rect id="RL" width="300" height="50" stroke-width="4" stroke="black"/>
34 <rect id="RS" width="100" height="50" stroke-width="4" stroke="black"/>
35 </defs>
36 <g transform="translate(20,20)">
37   <use xlink:href="#RL" x="0" y="0" fill="url(#GradBB)"/>
38   <use xlink:href="#RS" x="0" y="75" fill="url(#GradBB)"/>
39   <use xlink:href="#RS" x="200" y="75" fill="url(#GradBB)"/>
40 </g>
41 <g transform="translate(20,170)">

```

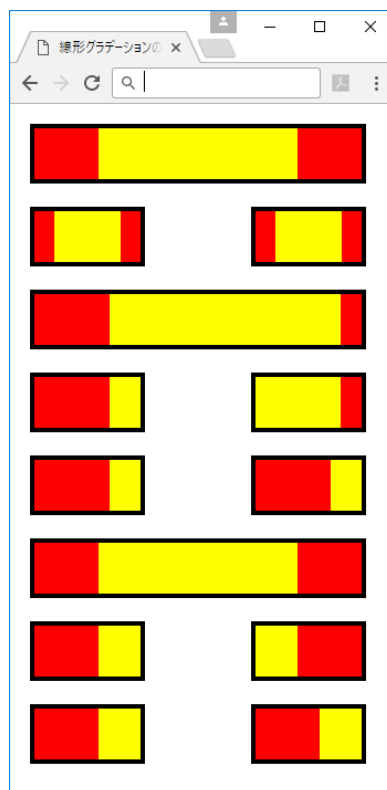


図 2.23: 線形グラデーションの gradientUnits の違い

```

42     <use xlink:href="#RL" x="0" y="0" fill="url(#GradUS1)" />
43     <rect x="0" y="75" width="100" height="50"
44           stroke-width="4" stroke="black" fill="url(#GradUS1)" />
45     <rect x="200" y="75" width="100" height="50"
46           stroke-width="4" stroke="black" fill="url(#GradUS1)" />
47     <use xlink:href="#RS" x="0" y="150" fill="url(#GradUS1)" />
48     <use xlink:href="#RS" x="200" y="150" fill="url(#GradUS1)" />
49 </g>
50 <g transform="translate(20,395)">
51     <use xlink:href="#RL" x="0" y="0" fill="url(#GradUS2)" />
52     <rect x="0" y="75" width="100" height="50"
53           stroke-width="4" stroke="black" fill="url(#GradUS2)" />
54     <rect x="200" y="75" width="100" height="50"
55           stroke-width="4" stroke="black" fill="url(#GradUS2)" />
56     <use xlink:href="#RS" x="0" y="150" fill="url(#GradUS2)" />
57     <use xlink:href="#RS" x="200" y="150" fill="url(#GradUS2)" />
58 </g>
59 </svg>

```

7行目から14行目では gradientUnits の値が objectBoundingBox となる線形グラデーションを定義しています。

```

7      <linearGradient id="GradBB" gradientUnits="objectBoundingBox">
8          <stop stop-color="red"      offset="0%" />
9          <stop stop-color="red"      offset="20%" />
10         <stop stop-color="yellow"    offset="20%" />
11         <stop stop-color="yellow"    offset="80%" />
12         <stop stop-color="red"      offset="80%" />
13         <stop stop-color="red"      offset="100%" />
14     </linearGradient>

```

- 色の变化位置を明確にするために同じ位置で二つの色を定義しています (9 行目と10 行目、11 行目と12 行目)。これにより左から 20% までの位置は赤に、そこから 80% までは黄色に、そして残りの部分は再び赤に塗られます。
- 15 行目から23 行目では gradientUnits の値が userSpaceOnUse となる線形グラデーションを定義しています。このグラデーションは16 行目で x1、y1、x2 と y2 を割合 (%) で定義しています。グラデーションの割合は前と同じです。
- 24 行目から32 行目でも gradientUnits の値が userSpaceOnUse となる線形グラデーションを定義しています。このグラデーションでは25 行目で x1、y1、x2 と y2 を数値で定義しています。このグラデーションの割合も前と同じです。
- 33 行目と34 行目ではグラデーションをつける二種類の長方形を定義しています。
- 初めのグラデーションを付けているグループは上から 2 つ並んでいるものです。

```

36     <g transform="translate(20,20)">
37         <use xlink:href="#RL" x="0"    y="0"    fill="url(#GradBB)"/>
38         <use xlink:href="#RS" x="0"    y="75"   fill="url(#GradBB)"/>
39         <use xlink:href="#RS" x="200"  y="75"   fill="url(#GradBB)"/>
40     </g>

```

- これらの長方形は33 行目と34 行目で定義されたものを<use> 要素を用いて利用しています。
- これらのグラデーションは gradientUnits の値が objectBoundingBox となっているので色が塗られるオブジェクトの位置が基準になっています。したがって、長方形の幅や位置に関係なくグラデーション全体が指定された割合でつくことになります。
- 次の 3 行にわたって並んでいる長方形は15 行目から23 行目で指定したグラデーションで塗られています。

```

41     <g transform="translate(20,170)">
42         <use xlink:href="#RL" x="0" y="0" fill="url(#GradUS1)" />
43         <rect x="0" y="75" width="100" height="50"
44             stroke-width="4" stroke="black" fill="url(#GradUS1)" />

```

```

45     <rect x="200" y="75" width="100" height="50"
46         stroke-width="4" stroke="black" fill="url(#GradUS1)" />
47     <use xlink:href="#RS" x="0" y="150" fill="url(#GradUS1)" />
48     <use xlink:href="#RS" x="200" y="150" fill="url(#GradUS1)" />
49 </g>

```

- 2 回目のグループの小さい長方形の色の变化の位置がその上の長方形と同じです。この列の位置の基準が41 行目にある<g> 要素で規定されている (userSpaceOnUse) からです。
- 3 回目のグループの小さい長方形は前に定義した小さな長方形を<use> 要素で引用しています。この場合にはこのなかで新しい基準が用いられるので両方とも左端が赤になっています。
- 最後のグループは24 行目から32 行目で定義したグラディエーションで塗られています。

```

50 <g transform="translate(20,395)">
51     <use xlink:href="#RL" x="0" y="0" fill="url(#GradUS2)" />
52     <rect x="0" y="75" width="100" height="50"
53         stroke-width="4" stroke="black" fill="url(#GradUS2)" />
54     <rect x="200" y="75" width="100" height="50"
55         stroke-width="4" stroke="black" fill="url(#GradUS2)" />
56     <use xlink:href="#RS" x="0" y="150" fill="url(#GradUS2)" />
57     <use xlink:href="#RS" x="200" y="150" fill="url(#GradUS2)" />
58 </g>

```

- この場合には上の二つの色の变化位置は同じです。また、一番最後の行はやはり小さな長方形の左端が赤くなっています。
- 2 番目のグループと3 番目のグループでは色の变化の場所が少し異なります。これは2 番目のグループの右端の基準がブラウザ画面の右端になっているためです。それが証拠に、ブラウザの画面の横幅を変えると2 番目のグループだけが色に変化が起こります。

問題 2.14 図 2.23 でブラウザの幅を変化させたとき、グラデーションがどのように変化するか調べ、その理由を考えなさい。

2.6.2 放射グラデーション

一点を中心として順次色の变化がつく放射グラデーションを表す<radialGradient> 要素があります。放射グラデーションには次の属性があります。

これらの属性の値は gradientUnits の値により解釈が異なります。userSpaceOnUse のときは数値がそのまま採用され、objectBoundingBox のときは使用されるオブジェクトの大きさに対する割合を意味します。次の二つの例を見比べてください。なお、これらの例では放射グラデーションの端を示すために行目から行目で円の縁を描いています (fill の値が none になっていることに注意すること)。

表 2.5: 放射グラデーションの属性

属性名	意味	値
cx	放射グラデーションの終了位置の円の中心の x 座標	数値または割合
cy	放射グラデーションの終了位置の円の中心の y 座標	数値または割合
r	放射グラデーションの終了位置の円の半径	数値または割合
fx	放射グラデーションの中心の x 座標	数値または割合
fy	放射グラデーションの中心の y 座標	数値または割合

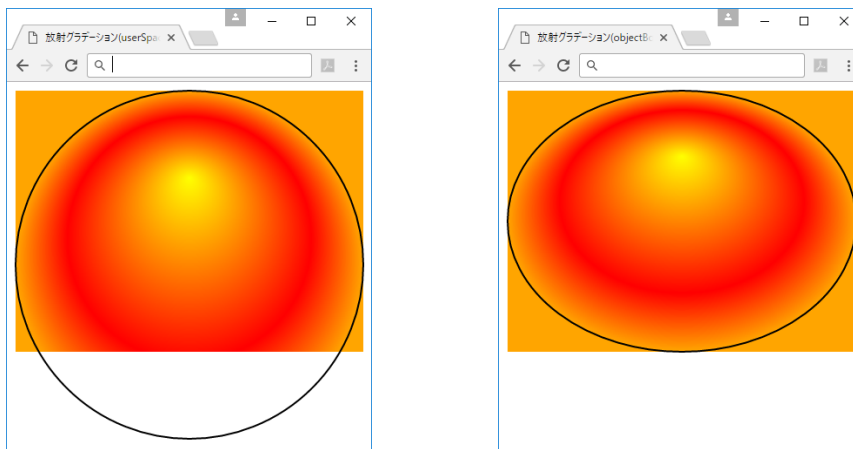


図 2.24: 放射グラデーション (userSpaceOnUse(左) と objectBoundingBox(右))

SVG リスト 2.12: 放射グラデーション (userSpaceOnUse の場合)

```

1  <?xml version="1.0" encoding="UTF-8" ?>
2  <svg xmlns="http://www.w3.org/2000/svg"
3      xmlns:xlink="http://www.w3.org/1999/xlink"
4      height="100%" width="100%">
5    <title>放射グラデーション (userSpaceOnUse の場合)</title>
6    <defs>
7      <radialGradient id="radGradient2" gradientUnits="userSpaceOnUse"
8        cx="200" cy="200" r="200" fx="200" fy="100" >
9        <stop stop-color="yellow" offset="0%"/>
10       <stop stop-color="red" offset="70%"/>
11       <stop stop-color="orange" offset="100%"/>
12     </radialGradient>
13   </defs>
14   <g transform="translate(10,10)">
15     <rect x="0" y="0" width="400" height="300" fill="url(#radGradient2)" />
16     <circle cx="200" cy="200" r="200"
17       stroke-width="2" stroke="black" fill="none"/>
18   </g>
19 </svg>

```

- 7 行目から12 行目で放射グラデーションを定義しています。
- ここでグラデーションをする基準の座標系を `userSpaceOnUse` としています。
- グラデーションの終了位置を示す円の位置と大きさとグラディエーションの開始位置は8 行目で指定しています。ここではすべての値は数値で指定しています。
- 9 行目から11 行目ではグラデーションの途中の色を線形グラデーションと同じ方法で指定しています。
- 15 行目で定義している長方形の内部をここで定義したグラディエーションで塗っています。終了位置の円の外側は最後の色をそのまま使うのがデフォルトです。

SVG リスト 2.13: 放射グラデーション (objectBoundingBox の場合)

```

1  <?xml version="1.0" encoding="UTF-8" ?>
2  <svg xmlns="http://www.w3.org/2000/svg"
3      xmlns:xlink="http://www.w3.org/1999/xlink"
4      height="100%" width="100%">
5      <title>放射グラデーション (objectBoundingBox の場合)</title>
6      <defs>
7          <radialGradient id="radGradient2" gradientUnits="objectBoundingBox"
8              cx="50%" cy="50%" r="50%" fx="50%" fy="25%" >
9              <stop stop-color="yellow" offset="0%" />
10             <stop stop-color="red" offset="70%" />
11             <stop stop-color="orange" offset="100%" />
12         </radialGradient>
13     </defs>
14     <g transform="translate(10,10)">
15         <rect x="0" y="0" width="400" height="300" fill="url(#radGradient2)" />
16         <ellipse cx="200" cy="150" rx="200" ry="150"
17             stroke-width="2" stroke="black" fill="none"/>
18     </g>
19 </svg>

```

- 7 行目から 12 行目で放射グラデーションを定義しています。
- ここでグラデーションをする基準の座標系を `objectBoundingBox` としています。
- グラデーションの終了位置を示す円の位置と大きさとグラディエーションの開始位置はすべて割合で指定しています (7 行目)。
- 9 行目から 11 行目ではグラデーションの途中の色を線形グラデーションと同じかたちで指定しています。

問題 2.15 図 2.25 はクレイク・オブライエン効果とよばれるものです。内側にある境界部分の黒が内部の白を外部よりより白く見えさせています。放射グラディエーションを利用してこの図を描きなさい。また、色を変えて同じような図を作成した場合にはどのようなになるか調べなさい。

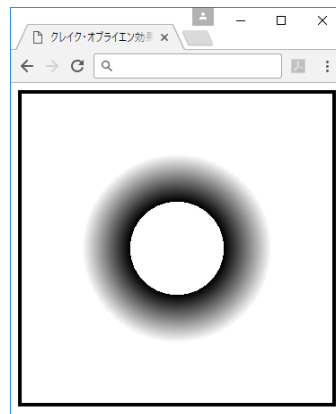


図 2.25: クレイク・オブライエン効果

2.7 不透明度

図形に対し不透明度を指定できます⁴。不透明度を設定した図形はその設定した値の応じて色合いが薄くなり、その下にある図形が見えるようになります。不透明度が 1 では下の図形がまったく見えず、0 では設定された図形がまったく見えなくなります。不透明度が設定できる属性は表 2.6 を参照してください。

表 2.6: 不透明度の種類

設定できる要素	設定のための属性名	説明
図形一般	opacity	対象の図形全体に不透明度が設定される。
図形一般	stroke-opacity	図形の属性 stroke に設定される。
図形一般	fill-opacity	図形の属性 fill に設定される。
<stop> 要素	stop-opacity	不透明度のグラデーションが設定できる。

図 2.26 は円の内部の不透明度を 0.2 に設定して重ね合わせたものです。

SVG リスト 2.14: 円の内部に不透明度を設定した例

```

1 <?xml version="1.0" encoding="UTF-8" ?>
2 <svg xmlns="http://www.w3.org/2000/svg"
3     xmlns:xlink="http://www.w3.org/1999/xlink"
4     height="100%" width="100%">
5   <title>円の内部に不透明度を設定した例</title>
6   <defs>
7     <circle id="Circle" cx="50" cy="0" r="100"/>
8     <g id="Fig0" >
9       <use xlink:href="#Circle"/>

```

⁴不透明度はアルファ値とかアルファチャンネルと呼ばれることがあります。ここでは opacity の意味を利用して不透明度と呼ぶことにします。

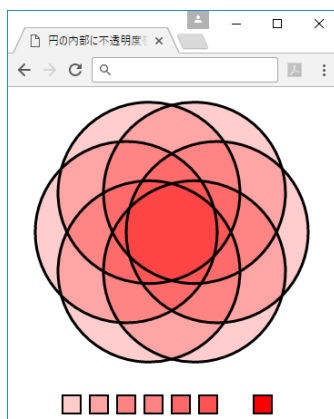


図 2.26: 円の内部に不透明度を設定した例

```

10     <g transform="rotate(60)">
11         <use xlink:href="#Circle"/>
12     </g>
13     <g transform="rotate(120)">
14         <use xlink:href="#Circle"/>
15     </g>
16 </g>
17 <g id="Fig">
18     <use xlink:href="#Fig0"/>
19     <g transform="rotate(180)">
20         <use xlink:href="#Fig0"/>
21     </g>
22 </g>
23 </defs>
24 <g transform="translate(180,160)">
25     <use xlink:href="#Fig" fill-opacity="0.2" fill="red" stroke="none"/>
26     <use xlink:href="#Fig" stroke-width="3" stroke="black" fill="none"/>
27     <rect x="-120" y="180" width="20" height="20"
28         fill="rgb(100%,80%,80%)" stroke-width="2" stroke="black"/>
29     <rect x="-90" y="180" width="20" height="20"
30         fill="rgb(100%,64%,64%)" stroke-width="2" stroke="black"/>
31     <rect x="-60" y="180" width="20" height="20"
32         fill="rgb(100%,51.2%,51.2%)" stroke-width="2" stroke="black"/>
33     <rect x="-30" y="180" width="20" height="20"
34         fill="rgb(100%,51.2%,51.2%)" stroke-width="2" stroke="black"/>
35     <rect x="0" y="180" width="20" height="20"
36         fill="rgb(100%,40.96%,40.96%)" stroke-width="2" stroke="black"/>
37     <rect x="30" y="180" width="20" height="20"
38         fill="rgb(100%,32.768%,32.768%)" stroke-width="2" stroke="black"/>
39     <rect x="90" y="180" width="20" height="20"
40         fill="rgb(100%,0%,0%)" stroke-width="2" stroke="black"/>
41 </g>
42 </svg>

```

- 7行目でこの図形を描くための共通の円を定義しています。この円には fill や stroke など通常の属性がまったく指定されていないことに注意してください。
- この円を 60° ずつ回転して全体で 6 個並べた図形を作成するためにまず、半分だけ <use> 要素を用いて作成します (8 行目から 16 行目)。
- これを 180° 回転したものと組み合わせて図形の雛形を作成します (17 行目から 22 行目)。
- 25 行目で fill-opacity、fill、stroke の値を設定しています。引用された図形ではこの値が採用されます。
- 26 行目では stroke-width、stroke、fill の値を設定しています。
- opacity がある図形の色は

$\text{opacityが定義された図形の色} \times \text{この図形の色} + (1 - \text{この図形の opacity}) \times \text{この図形の下にある色}$

という計算式で求められます。背景が白なのでこの図形ではすべての位置で RGB の赤の成分は 100% です。青と緑の成分はひとつ重なるごとに 0.8 倍されます。

- 具体的に rgb で指定した色に塗った正方形をこの図形の下に順番に描いています (27 行目から 40 行目)。なお、一番右は赤に塗っています。

問題 2.16 リスト 2.14 に関して次の問いに答えなさい。

1. fill と stroke を別に設定している図形を二つ重ねている理由はなにか。
2. 図のある部分の一番下をを青で塗ったらどのような図形になるか
3. 円を放射グラデーションで塗ったらどのようなになるか
4. 円を放射グラデーションに stop-opacity を入れたらどのようなになるか

問題 2.17 図 2.27 はピラミッドの稜線とよばれています ([37, カラー図版 13])。色の濃度が異なる正方形がいくつか並んでいますが、頂点に沿って直線が描いてあるように見えます。左の二つは色を RGB 値で直接指定し、右の二つは不透明度を利用して一番下に指定した色を透かして見せています。この図を作成しなさい。

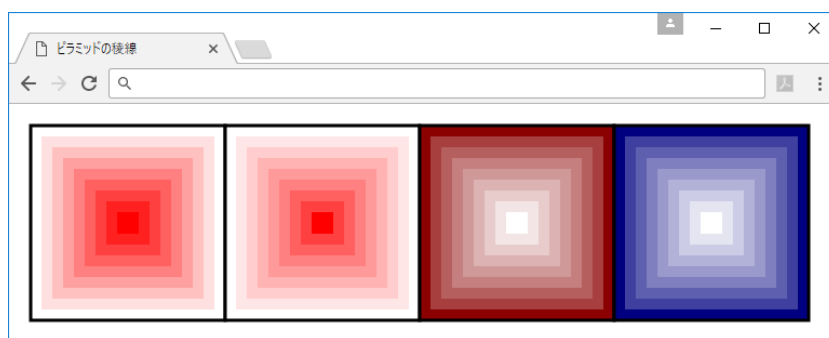








































































図 2.27: ピラミッドの稜線

付 録 A SVG の色名








































次の表は [27] にある SVG で利用できる色名とその rgb 値の一覧表です。

色	色名	rgb 値	16 進表示
	aliceblue	rgb(240, 248, 255)	#F0F8FF
	antiquewhite	rgb(250, 235, 215)	#FAEBD7
	aquamarine	rgb(127, 255, 212)	#7FFFD4
	aqua	rgb(0, 255, 255)	#00FFFF
	azure	rgb(240, 255, 255)	#F0FFFF
	beige	rgb(245, 245, 220)	#F5F5DC
	bisque	rgb(255, 228, 196)	#FFE4C4
	black	rgb(0, 0, 0)	#000000
	blanchedalmond	rgb(255, 235, 205)	#FFEBCD
	blueviolet	rgb(138, 43, 226)	#8A2BE2
	blue	rgb(0, 0, 255)	#0000FF
	brown	rgb(165, 42, 42)	#A52A2A
	burlywood	rgb(222, 184, 135)	#DEB887
	cadetblue	rgb(95, 158, 160)	#5F9EA0
	chartreuse	rgb(127, 255, 0)	#7FFF00
	chocolate	rgb(210, 105, 30)	#D2691E
	coral	rgb(255, 127, 80)	#FF7F50
	cornflowerblue	rgb(100, 149, 237)	#6495ED
	cornsilk	rgb(255, 248, 220)	#FFF8DC
	crimson	rgb(220, 20, 60)	#DC143C
	cyan	rgb(0, 255, 255)	#00FFFF
	darkblue	rgb(0, 0, 139)	#00008B
	darkcyan	rgb(0, 139, 139)	#008B8B
	darkgoldenrod	rgb(184, 134, 11)	#B8860B
	darkgray	rgb(169, 169, 169)	#A9A9A9
	darkgreen	rgb(0, 100, 0)	#006400
	darkgrey	rgb(169, 169, 169)	#A9A9A9
	darkkhaki	rgb(189, 183, 107)	#BDB76B
	darkmagenta	rgb(139, 0, 139)	#8B008B
	darkolivegreen	rgb(85, 107, 47)	#556B2F
	darkorange	rgb(255, 140, 0)	#FF8C00







































次のページへ

色	色名	rgb 値	16 進表示
	darkorchid	rgb(153, 50, 204)	#9932CC
	darkred	rgb(139, 0, 0)	#8B0000
	darksalmon	rgb(233, 150, 122)	#E9967A
	darkseagreen	rgb(143, 188, 143)	#8FBC8F
	darkslateblue	rgb(72, 61, 139)	#483D8B
	darkslategray	rgb(47, 79, 79)	#2F4F4F
	darkslategrey	rgb(47, 79, 79)	#2F4F4F
	darkturquoise	rgb(0, 206, 209)	#00CED1
	darkviolet	rgb(148, 0, 211)	#9400D3
	deeppink	rgb(255, 20, 147)	#FF1493
	deepskyblue	rgb(0, 191, 255)	#00BFFF
	dimgray	rgb(105, 105, 105)	#696969
	dimgrey	rgb(105, 105, 105)	#696969
	dodgerblue	rgb(30, 144, 255)	#1E90FF
	firebrick	rgb(178, 34, 34)	#B22222
	floralwhite	rgb(255, 250, 240)	#FFFAF0
	forestgreen	rgb(34, 139, 34)	#228B22
	fuchsia	rgb(255, 0, 255)	#FF00FF
	gainsboro	rgb(220, 220, 220)	#DCDCDC
	ghostwhite	rgb(248, 248, 255)	#F8F8FF
	goldenrod	rgb(218, 165, 32)	#DAA520
	gold	rgb(255, 215, 0)	#FFD700
	gray	rgb(128, 128, 128)	#808080
	greenyellow	rgb(173, 255, 47)	#ADFF2F
	green	rgb(0, 128, 0)	#008000
	grey	rgb(128, 128, 128)	#808080
	honeydew	rgb(240, 255, 240)	#F0FFF0
	hotpink	rgb(255, 105, 180)	#FF69B4
	indianred	rgb(205, 92, 92)	#CD5C5C
	indigo	rgb(75, 0, 130)	#4B0082
	ivory	rgb(255, 255, 240)	#FFFFF0
	khaki	rgb(240, 230, 140)	#F0E68C
	lavenderblush	rgb(255, 240, 245)	#FFF0F5
	lavender	rgb(230, 230, 250)	#E6E6FA
	lawngreen	rgb(124, 252, 0)	#7CFC00
	lemonchiffon	rgb(255, 250, 205)	#FFFACD
	lightblue	rgb(173, 216, 230)	#ADD8E6
	lightcoral	rgb(240, 128, 128)	#F08080
	lightcyan	rgb(224, 255, 255)	#E0FFFF

次のページへ

色	色名	rgb 値	16 進表示
	lightgoldenrodyellow	rgb(250, 250, 210)	#FAFAD2
	lightgray	rgb(211, 211, 211)	#D3D3D3
	lightgreen	rgb(144, 238, 144)	#90EE90
	lightgrey	rgb(211, 211, 211)	#D3D3D3
	lightpink	rgb(255, 182, 193)	#FFB6C1
	lightsalmon	rgb(255, 160, 122)	#FFA07A
	lightseagreen	rgb(32, 178, 170)	#20B2AA
	lightskyblue	rgb(135, 206, 250)	#87CEFA
	lightslategray	rgb(119, 136, 153)	#778899
	lightslategrey	rgb(119, 136, 153)	#778899
	lightsteelblue	rgb(176, 196, 222)	#B0C4DE
	lightyellow	rgb(255, 255, 224)	#FFFFE0
	limegreen	rgb(50, 205, 50)	#32CD32
	lime	rgb(0, 255, 0)	#00FF00
	linen	rgb(250, 240, 230)	#FAF0E6
	magenta	rgb(255, 0, 255)	#FF00FF
	maroon	rgb(128, 0, 0)	#800000
	mediumaquamarine	rgb(102, 205, 170)	#66CDAA
	mediumblue	rgb(0, 0, 205)	#0000CD
	mediumorchid	rgb(186, 85, 211)	#BA55D3
	mediumpurple	rgb(147, 112, 219)	#9370DB
	mediumseagreen	rgb(60, 179, 113)	#3CB371
	mediumslateblue	rgb(123, 104, 238)	#7B68EE
	mediumspringgreen	rgb(0, 250, 154)	#00FA9A
	mediumturquoise	rgb(72, 209, 204)	#48D1CC
	mediumvioletred	rgb(199, 21, 133)	#C71585
	midnightblue	rgb(25, 25, 112)	#191970
	mintcream	rgb(245, 255, 250)	#F5FFFA
	mistyrose	rgb(255, 228, 225)	#FFE4E1
	moccasin	rgb(255, 228, 181)	#FFE4B5
	navajowhite	rgb(255, 222, 173)	#FFDEAD
	navy	rgb(0, 0, 128)	#000080
	oldlace	rgb(253, 245, 230)	#FDF5E6
	olivedrab	rgb(107, 142, 35)	#6B8E23
	olive	rgb(128, 128, 0)	#808000
	orangered	rgb(255, 69, 0)	#FF4500
	orange	rgb(255, 165, 0)	#FFA500
	orchid	rgb(218, 112, 214)	#DA70D6
	palegoldenrod	rgb(238, 232, 170)	#EEE8AA

次のページへ

色	色名	rgb 値	16 進表示
	palegreen	rgb(152, 251, 152)	#98FB98
	paleturquoise	rgb(175, 238, 238)	#AFEEEE
	palevioletred	rgb(219, 112, 147)	#DB7093
	papayawhip	rgb(255, 239, 213)	#FFEFD5
	peachpuff	rgb(255, 218, 185)	#FFDAB9
	peru	rgb(205, 133, 63)	#CD853F
	pink	rgb(255, 192, 203)	#FFC0CB
	plum	rgb(221, 160, 221)	#DDA0DD
	powderblue	rgb(176, 224, 230)	#B0E0E6
	purple	rgb(128, 0, 128)	#800080
	red	rgb(255, 0, 0)	#FF0000
	rosybrown	rgb(188, 143, 143)	#BC8F8F
	royalblue	rgb(65, 105, 225)	#4169E1
	saddlebrown	rgb(139, 69, 19)	#8B4513
	salmon	rgb(250, 128, 114)	#FA8072
	sandybrown	rgb(244, 164, 96)	#F4A460
	seagreen	rgb(46, 139, 87)	#2E8B57
	seashell	rgb(255, 245, 238)	#FFF5EE
	sienna	rgb(160, 82, 45)	#A0522D
	silver	rgb(192, 192, 192)	#C0C0C0
	skyblue	rgb(135, 206, 235)	#87CEEB
	slateblue	rgb(106, 90, 205)	#6A5ACD
	slategray	rgb(112, 128, 144)	#708090
	slategrey	rgb(112, 128, 144)	#708090
	snow	rgb(255, 250, 250)	#FFFAFA
	springgreen	rgb(0, 255, 127)	#00FF7F
	steelblue	rgb(70, 130, 180)	#4682B4
	tan	rgb(210, 180, 140)	#D2B48C
	teal	rgb(0, 128, 128)	#008080
	thistle	rgb(216, 191, 216)	#D8BFD8
	tomato	rgb(255, 99, 71)	#FF6347
	turquoise	rgb(64, 224, 208)	#40E0D0
	violet	rgb(238, 130, 238)	#EE82EE
	wheat	rgb(245, 222, 179)	#F5DEB3
	whitesmoke	rgb(245, 245, 245)	#F5F5F5
	white	rgb(255, 255, 255)	#FFFFFF
	yellowgreen	rgb(154, 205, 50)	#9ACD32
	yellow	rgb(255, 255, 0)	#FFFF00

付 録 B 参考文献について

残念ながら SVG のまとまった解説がある日本語の本はないようです。このテキストを書き始めたころに参考にした本は [2] と [3] です。また、途中からは [7] の図書も参考にしました。これらの本のリンク先は amazon.co.jp です。平成 18 年 4 月現在で検索した書籍のページにジャンプします。

なお、SVG のフルの規格を調べるのであればやはり元の文書 [27] を参考にするしかありません。PDF 版をダウンロードしておけば文書内にリンクが張ってあるので比較的簡単に該当箇所へジャンプできます (700 ページ以上のあります)。

[2] の図書については次のような特徴があります。

- 発行年次が 2002 年と少し古く SVG1.0 に基づいた記述のようです。
- SVG の図形やテキストについての記述が詳しいです。
- フィルターについても基本的なところが詳しくかかれています。本テキストを作成するに当たっては参考になった部分が多いです。
- インターラクティブな SVG のところもかなり詳しく書かれています。

[3] の図書については次のような特徴があります。

- 似たような SVG の例が多いので前半部は飛ばして読めます。。
- JavaScript を用いた部分の解説がかなり多いのでインターラクティブな SVG を作成したい人には参考になるでしょう。
- その反面、フィルタの解説が少ないので物足りなく感じるかもしれません。

[7] の図書については次のような特徴があります。

- オーソドックスな SVG の解説書です。
- フィルターについては基本的なものだけです。
- フィルタの例については [27] のものと似ています。
- CSS の一覧表があります。

参考となる SVG のファイルや説明が日本語である Web サイトは検索するとたくさん見つかります。しかし、本書のように DOM を用いて SVG 文書进行操作する方法を説明しているサイトはほとんど見つかりません。

なお、今後 HTML も含めて XML ベースの文書进行操作するには DOM が基本になります。JavaScript の解説書でも最近のものは DOM を取り扱っています。

JavaScript の解説書は非常にたくさんあります。なかでも [8] は言語の解説書として手元に置いておくといいでしょう。また、姉妹書の [9] は関数の仕様を調べるのに役に立ちます。

JavaScript は注意してコーディングをすることが必要です。プログラミングスタイルも様々ですが、大規模なアプリケーションをチームで組む必要があるときなどは [31] を参考にしてください。また、より良いコーディングスタイルを身に付けておきたい方は [10] がお勧めです。また、使用を避けるべき JavaScript の点については [4] がおすすめです。この本の記述は EcmaScript のバージョン 3 に基づいているようです。現在 ECMA Script のバージョンは 2015 年に出的 6 が最新版です [5]。バージョン 5 では配列に関するいろいろなメソッドや JSON の処理が標準で備わっています。この部分が古くなっていることに注意してください。

また、本文ではそれほど解説しませんが Ajax による Web アプリケーションは標準の技術になってしまいました。

2014 年に策定された HTML5 は W3C のサイト [26] その内容を見ることができます最近のブラウザは HTML5 の機能をサポートしています。また、iPhone や iPod Touch に搭載されている Safari も HTML5 に対応してるばかりでなく、特有の機能であるマルチタッチを利用する API もあります¹。

テキスト内で簡単に触れた PHP については [20, 12] などがあります。PHP は Web 時代後の言語なので、Web 関係の処理が簡単にできる機能が言語自体に備わっています。しかし、PHP のスクリプト系の言語としての面も丁寧に解説した書籍はほとんどありません。PHP の全貌を知る唯一の情報は PHP のホームページ [38] です。

¹iPhone Javascript マルチタッチ などのキーワードで検索すれば解説しているサイトが見つかるでしょう。

関連図書

- [1] Andreas Anyuru(吉川 邦夫 訳), 実践プログラミング WebGL HTML & JavaScript による 3D グラフィックス, 翔泳社, 2012 年
- [2] Kurt Cagle, *SVG Programming: The Graphical Web*, Apress 2002
- [3] Oswald Cansepatto, *Fundamentals of SVG Programming: Concept to Source Code*, CHARLES RIVER MEDIA, INC. 2004
- [4] D. Crockford(水野 貴明 訳), JavaScript: The Good Parts 「良いパーツ」によるベストプラクティス, オライリー・ジャパン, 2008
- [5] ECMA, ECMAScript[®] 2015 Language Specification(6th edition)
<http://www.ecma-international.org/publications/standards/Ecma-262.htm>
- [6] ECMA, JSON, <http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf>, 2013
- [7] J. David Eisenberg, *SVG Essentials*, O'Reilly, 2002
- [8] David Flanagan(村上列 訳) JavaScript 第 6 版, オライリー・ジャパン, 2012
- [9] David Flanagan(木下哲也, 福龍興業 訳) JavaScript クイックリファレンス 第 6 版, オライリー・ジャパン, 2012
- [10] David Herman(吉川 邦夫 監修, 訳), Effective JavaScript JavaScript を使うときに知っておきたい 68 の作法, 翔泳社, 2013 年
- [11] Peter Gasston, CSS3 開発者ガイド 第 2 版 モダン Web デザインのスタイル設計, オライリー・ジャパン, 2015
- [12] Rasmus Lerdorf, Kevin Tatroe, Peter MacIntyre(高木正弘 訳) プログラミング PHP 第 2 版, オライリー・ジャパン, 2007
- [13] Stoyan Stefanov(水野貴明, 渋谷よしき 訳) オブジェクト指向 JavaScript, アスキー・メディアワークス, 2012
- [14] Document Object Model (DOM) Level 3 Events Specification, W3C Technical Reports and Publications, 2007, <http://www.w3.org/TR/DOM-Level-3-Events/>
- [15] Ray Erik T.(宮下尚、牧野聡、立堀道明訳) 入門 XML 第 2 版, オライリー・ジャパン 2004

- [16] Donald E. Knuth, *The METAFONTbook*, Computers & typesetting /C, Addison-Wesley, 2000
- [17] Hajime Ōuchi, *Japanese Optical and Geometrical Art*, Dover Pub., 1977
- [18] J. O. Robinson, *The Psychology of Visual Illusion*, Dover, N. Y., 1998(originally published in 1972)
- [19] Chris Shiflett (桑村 潤, 廣川 類 訳) 入門 PHP セキュリティ, オライリージャパン, 2006
- [20] David Sklar (桑村 潤, 廣川 類 訳) 初めての PHP5, オライリージャパン, 2005
- [21] Steve Souders, (武舎 広幸, 福地 太郎, 武舎 るみ),
ハイパフォーマンス Web サイト 高速サイトを実現する 14 のルール, オライリージャパン ,
2008
- [22] W3C, Document Object Model (DOM), <http://www.w3.org/DOM>
- [23] W3C, W3C DOM4 <https://www.w3.org/TR/2015/REC-dom-20151119/>
- [24] W3C, Document Object Model(DOM) Level 2 Events Specification, W3C Recommendation, 2000
<http://www.w3.org/TR/2000/REC-DOM-Level-2-Events-20001113/>
- [25] W3C, HTML 4.01 Specification, W3C Recommendation 24 December 1999
- [26] W3C, HTML5 A vocabulary and associated APIs for HTML and XHTML,
<http://www.w3.org/TR/html5/>
- [27] W3C, Scalable Vector Graphics (SVG) 1.1 Specification,
<http://www.w3.org/TR/2011/REC-SVG11-20110816/>
- [28] W3C, HTML & CSS , <http://www.w3.org/standards/webdesign/htmlcss>
- [29] W3C, Web Storage, <https://www.w3.org/TR/webstorage/>
- [30] Nicholas C. Zakas (水野 貴明 訳) ハイパフォーマンス JavaScript , オライリージャパン 2011
- [31] Nicholas C. Zakas(豊福 剛 訳), メンテナブル JavaScript 読みやすく保守しやすい JavaScript
コードの作法, オライリー・ジャパン, 2013 年
- [32] 後藤 倬男 (編集), 田中 平八 (編集), 錯視の科学ハンドブック, 東京大学出版会 (2005/02)
- [33] 北岡明佳, 錯視入門, 朝倉書店、2010 年
- [34] 杉原 厚吉, 錯視図鑑 脳がだまされる錯覚の世界, 誠文堂新光社, 2012 年
- [35] フィービ・マクノートン (著), 駒田曜 (翻訳), 錯視芸術 (アルケミスト双書) , 創元社, 2010
- [36] 馬場 雄二, 田中 康博, 試してナットク! 錯視図典 CD-ROM 付, 講談社 (2004/12/17)
- [37] ジャック・ニニオ (鈴木幸太郎、向井智子訳) 錯視の世界 古典から CG 画像まで, 新曜社 2004 年
- [38] 日本 PHP ユーザー会, <http://www.php.gr.jp/>

付 録 C JavaScript 入門

C.1 JavaScript とは

JavaScript というと HTML 文書文書の中で取り扱われ、主にブラウザで利用される言語と思っている人が多いと思います。しかし、JavaScript の言語仕様のもととなるものは ECMA が定めている `ecmascript` と呼ばれるものです。一般のプログラミング言語と同様に、言語の構成要素が定義されています。ブラウザなどで使用される場合には、この言語を基に、ブラウザ上で定義されたオブジェクトやそれを操作する方法を用いてプログラムを組むことになります。現に、JavaScript を網羅的に解説している [8] では第 I 部が「コア JavaScript」第 II 部が「クライアントサイド JavaScript」と分けられています。最近では「サーバーサイド JavaScript」と呼ばれるものも登場しています。

これらの違いは、プログラミングを記述するための手段として JavaScript を利用することで、利用する環境に応じて別のライブラリーが用意されているために、いろいろな環境で利用が可能となっています。これは C 言語で簡単なプログラムを記述する場合には複雑なライブラリーは必要としませんが、ウィンドウを開いて実行されるプログラムにはウィンドウを開くためのライブラリーが必要になってくるとの似ています。

とはいっても、コアの JavaScript だけを使用してプログラムを組むことを解説している参考書は皆無といえます。このテキストではコアの JavaScript によるプログラミングを解説をします。とはいっても、コアの JavaScript を実行する環境はやはりブラウザとなります。ブラウザ上でどのようにコアの JavaScript を実行し、デバッグする方法を示しながら、JavaScript のプログラミング言語としての特徴をつかんでください。

C.2 データの型

JavaScript のデータ型には大きく分けてプリミティブ型と非プリミティブ型の 2 種類があります。

C.2.1 プリミティブデータ型

プリミティブ型には表 C.1 のような種類のものがあります。

変数や値の型を知りたいときは `typeof` 演算子を使います。

Number 型 JavaScript で扱う数は 64 ビット浮動小数点形式です。数を表現する方法 (数値リテラル) としては次のものがあります。

表 C.1: JavaScript のプリミティブなデータ型

型	説明
Number	浮動小数点数だけ
String	文字列型、1 文字だけのデータ型はない。ダブルクォート (") かシングルクォート (') で囲む。
Boolean	true か false の値のみ
undefined	変数の値が定義されていないことを示す
null	null という値しか取ることができない特別なオブジェクト

- 整数リテラル 10 進整数は通常通りの形式です。16 進数を表す場合は先頭に 0x か 0X をつけます。0 で始まりそのあとに x または X が来ない場合には 8 進数と解釈される場合があるので注意が必要です。
- 浮動小数点リテラル 整数部、そのあとに必要なならば小数点、小数部そのあとに指数部がある形式です。

特別な Number Number 型には次のような特別な Number が定義されている。

- Infinity 無限大を表す読み出し可能な変数である。オーバーフローした場合や 1/0 などの結果としてこの値が設定されます。
- NaN Not a Number の略である。計算ができなかった場合表す読み出し可能な変数です。文字列を数値に変換できない場合や 0/0 などの結果としてこの値が設定されます。

String 型 文字列に関する情報や操作には次のようなものがあります。

表 C.2: 文字列操作のメソッド

メンバー	説明
length	文字列の長さ
indexOf(needle[,start])	needle が与えられた文字列内にあればその位置を返す。start の引数がある場合には、指定された位置以降から調べる。見つからない場合は -1 を返す。
split(separator[,limit])	separator で与えられた文字列で与えられた文字列を分けて配列で返す。セパレーターの部分は返されない。2 番目の引数はオプションで、分割する最大数を与える。
substring(start[,end])	与えられた文字列の start から end の位置までの部分文字列を返す。end がない場合には文字列の最後までがとられる。

Bool 型 `true` と `false` の 2 つの値をとります。この 2 つは予約語です。論理式の結果としてこれらの値が設定されたり、論理値が必要なところでこれらの値に設定されます。

undefined 値が存在しないことを示す読み出し可能な変数である。変数が宣言されたのに値が設定されていない場合などはこの値に初期化されます。

null `typeof null` の値が `"object"` であることを示すように、オブジェクトが存在しないことを示す特別なオブジェクト値（であると同時にオブジェクトでもある）です。

C.2.2 配列

配列の宣言と初期化 配列を使うためには、変数を配列で初期化する必要があります。変数の宣言と同時にすることもできます。配列の初期化は次のように行います。

```
var a = [];  
var b = [1,2,3];
```

- `a` は空の配列で初期化されています。
- `b` は `b[0]=1, b[1]=2, b[2]=3` となる配列で初期化されています。

次のことに注意する必要があります。

- 配列の各要素のデータの型は同じでなくてもかまいません。
- 実行時に配列の大きさを自由に変えることができます。
- 配列の要素に配列を置くことができます。たとえば、次のような配列の初期化が可能です。

```
var a=[1,[2,3,4],"a"];
```

この例では `a[0]` は数 1、`a[1]` は配列 `[2,3,4]`、`a[2]` は文字列 `"a"` でそれぞれ初期化されています。

配列のメソッドについては C.7 で解説します。

C.3 演算子

C.3.1 代入、四則演算

数に対しては C 言語と同様の演算子が使用できます。ただし、次のことに注意する必要があります。

- +演算子は文字列の接続にも使用できます。+演算子は左右のオペランドが Number のときだけ、ふたつの数の和をもとめます。どちらかが数でもう一方が文字列の場合は数を文字列に直して、文字列の接続を行います¹。

```
1+2    => 3
1+"2"  => 12
```

- そのほかの演算子 (-*//) については文字列を数に変換してから数として計算します。
- 文字列全体が数にならない場合には変換の結果が NaN になります。次の例を見てください。

```
"2" + 3    => "23"
"2"-0 +3    => 5
"2"*3       => 6
"2"*"3"     => 6
"f" *2      => NaN
"0xf"*2     => 30
```

最後の例では"0xf"が 16 進数と解釈され (15), その値が 2 倍されています。

- 整数を整数で割った場合、割り切れなければ小数となります。

```
1/3 => 0.3333333333333333
```

C.3.2 比較演算子

JavaScript の比較演算子は通常の>、>=などが使えます。数と文字列の比較も文字列が数に変換されて比較されます。文字列同士を比較する場合は文字コード順になります。次の例を見てください。

```
>(11>2)?"true":"false";
"true"
>(11>"2")?"true":"false";
"true"
>("11">2)?"true":"false";
"true"
>("11">"2")?"true":"false";
"false"
```

初めの 3 つは数として比較されていますが、最後のものは文字コードで"1"のほうが"2"より小さいので判定が逆になっています。

¹一般にどのようなオブジェクトにも toString というメソッドが用意されていて、文字列が必要な状況ではこのメソッドによってオブジェクトが文字列に変換されます。

また、比較演算子`==`で文字列と数を比較すると文字列は数に直されて比較されます。値の型を含めて等しいかどうかを調べるためには演算子`===`(等しい) や`!==`(等しくない) を用います。特別な事情がない限り`===`や`!==`を使いましょう [4, 127 ページ]。

```
>("1"==1)?"true":"false";
"true"
>("1"===1)?"true":"false";
"false"
```

C.4 制御構造

C.4.1 if 文

`if` 文はある条件が成立したときやしなかったときにだけ実行したい場合に使用します。一般の形は次のようになります。

```
if(条件式)
```

条件式が成立したときに実行したい式

実行したい式が複数ある場合にはブロック`{}`で囲みます。特別な場合を除いては実行したい文が一つだけであってもブロックにしておくほうが良いでしょう。

2 つ目の形式は `else` があるものです。

```
if(条件式)
```

条件式が成立したときに実行したい式

```
else
```

条件式が成立しなかったときに実行したい式

`else` の部分もブロックにしておいたほうがバグの発生が防げます。

C.4.2 switch 文

`if` 文においてある条件が成立しなかった場合に、`else` のなかでさらに次の条件が成立するか調べたい場合があります。これが多くなってくると `if` 文の入れ子が深くなりすぎてプログラムが読みにくくなります。このような場合には `switch` 文を使うとプログラムが見やすくなります。`switch` 文は次のような構造をとります。

```
switch(式){
  case 値 1:
    式の値が値 1 のときに実行される
    break;
  case 値 2:
    式の値が値 2 のときに実行される
```

```

break;
.
.
.
default:
  式の値が case に現れなかった場合に実行される
}

```

ここに現れる `break` 文は `switch` の処理を中断することを意味しています。これがないとその下の部分も実行されてしまいます。いくつかの場合に処理が同じになる場合以外には使わないほうが良いでしょう。

C.4.3 for 文と while 文

同じような処理を繰り返して行うためには `for` 文や `while` 文を用います。`for` 文は次のような構造を持ちます。

`for`(初期化 ; ループの終了条件 ; 次の繰り返しのための処理)
繰り返しの処理

- 初期化のところでは繰り返しを実行する前の変数の値の初期値を設定するのが普通です。
- ループの終了条件は通常は比較演算子を書きます。
- 次の繰り返しのための準備としてはループの終了条件に現れる変数の値を変更するのが普通です。
- 初期化を実行した後、ループの終了条件がチェックされます。したがって、繰り返し処理は 1 度も実行されない場合があります。
- 初期化と次の繰り返しのための処理を複数の変数に対して行う場合にはコンマオペレータ(,)で処理を記述します。

たとえば、1 から N までの値の総和を求めるプログラムは次のように書けます。

```

for(sum = 0, i = 0; i = 1; i<= N; i++) {
  sum += i;
}

```

- `for` 文の初期化、終了条件、次の繰り返しのための処理の部分は全くなくてもかまいません。例えば 3 つの処理の部分がない `for(;;)` も文法上は許されます。この記述は無限ループを実現できます。
- `for` 文の繰り返しを途中で打ち切りたいときには `break` 文を使用します。

while 文は for 文と同様に繰り返しを行うためのものです。while 文では次のような構造を取ります。

初期化;

```
while(終了条件) {
```

```
    繰り返しの処理
```

```
  次の繰り返しのための処理
```

```
}
```

終了条件のところで空にすることはできません。

C.5 関数

C.5.1 関数の定義と呼び出し

次の例は `sum()` という関数を定義している例です。

```
function sum(a,b) {  
    var c = a + b;  
    return c;  
}
```

関数の定義は次の部分から成り立っています。

- **function キーワード**
戻り値の型を記す必要はありません。
- **関数の名前**
`function` の後にある識別名が関数の名前になります。この場合は `sum` が関数の名前になります。
- **引数のリスト**
関数名の後に `()` 内にカンマで区切られた引数を記述します。この場合は変数 `a` と `b` が与えられています。引数はなくてもかまいません。
- **関数の本体であるコードブロック**
`{ }` で囲まれた部分に関数の内容を記述します。
- **return キーワード**
関数の戻り値をこの後に記述します。戻り値がない場合には戻り値として `undefined` が返されます。

実行例 次の部分はこの関数の実行例です。

```
>sum(1,2)  
3  
>sum(1)
```

NaN

```
>sum(1,2,3)
```

3

- 引数に 1 と 2 を与えれば期待通りの結果が得られます。
- 引数に 1 だけを与えた場合、エラーが起こらず、NaN となります。これは、不足している引数 (この場合には b) に `undefined` が渡されるためです。1`undefined`+の結果は通常の計算ができないので、数ではないことを示す NaN になります。
- 引数を多く渡してもエラーが発生しません。無視されるだけです。

JavaScript の関数はオブジェクト指向で使われるポリモーフィズムをサポートしていないことがわかります。さらに、次の例で見るように同じ関数を定義してもエラーにはなりません。後の関数の定義が優先されます。

```
function sum(a, b){  
  var c = a+b;  
  return c;  
}  
function sum(a, b, c){  
  var d = a+b+c;  
  return d;  
}
```

C.5.2 仮引数への代入

仮引数に値を代入してもエラーとはなりません。仮引数の値がプリミティブなときとそうでないときとは呼び出し元における変数の値がかわります。

次の例は呼び出した関数の中で仮引数の値を変化させたときの例です。

```
function func1(a){  
  a = a*2;  
  return 0;  
}  
function func2(a){  
  a[0] *=2;  
  return 0;  
}
```

- `func1()` では仮引数 `a` の値を 2 倍しています。これを次のように実行すると、呼び出し元の変数の値には変化がないことがわかります。つまり、プリミティブな値を仮引数で渡すと値そのものが渡されます (値渡し)。

```

>a = 4;
4
>func1(a);
0
>a;
4
\end{verbatim}
\item \verb+func2()+の仮引数は配列が想定しています。この配列の先頭の値だけ 2 倍さ
れる関数です。これに配列を渡すと、戻ってきたとき配列の先頭の値
が変化しています。つまり、プライミティブ型以外では仮引数の渡し方が
参照渡しであることがわかります。
\begin{Verbatim}
>a = [1,2,3];
[1, 2, 3]
>func2(a);
0
>a;
[2, 2, 3]

```

C.5.3 arguments について

JavaScript では引数リストで引数の値などが渡されるほかに `arguments` という配列のようなオブジェクトでもアクセスできます。

- 引き渡された変数の数は `length` で知ることができます。

```

function sumN(){
  var i, s = 0;
  for(i = 0; i <arguments.length;i++) {
    s += arguments[i];
  }
  return s;
}

```

実行例は次のとおりになります。

```

>sumN(1,2,3,4);
10
>sumN(1,2,3,4,5);
15

```

- 引数があっても無視できます。

```
function sumN2(a,b,c){
  var i, s = 0;
  for(i = 0; i <arguments.length;i++) {
    s += arguments[i];
  }
  return s;
}
```

この例では引数が 3 個より少なくても正しく動きます。実行例は次のとおりです。

```
>sumN2(1,2,3,4,5);
15
```

- 仮引数と `arguments` は対応していて、片方を変更しても他の方も変更されます。

実行例は次のとおりです。

```
function sum2(a, b){
  var c;
  a *= 3;
  console.log(arguments[0]);
  return a + b;
}

>sum2(1,2,3,4,5);
3
5
```

C.6 変数のスコープと簡単な例

JavaScript に限らず、どのプログラミング言語でも変数のスコープという概念は重要です。変数のスコープとは、使用している変数が通用する範囲のことです。JavaScript では次のようになっています。

- 変数は宣言しなくても使用できます。この場合はグローバル変数になります。つまり、どの範囲からも参照が可能になります。
- 関数内で `var` を用いて宣言した変数はその関数の中で有効です。つまり、ローカル変数となります。同じ名前のグローバル変数があった場合には、そのグローバル変数にはアクセスできません。
- 関数内での変数の宣言は、どこで行ってもローカルな変数と扱われます。したがって、関数の途中で `var` 宣言してそこで初期化した場合、その前で同じ変数を使用するとその値は初期化されていない `undefined` になります。

- JavaScript では関数も通常のオブジェクトで変数に代入することができます。今まで関数を定義していた方法

```
function foo() {...}
}
```

はグローバル変数 `foo` に対し次の形のコードと同じ意味を持ちます。

```
var foo = function(){...}
}
```

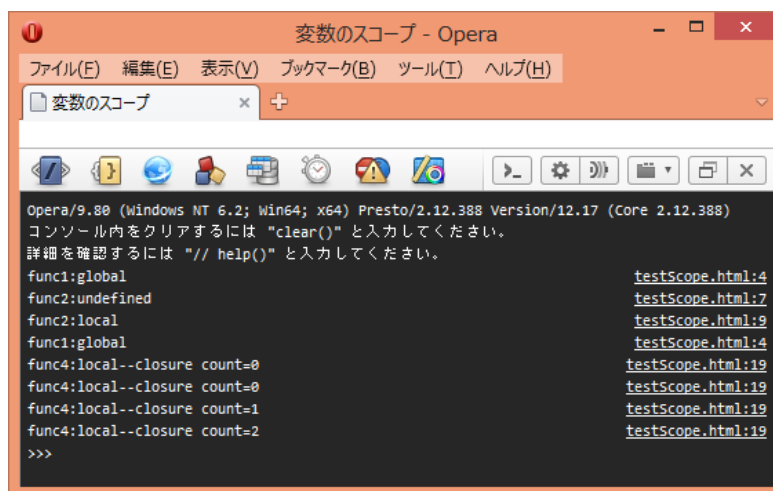


図 C.1: 変数のスコープ-コンソールを使う

HTML リスト C.1: 変数のスコープ (testScope.html)

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4 <title>変数のスコープ</title>
5 <script type="text/ecmascript">
6   var Variable = "global";
7   function func1() {
8     console.log("func1:"+Variable);
9   }
10  function func2() {
11    console.log("func2:"+Variable);
12    var Variable = "local";
13    console.log("func2:"+Variable);
14  }
15  function func3(arg) {
16    var Variable = "local";
17    func1();
18  }
```

```
19 function func4() {
20     var Variable = "local--closure";
21     var count = 0;
22     return function(){
23         console.log("func4:"+Variable+" count="+count);
24         count++;
25     };
26 }
27
28 func1();
29 func2();
30 func3();
31 f = func4();
32 f();
33 (func4())();
34 f();
35 f();
36 </script>
37 </head>
38 <body>
39 </body>
40 </html>
```

この HTML 文書は JavaScript の実行結果をコンソールに出力するものです。

- 6 行目でグローバル変数変数 Variable を定義して、値を"global"に定義しています。
- 関数関数 func1 は変数変数 Variable の値を関数名とともに出力します (7 行目から9 行目)。console.log() はコンソールに引数の値を表示する関数です。
- 関数関数 func2 は変数 Variable を関数内で定義しその値を"local"に定義しています。さらにその値を関数名とともに出力します (10 行目から14 行目)。
- 関数 func3 は関数 func2 と同様に変数 Variable を関数内で定義しその値を"local"に定義しています。出力は関数 func1 を呼び出すことで実現しています。
- 関数 func4 は関数 func3 と同様に変数 Variable を関数内で定義しその値を"local"に定義しています。この関数の戻り値は変数変数 Variable の値をコンソールに出力する関数です。
- 28 行目で関数 func1 が実行されています。この関数では変数 Variable は行目で定義されたものが参照されますので出力結果は func1:global となります。
- 29 行目で関数 func2 が実行されています。この関数では Variable は12 行目でローカル変数が定義されていることからグローバル変数のほうが参照されません。この時点では値が代入されていないので func2:undefined が出力されます。そのあとで値が設定されるので次の出力結果は func2:local となります。
- 30 行目で関数 func3 が実行されています。この関数では Variable は12 行目でローカル変数が定義されて、値が設定されています。出力は関数 func1 で行われているので、参照される Variable はグローバルに定義されているものになります。

- 31 行目で関数 `func4` が実行されていて、その戻り値が変数 `f` に代入されています。この時点では戻り値である関数は実行されていません。関数 `func4` の中でローカル変数 `Variable` が定義され、その値を戻り値の関数が利用しています。
- 32 行目で関数 `func1` が実行されています。このとき利用される変数 `Variable` は関数 `func4` 内で定義されたものです。つまり、この関数からは関数 `func4` 内で定義されたローカル変数が参照できるのです。しかしながら、この関数で定義されたローカル変数を外部から直接変更や参照する手段はありません。
- 33 行目戻り値が関数オブジェクトのとき、それを変数にしまわないでそのまま実行する方法です。関数 `func4` の内容を直接この () 内に記述することも可能です。この場合、関数自体に名前がありませんので無名関数と呼ばれます。
- 34 行目では31 行目で保存された関数を再び実行しています。この関数ではコンソールへの出力後、ローカル変数 `count` の値を 1 増加させているのでこの変数の値が 1 となり、その値が出力されます。この変数 `count` の値を変えることができるのはこの関数の呼び出しでしかできません。このような技術をクロージャと呼びます。クロージャについては後で詳しく説明します。
- 35 行目ではもう一度、同じ関数が実行され、変数 `count` の値として 2 が出力されます。この変数の値は関数 `f()` が呼び出されたときだけ、1 増加し、これ以外の方法でこの変数の値を変更することができません。つまり、クロージャを用いることで一種のカプセル化が可能となっています。

C.6.1 JavaScript における関数の特徴

JavaScript 関数ではほかの言語では見られない関数の取り扱い方法があります。

関数もデータ

関数もデータ型のひとつなので、関数の定義を変数に代入することができます。HTML リスト C.1 の 31 行目では関数の戻り値が関数オブジェクトで、その結果を変数に代入しています。代入はいつでもできるので、実行時に関数の定義を変えることも可能です。

無名関数とコールバック関数

HTML リスト C.1 の 22 行目の関数オブジェクトは `function` の後には関数名がありません。このような関数は無名関数と呼ばれます。HTML 文書などでは、いろいろなイベント (マウスがクリックされた、一定の時間が経過した) が発生したときに、その処理を行う関数を登録する必要があります。この関数をその場で定義して、無名関数で渡すことはよく使われる技法です。なお、ある関数に引数として渡される関数は渡された関数の中で呼び出されるのでコールバック関数と呼ばれます。

次の例は、一定の経過時間後にある関数を呼び出す window オブジェクトの `setTimeout()` メソッドの使用例です。

```
1 var T = new Date();
2 window.setTimeout(
3   function(){
4     var NT = new Date();
5     if(NT-T<10000) {
6       console.log(Math.floor((NT-T)/1000));
7       window.setTimeout(arguments.callee,1000);
8     }
9   },1000);
```

- 1 行目では実行開始時の時間を変数 `T` に格納しています。単位はミリ秒です。
- このメソッドは一定時間経過後に呼び出される関数と、実行される経過時間 (単位はミリ秒) を引数に取ります。
- 実行する関数は 3 行目から 9 行目で定義されています。
- この関数内で一定の条件のときはこの関数を呼び出すために、この関数に名前はありません (3 行目)。
- 4 行目で呼び出されたときの時間を求め、経過時間が 10000 ミリ秒以下であれば (5 行目)、経過時間を秒単位で表示します (6 行目)。
- さらに、自分自身を 1 秒後に呼び出す (7 行目)。arguments をもつ関数を arguments.callee で呼び出すことができます。つまり無名関数である自分自身を呼び出せます。

問題 C.1 次のプログラムは何を計算するか答えなさい。

```
var f = function(n) {
  if( n<=1) return 1;
  return n*arguments.callee(n-1);
}
```

自己実行関数

関数を定義してその場で直ちに実行することができます。次のコードを見てみましょう。

```
var i;
for(i=1;i<10;i++) {
  console.log(i+" "+i*i);
}
```

このプログラムを実行すると 1 から 9 までの値とその 2 乗の値がコンソールに出力されます。実行後に、コンソールに `i` と入力すれば 10 が出力されます。つまり、変数 `i` が存在しています。

ある関数を実行した後でその中で使用したグローバル変数を消してしまいたいことがある。それを実現するためには次のように記述する。

```
(function(){  
  var i;  
  for(i=1;i<10;i++) {  
    console.log(i+" "+i*i);  
  })();
```

この様に関数の定義を全体で `()` で囲み、そのあとに関数の呼び出しを示すための `()` を付けています。

この技法は、初期化の段階で 1 回しか実行しない事柄を記述し、かつグローバルな空間を汚さない(余計な変数などを残さない)手段として用いられています。

C.6.2 クロージャ

JavaScript をオブジェクト指向言語として使用するための基本的な概念です。ここで上げる例は実用に乏しいと思われるかもしれないが、この後に出てくるオブジェクトの項ではより実用的なものと理解できるでしょう。

スコープチェーン

関数の中で関数を定義すると、その内側の関数内で `var` で宣言された変数のほかに、一つ上の関数で利用できる(スコープにある)変数が利用できます。これがスコープチェーンです。

```
var G1, G2;  
function func1(a) {  
  var b, c;  
  function func2() [  
    var G2, c;  
    ...  
  ]  
}
```

- 関数 `func1()` ではグローバル変数である `G1` と `G2`、仮引数の `a` とローカル変数 `b` と `c` が利用できます。
- 関数 `func2()` ではグローバル変数である `G1`、`func1()` の仮引数の `a` と `func1()` のローカル変数 `b`、`func2()` のローカル変数 `G2` と `c` が利用できます。

このように内側で定義された関数は自分自身の中で定義されたローカル変数があるかを探し、見つからない場合には一つ上のレベルでの変数を探します。これがスコープチェーンです。JavaScript の関数のスコープは関数が定義されたときのスコープチェーンが適用されます。これをレキシカルスコープと呼びます。レキシカルスコープは静的スコープとも呼ばれます。これに対して実行時にスコープが決まるものは動的スコープと呼ばれます。

クロージャ

このように関数内部で宣言された変数は、その外側から参照することができません。つまり、その関数は関数内のローカル変数を閉じ込めています。しかし、関数内部で定義された関数を外部に持ち出す（グローバルな関数にする）と、持ち出された関数のスコープチェーン内に定義された親の関数のスコープを引き継いでいることから、親の関数のローカル変数の参照が可能となります。

このような関数に対して依存する環境（変数や呼び出せる関数などのリスト）を合わせたものをその関数のクロージャと呼ばれます。

次の例は関数を定義したのちに、仮引数の値を 1 増加しています。

```
function f(arg) {  
  var n = function() {  
    return arg;  
  }  
  arg++;  
  return n;  
}  
  
>var m = f(123);  
undefined  
>m();  
124
```

定義時の `arg` の値ではなく、参照時の `arg` の値が参照されていることに注意してください。

このことが連続して関数を作成したときにバグを引き起こすことがあります。

次の例は配列の添え字を戻り値にする関数を 3 つ定義しています。配列に保存された関数が添え字の値を返すように見えますが、実際にはそのようには動きません。

```
function f() {  
  var a = [];  
  var i;  
  for(i=0; i<3; i++) {  
    a[i] = function() {  
      return i;  
    };  
  }  
}
```

```
    return a;
}

>var a = f();
undefined
>a[0]();
3
>a[1]();
3
>a[2]();
3
```

すべて同じ値が返る関数になってしまっています。これは関数 `f()` が実行されるとローカルな変数 `i` の値は `for` 文が終了した時点で値が 3 となり、戻り値の関数が実行された時点では、その値が参照されるからです。

これを避けるためには関数にその値を渡してスコープチェーンを切る必要があります。

```
function f2() {
  var a = [];
  var i;
  for(i=0; i<3; i++) {
    a[i] = (function(x){
      return function() {
        return x;
      }
    })(i);
  };
  return a;
}
```

- 引数を取る無名関数を用意し、その場で与えられた引数を返す無名関数を返す関数を実行しています。
- 仮引数には、実行されたときの `i` のコピーが渡されるので、その後変数の値が変わっても呼び出された時の値が保持されます。

```
>var a = f2();
undefined
>a[0]();
0
>a[1]();
1
>a[2]();
2
```

最後の例題は同じオブジェクトを連続的に作成して、通し番号を付けたい場合に応用できます。
 Ajax で非同期通信を行う場合では通信が終了したときに呼び出される関数 (<コールバック関数>要素) を使用します。このとき、コールバック関数を定義したときに特別な変数の値を利用する場合があります。非同期通信を同時に複数行う場合にはその変数の値が呼び出し時と実行時で変わってしまう場合があります。定義したときの変数の値を実行時にそのまま利用するためにはこのような手法が必要になります。

C.7 配列のメソッド

配列には表 C.3 のようなメソッドが定義されています。このうちいくつかを具体的な例で示します。

表 C.3: 配列のメソッド

メンバー	説明
length	配列の要素の数。このメンバーに値を代入すると配列の大きさが変えられる。
join(separator)	配列を文字列に変換する。separator はオプションの引数で、省略された場合はカンマ, である。
concat(i1,i2,...)	指定した引数の値をもとの配列に付け加えた配列を新たに作成する。引数が配列の場合は配列の要素を付け加える。元の配列は変化しない。
sort([func])	配列の要素をアルファベット順に並べ替える。func は並べ替えを指定するための関数である。
indexOf(value[,start])	start 以降 (指定しない場合は 0) の要素で value の値と等しい (===) 最初のインデックスを返す。見つからない場合は -1。
lastIndexOf(value[,start])	start 以前 (指定しない場合は配列の最後) の要素で value の値と等しい (===) 最初のインデックスを返す。見つからない場合は -1。
pop()	配列の最後の要素を削除し、その値を返す。配列をスタックとして利用できる。
push(i1,i2,...)	引数で渡された要素を配列の最後に付け加える。配列をスタックやキューとして利用できる。
shift()	配列の最初の要素を削除し、その値を返す。配列をキューとして利用できる。
unshift(i1,i2,...)	引数で渡された要素を配列の最初の要素とする。
reverse()	与えられた配列の要素を逆順に並べ替えたものに変更する。

次ページへ続く

表 C.3: 配列のメソッド (続き)

メンバー	説明
<code>slice(start[,end])</code>	<code>start</code> から <code>end</code> の前の位置にある要素を取り出した配列を返す。 <code>end</code> がいないときは配列の最後までを指定したことになる。元の配列は変化しない。
<code>splice(start,No,i1,i2,...)</code>	<code>start</code> の位置から <code>No</code> 個の要素を取り除き、その位置に <code>i1,i1,...</code> 以下の要素を付け加える。
<code>forEach(func)</code>	引数 <code>func</code> に与えられた関数を配列の各要素に対して実行する。関数の引数は配列の値、配列の <code>index</code> 、配列の順となる。与えられた関数の戻り値は無視される。途中でループの処理を中断できない。
<code>map(func)</code>	引数に与えられた関数を配列の各要素に対して実行し、関数の戻り値からなる新しい配列を作成する。引数として与えられた関数の引数は <code>forEach</code> と同じである。
<code>reduce(func[,initial])</code>	<code>func</code> は 2 つの仮引数をとる関数。 <code>initial</code> がいないときは <code>func</code> が初めて呼び出されるときは配列の 1 番目と 2 番目の要素が引数として与えられる。2 回目以降の呼び出しでは 1 番目の引数はその前に呼び出された関数の戻り値が使用される。 <code>initial</code> があるときは初めての呼び出しで 1 番目の引数として使われる。
<code>reduceRight(func[,initial])</code>	<code>reduce</code> と同様のメソッド。配列の要素のアクセスが大きい方から小さい方になる。
<code>every(func)</code>	<code>func()</code> を各配列の要素に適用し、すべてが <code>true</code> と解釈される値のとき、 <code>true</code> を返す。どこか一つで <code>false</code> のときにはそこで実行が打ち切れ、 <code>false</code> の値が返る。 <code>func</code> の引数は <code>forEach</code> と同じである。
<code>some(func)</code>	<code>func()</code> を各配列の要素に適用し、どれかひとつが <code>true</code> と解釈される値のとき、 <code>true</code> を返す。どこか一つで <code>true</code> のときにはそこで実行が打ち切られる。すべての結果が <code>false</code> のときは <code>false</code> となる。 <code>func</code> の引数は <code>forEach</code> と同じである。
<code>filter(func)</code>	<code>func</code> で呼び出される関数の戻り値が <code>true</code> に変換される要素を集めて新しい配列を作成する。 <code>func</code> の引数は <code>forEach</code> と同じである。

`slice` と `splice` は似たメソッドですが、引数の意味が異なるところがあるので注意が必要です。

```

1 >A=[1,2,3,4,5,6,7,8,9];
2 [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
3 >A.slice(2,4)
4 [3, 4]
5 >A;
6 [1, 2, 3, 4, 5, 6, 7, 8, 9]
7 >A.splice(2,4)
8 [3, 4, 5, 6]
9 >A;
10 [1, 2, 7, 8, 9]
11 >A.splice(2,0,"a","b")
12 []
13 >A;
14 [1, 2, "a", "b", 7, 8, 9]
```

- `slice` の引数は配列から取り出す開始位置と終了位置の直後までを指定します。したがって、3 行目の例では 2 番目と 3 番目の要素が切り出されます。また、5 行目の実行例からも元の配列が変化していないことがわかります。
- `splice` の 1 番目の引数は配列から取り出す開始位置で、2 番目の引数は `slice` とは異なり、取り出す個数です (7、8 行目参照)。
- `splice` を行った配列は戻り値の部分を取り除かれています (9、10 行目)。
- `splice` は追加の引数で取り除いた位置に挿入する要素を指定できます。

`forEach` と `map` の引数で与えられる関数は 3 つの引数 `value`、`index` と `array` をとることができます。したがって、次の関係が成立します。

```
array[index] = value
```

元の配列の要素を変更しないのであれば、3 つ目の引数 `array` は必要ありません。

```
>A=[1,2,3];
A.forEach(function(V, i, AA) {AA[i] = V*V;});
>A;
[1, 4, 9]
```

この例では、元の配列の各要素を 2 乗して元の配列に格納しています。

これに対し、`map` では新しい配列を作成します。

```
>A=[1,2,3];
[1, 2, 3]
>A.map(function(V) {return V*V;});
[1, 4, 9]
>A;
[1, 2, 3]
```


次の例は `reduce` の実行例です。

```
>[1,2,3].reduce(function(x,y){return x+y;});  
6
```

与えられた関数は引数の和を求めているので、全体としては配列の中の要素の和が求められています。

問題 C.2 次のプログラムを実行したときの結果を述べなさい。

```
A=[3,1,4,5,8,10]  
A.filter(function(X) {return X%2;});  
A.reduce(function(x,y){ return x+y%2;});  
A.reduce(function(x,y){ return x+y%2;},0);
```

C.5.3 で `arguments` の総和をとる関数を作成しています。この部分を `reduce` で置き換えるとうまくいきません。

```
>function sum() {return arguments.reduce(function(x,y){return x+y;})}  
undefined  
>sum(1,2,3);  
VM510:2 Uncaught TypeError: arguments.reduce is not a function(...)
```

「`arguments` には `reduce` という関数がない」というメッセージが表示されています。これからも、`arguments` は配列のようなものだけれども配列とは違うことがわかります。

これを解決するにはオブジェクトのメソッドとして関数を呼び出す `call` を使用します。

```
>function sum(){return Array.prototype.reduce.call(arguments,function(x,y){return x+y;})}  
undefined  
>sum(1,2,3);  
6
```

`reduce` は `Array` オブジェクトの `prototype` で定義されている関数です。それを `arguments` のメソッドとして呼び出すために `call` を用いています。`reduce` の引数として関数を必要としますので、それを無名関数として2番目の引数に与えています。

`document.getElementsByTagName` などで得られた配列のようなものに対して `forEach` などを利用したい場合にもこの手法が利用できます。

C.8 オブジェクト

C.8.1 配列とオブジェクト

配列はいくつかのデータをまとめて一つの変数に格納しています。各データを利用するためには `foo[1]` のように数による添え字を使います。これに対し、オブジェクトでは添え字に任意の文字列を使うことができます。

次の例はあるオブジェクトを定義して、その各データにアクセスする方法を示しています。

```
var person = {  
  name : "foo",  
  birthday : {  
    year : 2001,  
    month : 4,  
    day : 1  
  },  
  "hometown" : "神奈川",  
}
```

- オブジェクトは全体を {} で囲みます。
- 各要素はキーと値の組で表されます。両者の間は : で区切ります。
- キーは任意の文字列でかまいません。キー全体を "" で囲わなくてもかまいません。
- 値は JavaScript で取り扱えるデータなならば何でもかまいません。上の例ではキー birthday の値がまたオブジェクトとなっています。
- 各要素の値を取り出す方法は 2 通りあります。
一つは、演算子を用いてオブジェクトのキーをそのあとに書きます。もう一つは配列と同様に [] 内にキーを文字列として指定する方法です。

```
>person.name;  
"foo"  
>person["name"];  
"foo"
```

オブジェクトの中にあるキーをすべて網羅するようなループを書く場合や変数名として利用できないキーを参照する場合には後者の方法が利用されます。

- キーの値が再びオブジェクトであれば、前と同様の方法で値を取り出せます。

```
>person.birthday;  
Object {year: 2001, month: 4, day: 1}  
>person.birthday.year;  
2001  
>person.birthday["year"];  
2001
```

この例のように取り出し方は混在しても問題ありません。

- キーの値は代入して変更できます。

```
>person.hometown;
"神奈川"
>person.hometown="北海道";
"北海道"
>person.hometown;
"北海道"
```

- 存在しないキーを指定すると値として `undefined` が返ります。

```
>person.mother;
undefined
```

- 存在しないキーに値を代入すると、キーが自動で生成されます。

```
>person.mother = "aaa";
"aaa"
>person.mother;
"aaa"
```

- オブジェクトのキーをすべて渡るループは `for-in` で実現できます。

- `for(v in obj)` の形で使用します。変数 `v` はループ内でキーの値が代入される変数、`obj` はキーが走査されるオブジェクトです。
- キーの値は `obj[v]` で得られます。

```
>for(i in person) { console.log(i+" "+person[i]);};
name foo
birthday [object Object]
hometown 北海道
mother aaa
undefined
```

最後の `undefined` は `for` ループの戻り値です。

なお、オブジェクトを `{}` の形式で表したものをオブジェクトリテラルとよびます。

C.8.2 コンストラクタ関数

オブジェクトを定義する方法としてはコンストラクタ関数を使う方法がある。次の例はコンストラクタ関数を用いて、前の例と同じオブジェクト (インスタンス) を構成しています。

```
function Person(){
  this.name = "foo";
  this.birthday = {
```

```
    year : 2001,
    month : 4,
    day : 1
  };
  this["hometown"] = "神奈川";
}
```

- 通常、コンストラクタ関数は大文字で始まる名前を付けます。
- そのオブジェクト内にメンバーを定義するために、`this` をつけて定義します。ここでは、前の例と同じメンバー名で同じ値を設定しています。
- この関数には `return` がないことに注意すること。
- この関数を用いてオブジェクトを作成するためには、`new` をつけて関数を呼び出します。

```
>var person = new Person();
undefined
```

- 元来、戻り値がないので `undefined` が表示されているが、オブジェクトは作成されています。
- 前と同じ文を実行すれば同じ結果が得られます。

ここの例はコンストラクタ関数に引数がないが、引数を持つコンストラクタ関数も定義が可能です。これにより同じメンバーを持つオブジェクトをいくつか作る必要がある場合にプログラムが簡単になります。

次の例はコンストラクタ関数を `new` を用いないで実行した場合です。

```
>p = Person();
undefined
>p;
undefined
>name;
"foo"
>window.name;
"foo"
>birthday == window.birthday
true
```

- この関数は戻り値がないので、`undefined` が変数 `p` に代入されます。
- このとき、キーワード `this` が指すのはグローバルオブジェクトです。
- 現在の実行環境はブラウザ上なので、このときのグローバルオブジェクトは `window` です。

- このとき、グローバル変数はすべてグローバルオブジェクトのメンバーとしてアクセス可能です。この例では `this.name` に値を代入した時点で変数 `name` が定義されています。
- 最後の例からも、`name` と `window.name` が同じものであることがわかります。

問題 C.3 上の実行例を次のように変えます。

```
function Person(D){
  this.name = "foo";
  this.birthday = {
    year : 2001,
    month : 4,
    day : 1
  };
  this["hometown"] = "神奈川";
  return D;
}
```

これに対して次のように実行したとき、作成されるオブジェクトは何か答えなさい。

1. `p = new Person(1);`
2. `p = new Person([1,2,3]);`
3. `p = new Person({o:"1"});`
4. `p = new Person(function(){return 2;});`
5. `p = new Person(new function(){this.a = "a"});`

constructor プロパティ

オブジェクトが作成されると、`constructor` プロパティとよばれる特殊なプロパティも設定されます。このプロパティはオブジェクトを作成したときに使われたコンストラクタ関数を返します。

```
>var p = new Person();
undefined
>p.name;
"foo"
>p.constructor;
function Person(){
  this.name = "foo";
  this.birthday = {
    year : 2001,
```

```

        month : 4,
        day : 1
    };
    this["hometown"] = "神奈川県";
}

```

このプロパティに含まれるものは関数なので、コンストラクタの名前を知らなくても、元と同じオブジェクトのコピーが作成できます。

```

>np = new p.constructor();
Person {name: "foo", birthday: Object, hometown: "神奈川県"}
>np.constructor;
function Person(){
    this.name = "foo";
    this.birthday = {
        year : 2001,
        month : 4,
        day : 1
    };
    this["hometown"] = "神奈川県";
}

```

オブジェクトリテラルを使ってオブジェクトを作ると、組み込み関数の `\ElmJ{Object()}` コンストラクタ関数がセットされます。

```

\begin{Verbatim}
>o = {}
Object {}
>o.constructor;
function Object() { [native code] }

```

このプロパティは `for-in` ループ内では表示されません。

instanceof 演算子

`instanceof` 演算子はオブジェクトを生成したコンストラクタ関数が指定されたものを判定できます。次の結果は Opera 29 で C.8.2 を実行した後にさらに実行したものです。

```

>p instanceof Person
true
>p instanceof Object;
true
>o instanceof Object;
true

```

```
>o instanceof Person
false
```

C.8.3 オブジェクトリテラルと JSON

JSON(JavaScript Object Notation) はデータ交換のための軽量なフォーマットです。形式は JavaScript のオブジェクトリテラルの記述法と全く同じです。

- 正しく書かれた JSON フォーマットの文字列をブラウザとサーバーの間でデータ交換の手段として利用できます。
- JavaScript 内で、JSON フォーマットの文字列を JavaScript のオブジェクトに変換できます。
- JavaScript 内のオブジェクトを JSON 形式の文字列に変換できます。

JavaScript のオブジェクトと JSON フォーマットの文字列の相互変換の手段を提供するのが JSON オブジェクトです。

次の例は 2 つの同じ形式からなるオブジェクトを通常の配列に入れたものを定義しています。

```
var persons = [{
  name : "foo",
  birthday : { year : 2001, month : 4, day : 1},
  "hometown" : "神奈川",
},
{
  name : "Foo",
  birthday : { year : 2010, month : 5, day : 5},
  "hometown" : "北海道",
}];
```

次の例はこのオブジェクトを JSON に処理させたものです。

```
>s = JSON.stringify(persons);
"[{"name":"foo","birthday":{"year":2001,"month":4,"day":1},
" hometown":"神奈川"},
{"name":"Foo","birthday":{"year":2010,"month":5,"day":5},
" hometown":"北海道"}]"
>s2 = JSON.stringify(persons,["name"," hometown"]);
"[{"name":"foo"," hometown":"神奈川"}, {"name":"Foo"," hometown":"北海道"}]"
>o = JSON.parse(s2);
[Object, Object]
>o[0];
Object {name: "foo", hometown: "神奈川"}
```

- JavaScript のオブジェクトを文字列に変更する方法は `JSON.stringify()` を用います。このまま見ると"がおかしいように見えるが表示の関係でそうになっているだけです。なお、結果は途中で改行を入れているが実際は一つの文字列となっています。
- `JSON.stringify()` の二つ目の引数として対象のオブジェクトのキーの配列を与えることができます。このときは、指定されたキーのみが文字列に変換されます。
- ここでは、`"name"` と `"hometown"` が指定されているので `"birthday"` のデータは変換されていません。
- JSON データを JavaScript のオブジェクトに変換するための方法は `JSON.parse()` を用います。
- ここではオブジェクトの配列に変換されたことがわかります。
- 各配列の要素が正しく変換されていることがわかります。

問題 C.4 上の実行例で

```
s3 = JSON.stringify(persons, ["year"]);
```

としたときの結果はどうなるか調べなさい。

C.8.4 ECMAScript5 のオブジェクト属性

オブジェクト指向言語におけるプロパティとメソッドの属性

オブジェクト指向言語ではオブジェクトのプロパティやメソッドは次のように分類されます。

- インスタンスフィールド
インスタンスごとに異なる値を保持できるプロパティ
- インスタンスメソッド
クラスのすべてのインスタンスで共有されるメソッド
- クラスフィールド
クラスに関連付けられたプロパティ
- クラスメソッド
クラスに関連付けられたメソッド

JavaScript では関数もデータなのでフィールドとメソッドに厳密な区別はないのでフィールドとメソッドは同一視できます。また、`prototype` を用いればクラスフィールドなども作成できます。

通常、オブジェクト指向の言語ではフィールドを勝手に操作されないようにするために、フィールドを直接操作できなくして、値を設定や取得するメソッドを用意します。そのために、フィールドにアクセスするため記述が面倒になるという欠点もあります。

一方、プロパティの代入の形をとっても実際はゲッターやセッター関数を呼ぶ形になっている言語もあります。JavaScript の最新版 1.8.1 以降ではこの方式が可能となっています。

プロパティ属性

JavaScript は Ecma International が定義している ECMAScript の仕様に基づいています。2014 年現在、最新バージョンの ECMAScript 5.1 の仕様は ECMA-262² で公開されています。

このバージョンではオブジェクトのプロパティやメソッドにプロパティ属性という機能が追加されました。オブジェクトのプロパティの属性には表 C.4 のようなものがあります。また、メソッド

表 C.4: プロパティの属性のリスト

属性名	値の型	説明	デフォルト値
value	任意のデータ	プロパティの値	undefined
writable	Boolean	false のときは value の変更ができない	false
enumerable	Boolean	true のときは for-in ループでプロパティが現れる。	false
configurable	Boolean	false のときはプロパティを消去したり、value 以外の値の変化ができない	false

に関しては表 C.5 のものがあります。

表 C.5: メソッドの属性のリスト

属性名	値の型	説明	デフォルト値
get	オブジェクトまたは未定義	関数オブジェクトでなければならない。プロパティの値が読みだされるときに呼び出される	undefined
set	オブジェクトまたは未定義	関数オブジェクトでなければならない。プロパティの値を設定するとき呼び出される	undefined
enumerable	Boolean	true のときは for-in ループでプロパティが現れる。	false
configurable	Boolean	false のときはプロパティを消去したり、value 以外の値の変化ができない	false

これらの属性を使うとオブジェクトのプロパティの呼び出しや変更に関して、いわゆるゲッター関数やセッター関数を意識しないで呼び出すことが可能となります。次の例は [8] にある 9 章の例 9.2 の Range を改良した例 9.18 と 9.21 をまとめたものです。この例はさらにコンストラクタやセッターのところで不適切な値が設定されないようにチェックを加えています。

²<http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf>

```

1  function Range(from, to) {
2      if(from > to ) throw Error("Range: from must be <= to");
3      function getF() { return from;};
4      function setF(v) {
5          if(v <= to ) from = v;
6          else throw Error("Range: from must be <= to");
7      };
8      function getT() {return to;};
9      function setT(v) {
10         if(v >= from ) to = v;
11         else throw Error("Range: from must be <= to");
12     }
13     Object.defineProperty(this, "from",
14         {get: getF, set: setF, enumerable:true, configurable:false});
15     Object.defineProperty(this, "to",
16         {get: getT, set: setT, enumerable:true, configurable:false});
17 }

```

- 前と同様に、二つの引数を持つ Range 関数を作成します。
- 2 行目では下限の値が上限の値より大きくなったらエラーを発生させています。
- 3 行目から 12 行目では下限 (from) と上限 (to) のゲッターとセッターを定義しています。各セッターではコンストラクタと同様に値に矛盾が起きていたらエラーを発生させるようにしています。
- 13 行目から 14 行目で作成するインスタンスのプロパティ from のオブジェクト属性を defineProperty() を用いて定義しています。この関数の引数は次のとおりです。
 - 一番目の引数はプロパティを設定するオブジェクト
 - 2 番目の引数はプロパティの名前。ここでは文字列で与えている。
 - 3 番目の引数は設定するプロパティ属性。ここではゲッター関数とセッター関数を指定し、for-in ループで列挙可能にし、関数の置き換えや再設定ができないように設定しています。
- 15 行目から 16 行目では同様に作成するインスタンスのプロパティ to のオブジェクト属性を設定しています。

```

Range.prototype = {
    includes : function(v) {
        return this.from <= v && v <= this.to;
    },
    foreach : function(f) {

```

```

    for(var k = Math.ceil(this.from); k<= this.to; k++) f(k);
  },
  toString : function() { return "[" + this.from+",...,"+this.to+"]";}
};
Object.defineProperties(Range.prototype,
{ includes : {enumerable : false},
  foreach : {enumerable : false},
  toString : {enumerable : true}
});

```

- 18 行目から 26 行目までは前と同じく prototype にクラスメソッドを定義しています。
- これらのクラスメソッドの一部を列挙可能にしないために definePropaties() を用いて設定しています。この関数は definePropaty() が一つのプロパティごとに設定するのに対し、複数のプロパティに対して設定が可能です。なお、ここでは機能を確認するため設定の値を変えています。
 - 一番目の引数は設定するオブジェクト
 - 2 ア番目の引数は設定するプロパティをキーとし、設定する属性のリストを表すオブジェクトの値

これらの設定が正しく動作しているか検証します。

- オブジェクトを作成し、プロパティを列挙します。

```

>r = new Range(1,5);
Range {from: (...), to: (...), toString: function}
>for(key in r) console.log(key+": "+r[key]);
from:1
to:5
toString:function () { return "[" + this.from+",...,"+this.to+"]";}
undefined
>r.includes;
function (v) {
  return this.from <= v && v <= this.to;
}

```

- include と foraeach の enumerable のプロパティを false、toString の enumerable のプロパティを true に設定したので、メソッドは toString しか表示されていません。
- 関数が存在することは確認できます。
- 各メソッドが正しく動作するか確認します。

```
>r.includes(3);
true
>r.includes(10);
false
```

以前と同じ動作をしています。

- プロパティに代入します。

```
>r.from = 10;
Uncaught Error: Range: from must be <= to
>r.from = -5;
-5
>r.from;
-5
```

- 上限より大きな値を下限に設定するとエラーが起きます。get で指定した関数が動作していることがわかります。
- 条件を満たす値を設定すれば、正しく設定されます。

念のため、オブジェクトの値がどうなっているか確認します。

```
>for(key in r) console.log(key+": "+r[key]);
from:-5
to:5
toString:function () { return "[" + this.from+", ..., "+this.to+""];}
undefined
```

- プロパティが削除できるか確認します。

```
>delete r.from;
false
>r.from
-5
```

delete の結果が false なので、取り除きに失敗しています。値は元の値のままです。

問題 C.5 上の実行例に対して次のことをしなさい。

1. 各メソッドが正しく動くことを確認しなさい。
2. 13 行目から 16 行目にある 2 つの Object.defineProperty() 関数を Object.defineProperties() 関数で置き換えなさい。
3. 3 行目から 12 行目で定義されている関数はグローバルな関数か答えなさい。また、13 行目から 16 行目の部分は関数 Range() の外に記述してもよいか答えなさい。

4. 3 行目から 12 行目で定義されている関数を無名関数にしてゲッターとセッターを定義しなさい。
5. 関数 `Range()` 内にある変数 `from` と `to` はどこで定義されているか答えなさい。
6. 関数 `Range()` 内にある変数 `from` と `to` と `this.from`、`this.to` は同じものを指すか答えなさい。たとえば、3 行目の `from` を `this.from` としたら何が起こるか確認しなさい。
7. 上記の実行例で `delete r.from` が失敗する理由を説明しなさい。

C.8.5 エラーオブジェクトについて

前節の例ではオブジェクトの条件に合わない値を設定すると、エラーを発生するようにしています。ここではエラーオブジェクトについて詳しく説明します。次の例は前の実行例に、実行時に上限と下限の値を設定するためのテスト関数です。コンソールから `var r = test()` などで行うと上限と下限の値が正しくなるまで繰り返されます。

なお、このプログラムはブラウザ上で実行されることを想定しています。

```
function test() {  
  var f, t;  
  for(;;) {  
    try {  
      f = Number(prompt("区間の下限の値を入力してください"));  
      t = Number(prompt("区間の上限の値を入力してください"));  
      return new Range(f,t);  
    } catch(e) {  
      console.log(e.name+": "+e.message);  
      console.log("from:"+f+", to:"+t);  
    }  
  }  
}
```

- 2 行目でこの関数内で使用するローカル変数を宣言しています。
- 3 行目の `for(;;)` は初期条件、終了条件、後処理がすべて記述されていない `for` ループです。これにより無限ループが構成できます。
- エラー処理をする構文が `try/catch/finally` 構文です。
 - `try` の後に書かれたブロック (ここでは 5 行目から 7 行目を含むブロック) が通常実行されます。
 - この実行の間エラーが発生しなければ `catch` の後のブロックは実行されません。
 - エラーが起きると `catch(e)` に書かれた変数 `e` にエラーオブジェクトが設定されています。この変数はこのブロック内でしか有効ではありません。

- try ブロック内ではキーボードから 2 つの数を読むため、window オブジェクトの `prompt()` を呼び出しています。
- この関数の戻り値 (文字列) を `Number` コンストラクタで数に変換しています (5 行目と 6 行目)。
- 7 行目で作成した `Range` オブジェクトを戻り値としています。
- `Range` コンストラクタでは上限の値が下限の値より小さいときはエラーが発生させています。エラーオブジェクトではエラーが起きたコンストラクタの名前が `name` プロパティに、投げられたエラーで引数に書かれた文字列が `message` プロパティに設定されているので、その内容を 9 行目に表示させています。
- その後、コンストラクタを呼んだときの値を表示させています。

次の実行例は、はじめに下限値を 5、上限値を 1 に設定しエラーを表示させ、その後、下限値を 1、上限値を 5 に設定したときのものです。希望したオブジェクトが作成できていることがわかります。

```
>r = test();
Error:Range: from must be <= to
from:5, to:1
Range {from: (...), to: (...), toString: function}
>r.from
1
>r.from =10;
Error: Range: from must be <= to
```

付 録 D CSS について

カスケーディングスタイルシート (CSS) は HTML 文書の要素の表示方法を指定するものです。CSS は JavaScript から制御できます。

文書のある要素に適用されるスタイルルールは、複数の異なるルールを結合 (カスケード) したものです。スタイルを適用するためには要素を選択するセレクトラで選びます。

表 D.1 は CSS3 におけるセレクトラを記述したものです¹。

表 D.1: CSS3 のセレクトラ

セレクトラ	解説
*	任意の要素
E	タイプが E の要素
E[foo]	タイプが E で属性 "foo" を持つ要素
E[foo="bar"]	タイプが E で属性 "foo" の属性値が "bar" である要素
E[foo~="bar"]	タイプが E で属性 "foo" の属性値がスペースで区切られたリストでその一つが "bar" である要素
E[foo^="bar"]	タイプが E で属性 "foo" の属性値が "bar" で始まる要素
E[foo\$="bar"]	タイプが E で属性 "foo" の属性値が "bar" で終わる要素
E[foo*="bar"]	タイプが E で属性 "foo" の属性値が "bar" を含む要素
E[foo = "en"]	タイプが E で属性 "foo" の属性値がハイフンで区切られたリストでその一つが "en" で始まる要素
E:root	document のルート要素
E:nth-child(n)	親から見て n 番目の要素
E:nth-last-child(n)	親から見て最後から数えて n 番目の要素
E:nth-of-type(n)	そのタイプの n 番目の要素
E:nth-last-of-type(n)	そのタイプの最後から n 番目の要素
E:first-child	親から見て一番初めの子要素
E:last-child	親から見て一番最後の子要素
E:first-of-type	親から見て初めてのタイプである要素
E:last-of-type	親から見て最後のタイプである要素
E:only-child	親から見てただ一つしかない子要素

次ページへ続く

¹<http://www.w3.org/TR/selectors/>より引用。

表 D.1: CSS3 のセレクタ (続き)

セレクタ	解説
E:only-of-type	親から見てただ一つしかないタイプの要素
E:empty	テキストノードを含めて子要素がない要素
E:link, E:visited	まだ訪れたことがない (:link) か訪れたことがある (visited) ハイパーリンクのアンカーである要素
E:active, E:hover, E:focus	ユーザーに操作されている状態中の要素
E:target	参照 URI のターゲットである要素
E:enabled, E:disabled	使用可能 (:enable) か使用不可のユーザーインターフェイスの要素
E:checked	チェックされているユーザーインターフェイスの要素
E::first-line	要素のフォーマットされたはじめの行
E::first-letter	要素のフォーマットされたはじめの行
E::before	要素の前に生成されたコンテンツ
E::after	要素の後に生成されたコンテンツ
E.warning	属性 class が "warning" である要素
E#myid	属性 id の属性値が "myid" である要素
E:not(s)	単純なセレクタ s にマッチしない要素
E F	要素 E の子孫である要素 F
E > F	要素 E の子である要素 F
E F+	要素 E の直後にある要素 F
E ~ F	要素 E の直前にある要素 F

いくつか注意する点を挙げます。

- 属性 id の属性値の前に # をつけることでその要素が選ばれます。
- 属性 class の属性値の前に . をつけることでその要素が選ばれます。
- nth-child(n) には単純な式を書くことができます。詳しくは実行例 D.1 を参照してください。このセレクタは複数書いてもかまいません。
- E F と E > F の違いを理解しておくこと。たとえば div div というセレクタは途中で別の要素が挟まれていてもかまいません。また、<div>要素が 3 つある場合にはどのような 2 つの組み合わせも対象となります。

問題 D.1 次の HTML 文書において nth-child の () 内に次の式を入れた時どうなるか報告しなさい。ここで は箇条書きの開始を示す要素であり、 は箇条書きの各項目を示す要素です。

```
<!DOCTYPE html>
<html>
<head>
```



```

<meta http-equiv="Content-Type" content="text/html; charset=utf-8"/>
<title>nth-child のチェック</title>
<style type="text/css" >
li:nth-child(n){
    background:yellow;
}
</style>
</head>
<body>
    <ol>
        <li>1 番目</li>
        <li>2 番目</li>
        <li>3 番目</li>
        <li>4 番目</li>
        <li>5 番目</li>
        <li>6 番目</li>
        <li>7 番目</li>
        <li>8 番目</li>
    </ol>
</body>
</html>

```

[firstnumber=1]

1. n (ここでのリストの設定)
2. $2n$
3. $n+3$
4. $-n+2$

問題 D.2 前問のリストに対し、背景色が次のようになるように CSS を設定しなさい。

1. 偶数番目が黄色、基数番目がオレンジ色
2. 1 番目、4 番目、...のように 3 で割ったとき、1 余る位置が明るいグレー
3. 4 番目以下がピンク
4. 下から 2 番目以下が緑色

問題 D.3 次の HTML 文書を考えます。

```

<!DOCTYPE html>
<html>

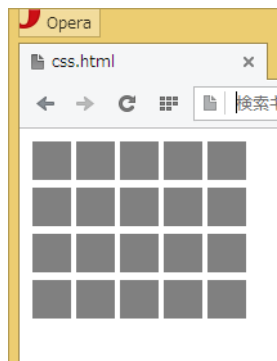
```

```

<head><style>
.Row div {
    display:inline-block;
    width:30px;
    height:30px;
    margin: 2px 2px 2px 2px;
    background: gray;
}
</style>
</head>
<body>
<div class="Row"><div></div><div></div><div></div><div></div><div></div></div>
<div class="Row"><div></div><div></div><div></div><div></div><div></div><div></div></div>
<div class="Row"><div></div><div></div><div></div><div></div><div></div><div></div></div>
<div class="Row"><div></div><div></div><div></div><div></div><div></div><div></div></div>
</body>
</html>

```

[firstnumber=1] このリストで表示されるページは次のようになります。



下図の表示になるように CSS を設定しなさい。

