SVG 実習マニュアル

第五版 revision01

大黒学

SVG 実習マニュアル・第五版 revision01
著者——大黒学

2002 年　5 月　6 日（月）　第零版発行
2006 年　7 月　6 日（木）　第一版発行
2007 年　2 月　3 日（土）　第二版発行
2008 年　2 月 23 日（土）　第三版発行
2009 年　2 月　2 日（月）　第四版発行
2011 年　2 月　3 日（木）　第五版発行
2011 年　2 月 16 日（水）　第五版 revision01 発行

目次

第 1 章　SVG の基礎

1.1　XML の基礎

1.1.1　この節について

この「SVG 実習マニュアル」という文章は、SVG という言語について解説することを目的として書かれたものです。

SVG は、XML という言語を土台にして作られた言語です。そこで、この節では、SVG について理解するための予備知識として、XML というのがどんな言語なのかという説明をしておきたいと思います。

1.1.2　XML とは何か

XML は、「メタ言語」(metalanguage) と呼ばれる言語のひとつです。XML という名前は、Extensible Markup Language という言葉から作られた頭字語です。

メタ言語というのは、言語を定義するための一般的な規則から構成されている言語のことです。言い換えれば、何らかの特定の言語を定義するときに、その土台として使われる言語が、メタ言語です。XML は、主としてインターネットで交換される文書を作るための言語を定義するためのメタ言語です。

XML を土台として作られた言語は、「XML 応用言語」(XML application) と呼ばれます。SVG は、XML 応用言語のひとつです。SVG 以外の XML 応用言語としては、XHTML、MathML、SMIL、RSS、OWL などがあります。

XML は、インターネットで交換されるデータに関する規格について協議する、W3C(World Wide Web Consortium) という組織[1]によって定められた、「勧告」(recommendation) と呼ばれる形式の標準規格になっています。

1.1.3　タグ

XML 応用言語を使って書かれた文書は、「XML 文書」(XML document) と呼ばれます。

XML 文書の中では、「タグ」(tag) と呼ばれるテキスト[2]が重要な役割を果たします。タグというのは、

`<` ・・・ `>`

という形になっているテキスト、つまり、小なり (less than, `<`) という文字で始まって大なり (greater than, `>`) という文字で終わるテキストのことです。

タグは、大きく三種類に分類することができます。三種類のタグは、それぞれ、「開始タグ」(start-tag)、「終了タグ」(end-tag)、「空要素タグ」(empty-element tag) と呼ばれます。それぞれのタグは、

開始タグ	`<` 要素型名 `>`	例 `<tamanegi>`
終了タグ	`</` 要素型名 `>`	例 `</daikon>`
空要素タグ	`<` 要素型名 `/>`	例 `<hakusai/>`

という構文を持っています。このように、タグの種類は、スラッシュ(slash, `/`) という文字の有無と位置によって示されます。

1.1.4　要素

XML 文書は、小さな部品が組み合わさって大きな部品ができている、という構造を持っています。XML 文書の構造を作るための部品となるテキストは、「要素」(element) と呼ばれます。

ひとつの要素は、一組の開始タグと終了タグのペアによって作られるか、または、ひとつの空要素タグによって作られます。開始タグと終了タグのペアによって作られる要素というのは、

開始タグ | テキスト | 終了タグ

という形のテキストのことです。たとえば、

---

[1] W3C の公式サイトの URL は、`http://www.w3.org/` です。

[2] 「テキスト」というのは、「文字列」と同じ意味の言葉だと考えていただいていいでしょう。

```
<renkon>私はこの要素の内容です。</renkon>
```

というのは、開始タグと終了タグのペアによって作られた要素の例です。開始タグと終了タグのあいだに書かれたテキストは、要素の「内容」(content) と呼ばれます。上の例では、「私はこの要素の内容です。」というテキストが要素の内容になっています。

空要素タグは、それ自体がひとつの要素になります。たとえば、

```
<ninjin/>
```

というのは、空要素タグによって作られた要素の例です。空要素タグによって作られた要素は、内容を持ちません。

要素は、かならず何らかの種類に属しています。要素の種類は、「要素型」(element type) と呼ばれます。要素型は、「要素型名」(element type name) と呼ばれる名前によって識別されます。

タグの中には、かならず要素型名を書く必要があります。要素の要素型は、その要素を作るために書かれたタグの中の要素型名によって決定されます。

1.1.5 属性

要素は、1 個のテキストを保持することができる容器のようなものをいくつでも持つことができます。それらの容器は、「属性」(attribute) と呼ばれます。属性は、「属性名」(attribute name) と呼ばれる名前によって識別されます。

それぞれの属性が保持しているテキストは、その属性の「属性値」(attribute value) と呼ばれます。

要素が持っている属性に対してテキストを設定したいときは、その要素の開始タグまたは空要素タグの中に、「属性指定」(attribute specification) と呼ばれるテキストを書きます。属性指定というのは、

属性名 =" テキスト "

という構文を持つテキストのことです。このようなテキストを書くことによって、二重引用符 (double quote, ") で囲まれた内側にあるテキストを、属性名で指定された属性に対して属性値として設定することができます。

たとえば、distance という名前の属性に 83km という属性値を設定したいという場合は、

```
distance="83km"
```

という属性指定を書きます。

属性指定を書くことのできる場所は、開始タグまたは空要素タグの要素型名の右側です。ひとつのタグの中には、属性指定を何個でも書くことができます。たとえば、

```
<student grade="3" class="d" no="28" sex="female"
    name="Mita Hiroko" tel="0874-20-9517"/>
```

というのは、6 個の属性指定を含む空要素タグの例です。

要素型名と属性指定とのあいだと、属性指定と属性指定とのあいだには、少なくとも 1 個、「ホワイトスペース」(whitespace) と呼ばれる文字を書く必要があります。ホワイトスペースというのは、文書のレイアウトに使われる、それ自体の形を持たない文字のことです。XML 文書では、空白 (space)、タブ (tab)、復帰 (carriage return)、改行 (line feed) という文字がホワイトスペースとして使われます。

1.1.6 注釈

XML 文書を書くとき、その文書を読む人間（自分自身も含んでいます）に伝えたいことを、その文書の一部分として書いておきたいけれども、その文書を処理するプログラムがその部分を処理してしまわないようにしたい、ということがしばしばあります。そのようなテキストの部分は、「注釈」(comment) と呼ばれます。

注釈を書くときは、XML 文書を処理するプログラムが、「ここからここまでは注釈だ」ということを認識することができるように、XML の文法にしたがって書く必要があります。XML の文法では、注釈は、

<!-- テキスト -->

というように、<!-- で始まって --> で終わるように書く、と定められています。<!-- と --> とのあいだには、基本的にはどんなテキストを書いてもかまわないのですが、2 個の連続するマイナス、つまりマイナスマイナス (--) というテキストを含むテキストを書く、ということだけはできません。

注釈は、基本的には、XML 文書の中のどこに書いてもかまいません。ただし、タグの中に注釈を書くことはできません。

1.1.7　XML 文書

XML 文書は三つの部分から構成されます。それらの部分は、「XML 宣言」(XML declaration)、「文書型宣言」(document type declaration)、「ルート要素」(root element) と呼ばれます。

XML 宣言というのは、使用している XML のバージョンなどに関する記述のことです。XML 宣言は、たとえば、

```
<?xml version="1.0"?>
```

というように書きます。

文書型宣言というのは、使用している XML 応用言語に関する記述のことです。たとえば、XHTML 1.1 という XML 応用言語を使っている場合、文書型宣言は、

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
    "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
```

というように書きます。

ルート要素というのは、XML 文書の本体に相当する部分です。ルート要素は、ひとつの XML 文書の中にかならずひとつだけ存在していないといけません。

1.1.8　名前空間

ひとつの文書の中で使われる XML 応用言語は、ひとつだけとは限りません。一般に、複数の言語がひとつの文書の中で使われる場合には、それぞれの言語で定義された同一の名前をどのように区別すればいいのかということが問題になります。

XML では、異なる XML 応用言語のそれぞれで定義された同一の名前がひとつの文書の中で衝突することを避けるために、「名前空間」(namespace) と呼ばれるものが使われます。

名前空間というのは、個々の XML 応用言語で定義される要素型名や属性名などを囲い込む領域のことです。異なる XML 応用言語のそれぞれで同一の名前が定義されていたとしても、それらの名前は、名前空間によって区別することができます。

名前空間は、「名前空間名」(namespace name) と呼ばれる名前によって識別されます。たとえば、XHTML で使われる要素型名や属性名は、

```
http://www.w3.org/1999/xhtml
```

という名前空間名を持つ名前空間の中にあります。

名前空間の中にある要素型名や属性名は、XML 文書の中では、「QName」(qualified name) と呼ばれる名前によって識別されます。QName は、たとえば、

```
xhtml:html
```

というように、二つの名前をコロン (:) で結合した形になっています。コロンの左側の部分は名前空間を識別するための名前で、「名前空間接頭辞」(namespace prefix) と呼ばれます。そしてコロンの右側の部分は名前空間の中にある名前で、「ローカル部分」(local part) と呼ばれます。

1.1.9　名前空間宣言

名前空間接頭辞を使うためには、それと名前空間名とを結び付ける必要があります。それらは、「名前空間宣言」(namespace declaration) と呼ばれる特殊な属性指定を書くことによって結び付けることができます。名前空間宣言というのは、

xmlns:[名前空間接頭辞]="[名前空間名]"

という構文を持つ属性指定のことです。たとえば、

```
xmlns:xhtml="http://www.w3.org/1999/xhtml"
```

という名前空間宣言を書くことによって、xhtml という名前空間接頭辞と、XHTML の名前空間名とを結び付けることができます。

名前空間宣言を要素の開始タグの中に書いておくと、その開始タグで始まる要素自身と、その中にある要素で、その名前空間接頭辞を使うことができるようになります。たとえば、

```
<xhtml:html xmlns:xhtml="http://www.w3.org/1999/xhtml">
```

という開始タグを書くことによって、この開始タグで始まる要素自身と、その中にある要素で、xhtml という名前空間接頭辞を使うことができるようになります。

1.1.10 デフォルト名前空間

XML 文書の中で、QName ではない単独の名前を書いたとしても、それが特定の名前空間の中にある名前として解釈されるようにする、という設定をすることも可能です。そのように設定された名前空間は、「デフォルト名前空間」(default namespace) と呼ばれます。

何らかの名前空間をデフォルト名前空間にするという設定をしたいときは、「デフォルト名前空間宣言」(default namespace declaration) と呼ばれる属性指定を書きます。デフォルト名前空間宣言というのは、

```
xmlns="名前空間名"
```

という形の属性指定のことです。たとえば、

```
xmlns="http://www.w3.org/1999/xhtml"
```

というデフォルト名前空間宣言を書くことによって、XHTML の名前空間をデフォルト名前空間にすることができます。

1.2 SVG の基礎の基礎

1.2.1 SVG とは何か

SVG というのは、XML 応用言語のひとつで、平面（つまり 2 次元空間）の上のグラフィックスを記述することを目的として設計されたものです。SVG で書かれたテキストは、「SVG 文書」(SVG document) と呼ばれます。

SVG という名前は、Scalable Vector Graphics という言葉から作られた頭字語です。この名前は、SVG によって記述されたグラフィックスが scalable であるということ（つまり拡大や縮小が可能であるということ）、そしてそれが vector graphics であるということ（つまり直線や曲線の組み合わせであるということ）を意味しています。

SVG は、XML と同様に、W3C(World Wide Web Consortium) によって定められた勧告になっています。

1.2.2 ウェブブラウザー

ウェブを閲覧するために使われるソフトウェアは、「ウェブブラウザー」(web browser) と呼ばれます（単に「ブラウザー」と呼ばれることもあります）。ウェブブラウザーとしては、Opera、Firefox、Safari、Amaya、Google Chrome などがあります。

ウェブブラウザーは、ウェブを構成しているさまざまなデータを表示する機能を持っています。SVG というのもウェブを作るために使われる言語のひとつですから、ウェブブラウザーの大多数は、SVG によって記述されたグラフィックスを表示する機能を持っています。

この文章（「SVG 実習マニュアル」）の中に書かれている SVG 文書は、記述されたグラフィックスが正しく表示されるかどうかを、Opera というウェブブラウザーを使って確認しています。ですから、それらの SVG 文書の中には、Opera 以外のウェブブラウザーでは正しく表示されないものも含まれている可能性があります。

1.2.3 SVG エディター

SVG 文書を扱うソフトウェアとしては、ウェブブラウザーのほかに、SVG 文書によって記述されているグラフィックスを対話的に編集するソフトウェアというものもあります。そのようなソフトウェアは、「SVG エディター」(SVG editor) と呼ばれます。SVG エディターとしては、Inkscape、SVG Cats、Sketsa SVG Editor などがあります。

1.2.4 SVG 文書の作成と表示

SVG 文書は、SVG エディターを使って作成することもできますが、SVG 文書というのは単なるテキストデータですので、テキストエディターを使って作成することも可能です。

それでは、実際に、テキストエディターを使って簡単な SVG 文書を入力して、それをウェブブラウザーに表示させる、ということを試してみましょう。

まず、テキストエディターを使って次の SVG 文書を入力してください。

SVG 文書の例 sample.svg

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="10cm" height="7cm"
    xmlns="http://www.w3.org/2000/svg">
  <rect x="1.5cm" y="1cm" width="7cm" height="4cm"/>
</svg>
```

入力できたら、それを sample.svg という名前のファイルに保存してください。このように、SVG 文書を格納するファイルには、.svg という拡張子を付けます。

入力と保存ができたら、それをウェブブラウザーで開いてみてください。SVG 文書に間違いがなければ、ウェブブラウザーのウィンドウの中に 1 個の長方形が表示されるはずです。

1.2.5 SVG 文書の構成

SVG というのは XML 応用言語のひとつですから、SVG 文書は XML 文書の一種です。ですからそれは、XML 宣言、文書型宣言、ルート要素、という三つの部分から構成されます。

それでは、もう一度、先ほど入力した sample.svg を見てください。その 1 行目にある、

```
<?xml version="1.0" standalone="no"?>
```

というのが XML 宣言で、2 行目と 3 行目にある、

```
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
```

というのが文書型宣言です。そして、それらの下にある、

```
<svg width="10cm" height="7cm"
    xmlns="http://www.w3.org/2000/svg">
  <rect x="1.5cm" y="1cm" width="7cm" height="4cm"/>
</svg>
```

という部分がルート要素です。この例のように、SVG 文書では、svg という要素型の要素がルート要素になります。また、SVG の要素型名は、

```
http://www.w3.org/2000/svg
```

という名前の名前空間の中にありますので、SVG 文書のルート要素の開始タグには、通常、この名前空間をデフォルト名前空間にするためのデフォルト名前空間宣言を書きます。

sample.svg では、ルート要素の内容として、

```
<rect x="1.5cm" y="1cm" width="7cm" height="4cm"/>
```

という要素が書かれています。この rect という要素型の要素は、長方形を描画するためのものです。この例の場合は、画面の左端から 1.5cm、画面の上端から 1cm の位置に左上の頂点があって、横の長さが 7cm、縦の長さが 4cm であるような長方形を描画する、という意味です。

なお、長方形の描画については、第 2.1 節で、もう少し詳しく説明したいと思います。

1.3 座標系

1.3.1 キャンバスとビューポート

SVG では、グラフィックスがその上に描画される平面のことを「キャンバス」(canvas) と呼びます。SVG のキャンバスは無限の広さを持っているため、その全体をウェブブラウザーに表示させることはできません。ウェブブラウザーは、有限の広さの長方形の領域をキャンバスから切り取って、その部分だけを表示します。ウェブブラウザーがキャンバスの一部分を切り取る長方形の領域は、「ビューポート」(viewport) と呼ばれます。

ビューポートの大きさは、SVG 文書のルート要素（つまり svg 要素）の開始タグの中に属性指定を書くことによって設定します。ビューポートの大きさを設定する属性は、次の二つです。

`width` 横の長さ。

`height` 縦の長さ。

次の SVG 文書は、ビューポートの横の長さを 8cm、縦の長さを 12cm に設定して、それと同じ大きさの長方形を描画します。

SVG 文書の例 `viewport.svg`

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="8cm" height="12cm"
    xmlns="http://www.w3.org/2000/svg">
  <rect x="0cm" y="0cm" width="8cm" height="12cm"/>
</svg>
```

1.3.2 長さの記述

SVG では、長さは、10 進数の右側に単位を書いたものによってあらわされます。単位としては次のようなものを使うことができます。

`px` ピクセル。省略可。

`mm` ミリメートル。`1mm＝3.543307px`

`cm` センチメートル。`1cm＝35.43307px`

`in` インチ。`1in＝90px`

`pt` ポイント。`1pt＝1.25px`

`pc` パイカ。`1pc＝15px`

`em` 現在のフォントの大きさ。

`ex` 現在のフォントでの文字 x の高さ。

`%` ビューポートの大きさに対する百分率。

たとえば、

```
width="238pt"
```

という属性指定を書くことによって、`width` という属性に 238 ポイントという長さを設定することができます。

次の SVG 文書は、ビューポートの横の長さを 320 ポイント、縦の長さを 224 ポイントに設定して、それと同じ大きさの長方形を描画します。

SVG 文書の例 `lenunit.svg`

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="320pt" height="224pt"
    xmlns="http://www.w3.org/2000/svg">
  <rect x="0pt" y="0pt" width="320pt" height="224pt"/>
</svg>
```

`px` という単位は、基本的には物理的な装置のピクセルによって決定されるのですが、実際には SVG 文書の中で自由に定義することが可能です。そして、`mm`、`cm`、`in`、`pt`、`pc` は、`px` に連動して定義されます。

なお、`px` という単位は省略することができますので、`38px` という長さは、`38` と書いても同じ意味になります。

1.3.3 デフォルトの座標系

SVG では、キャンバスの上での位置は、x 軸と y 軸による座標系を使って指定されます。

デフォルトでは、座標系の原点はビューポートの左上の隅にあって、座標軸の向きは、x 軸が右向き、y 軸が下向きになっています。

1.3.4 px と原点の定義

px という単位の長さと座標系の原点は、

viewBox="数値 数値 数値 数値"

という属性指定を書くことによって自由に定義することができます。

viewBox という属性に設定する属性値は、4 個の数値を空白で区切ったものです。

px という単位の長さを定義したいときは、定義後の px を使ってビューポートの横の長さと縦の長さを計った結果を、viewBox に 3 個目と 4 個目の数値として設定します。たとえば、

```
viewBox="0 0 700 500"
```

という属性指定を書いたとすると、ビューポートの横の長さが 700、縦の長さが 500 になるように、px の長さを定義します。

次の SVG 文書は、ビューポートの横の長さが 3、縦の長さが 2 になるように px の長さを定義して、ビューポートと同じ大きさの長方形を描画します。

SVG 文書の例 definepx.svg

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="300pt" height="200pt" viewBox="0 0 3 2"
    xmlns="http://www.w3.org/2000/svg">
  <rect x="0" y="0" width="3" height="2"/>
</svg>
```

座標系の原点を定義したいときは、定義後の原点を使ってビューポートの左上の隅をあらわす座標を求めて、その x 座標と y 座標を viewBox に 1 個目と 2 個目の数値として設定します。たとえば、

```
viewBox="-30 -60 700 500"
```

という属性指定を書いたとすると、ビューポートの左上の隅の x 座標が -30、y 座標が -60 になるように原点を定義します。

次の SVG 文書は、ビューポートの左上の隅の x 座標が -20、y 座標が -10 になるように原点を定義して、ビューポートと同じ大きさの長方形を描画します。

SVG 文書の例 origin.svg

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="300pt" height="200pt" viewBox="-20 -10 300 200"
    xmlns="http://www.w3.org/2000/svg">
  <rect x="0" y="0" width="300" height="200"/>
</svg>
```

1.4 色

1.4.1 塗りつぶしの色

rect のような、グラフィックスを描画する要素は、fill という属性を持っています。この属性に対して色を記述した文字列を設定すると、その色で内部が塗りつぶされたグラフィックスが描画されます。

SVG では、色を記述する方法として、

- 色名を使う方法
- 16 進数を使う方法
- 10 進数を使う方法

という 3 種類のものを使うことができます。

1.4.2 色名

「色名」(color keyword) というのは、色を識別するための英語の単語のことです。たとえば、紫色は、purple という色名であらわされます。SVG で使うことのできる色名は、SVG の勧告に記載されていますので、必要に応じてそちらを参照してください。

次の SVG 文書は、三つの長方形のそれぞれを、

赤	red
オレンジ色	orange
黄色	yellow

で描画します。

SVG 文書の例 colword.svg

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="320pt" height="224pt" viewBox="0 0 100 70"
    xmlns="http://www.w3.org/2000/svg">
  <rect x="5" y="5" width="70" height="40" fill="red"/>
  <rect x="15" y="15" width="70" height="40" fill="orange"/>
  <rect x="25" y="25" width="70" height="40" fill="yellow"/>
</svg>
```

この SVG 文書をウェブブラウザーで開くと、三つの長方形が重なって表示されます。その重なり方から、SVG 文書の中に記述されているそれぞれのグラフィックスはどのような順序で描画されるのか、ということが判ります。

それぞれのグラフィックスは、それを記述している要素が上にあるものから下にあるものへという順序で描画されます。あとから描画されたグラフィックスは、それ以前に描画された、同じ位置にあるグラフィックスを覆い隠すことになります。

1.4.3 16 進数による色の記述

16 進数で色を記述したいときは、

#赤緑青

という形で、光の三原色のそれぞれの色の強さを書きます。「赤」「緑」「青」というそれぞれの場所には、その原色の光を混ぜ合わせる強さを、2 桁の 16 進数で記述します。たとえば、紫色を 16 進数で記述すると、

```
#800080
```

という文字列になります。

次の SVG 文書は、三つの長方形のそれぞれを、

ライム	lime	#00ff00
スプリンググリーン	springgreen	#00ff7f
水色	aqua	#00ffff

で描画します。

SVG 文書の例 colhex.svg

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="320pt" height="224pt" viewBox="0 0 100 70"
    xmlns="http://www.w3.org/2000/svg">
  <rect x="5" y="5" width="70" height="40" fill="#00ff00"/>
  <rect x="15" y="15" width="70" height="40" fill="#00ff7f"/>
  <rect x="25" y="25" width="70" height="40" fill="#00ffff"/>
</svg>
```

1.4.4　10 進数による色の記述

10 進数で色を記述したいときは、

rgb(赤, 緑, 青)

という形で、光の三原色のそれぞれの色の強さを書きます。「赤」「緑」「青」というそれぞれの場所には、その原色の光を混ぜ合わせる強さを、0 から 255 までのあいだの 10 進数で書きあらわします。たとえば、紫色を 10 進数で記述すると、

```
rgb(128, 0, 128)
```

という文字列になります。

次の SVG 文書は、三つの長方形のそれぞれを、

青	blue	rgb(0, 0, 255)
ブルーバイオレット	blueviolet	rgb(138, 43, 226)
マゼンタ	magenta	rgb(255, 0, 255)

で描画します。

SVG 文書の例　coldec.svg

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="320pt" height="224pt" viewBox="0 0 100 70"
    xmlns="http://www.w3.org/2000/svg">
  <rect x="5" y="5" width="70" height="40"
      fill="rgb(0, 0, 255)"/>
  <rect x="15" y="15" width="70" height="40"
      fill="rgb(138, 43, 226)"/>
  <rect x="25" y="25" width="70" height="40"
      fill="rgb(255, 0, 255)"/>
</svg>
```

1.4.5　線

グラフィックスを描画する要素は、色を設定する属性を、fill のほかにもうひとつ持っています。それは、stroke という属性です。この属性に対して色を記述した文字列を設定すると、グラフィックスの輪郭をあらわす線が、その色で描画されます。

ただし、グラフィックスの輪郭をあらわす線を描画するためには、その線の幅を設定する必要があります。

グラフィックスを描画する要素が持っている stroke-width という属性に長さを設定すると、その長さが、線の幅として設定されます。

次の SVG 文書は、長方形の輪郭をあらわす、幅が 6 の線を描画します。

SVG 文書の例　stroke.svg

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="320pt" height="224pt" viewBox="0 0 100 70"
    xmlns="http://www.w3.org/2000/svg">
  <rect x="10" y="10" width="80" height="50"
      stroke-width="6" stroke="navy" fill="lavender"/>
</svg>
```

1.4.6　塗りつぶしの省略

stroke の属性指定を省略した場合は、グラフィックスの輪郭を描画しないで、内部が塗りつぶされるだけになります。それとは逆に、内部を塗りつぶさないで輪郭だけを描画したいときは、fill 属性に対して none という属性値を設定します。

次の SVG 文書が描画する三つの長方形のうちで、もっとも右下にあるものは、内部が塗りつぶされていません。

SVG 文書の例　none.svg

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="320pt" height="224pt" viewBox="0 0 100 70"
    xmlns="http://www.w3.org/2000/svg">
  <rect x="5" y="5" width="70" height="40" fill="aqua"/>
  <rect x="15" y="15" width="70" height="40"
      stroke-width="4" stroke="teal" fill="white"/>
  <rect x="25" y="25" width="70" height="40"
      stroke-width="4" stroke="turquoise" fill="none"/>
</svg>
```

1.4.7 不透明度

グラフィックスの上に別のグラフィックスを重ねて描画した場合に、下にあるグラフィックスがどれくらい明瞭に見えるかというのは、上に重なっているグラフィックスの不透明度 (opacity) によって決まります。

不透明度は、0 から 1 までの数値であらわされます。0 というのは完全に透明という意味で、1 というのは完全に不透明という意味です。

グラフィックスの内部の不透明度は、opacity という属性に設定された数値によって決定されます。

SVG 文書の例 opacity.svg

```
<?xml version="1.0" standalone="no" ?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="320pt" height="224pt" viewBox="0 0 100 70"
    xmlns="http://www.w3.org/2000/svg">
  <rect x="10" y="20" width="80" height="30" fill="blue"/>
  <rect x="15" y="10" width="10" height="50" fill="lime"
    opacity="1.0"/>
  <rect x="35" y="10" width="10" height="50" fill="lime"
    opacity="0.8"/>
  <rect x="55" y="10" width="10" height="50" fill="lime"
    opacity="0.6"/>
  <rect x="75" y="10" width="10" height="50" fill="lime"
    opacity="0.4"/>
</svg>
```

同じように、グラフィックスの輪郭をあらわす線の不透明度は、stroke-opacity という属性に設定された数値によって決定されます。

SVG 文書の例 stropac.svg

```
<?xml version="1.0" standalone="no" ?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="320pt" height="224pt" viewBox="0 0 100 70"
    xmlns="http://www.w3.org/2000/svg">
  <rect x="10" y="20" width="80" height="30" fill="blue"/>
  <rect x="15" y="10" width="10" height="50" fill="none"
    stroke-width="6" stroke="lime" stroke-opacity="1.0"/>
  <rect x="35" y="10" width="10" height="50" fill="none"
    stroke-width="6" stroke="lime" stroke-opacity="0.8"/>
  <rect x="55" y="10" width="10" height="50" fill="none"
    stroke-width="6" stroke="lime" stroke-opacity="0.6"/>
  <rect x="75" y="10" width="10" height="50" fill="none"
    stroke-width="6" stroke="lime" stroke-opacity="0.4"/>
</svg>
```

1.5 グループ

1.5.1 グループの基礎

SVG には、何個かのグラフィックスをひとつにまとめる、つまりグラフィックスのグループを作る、という機能があります。この機能を使えば、色とか線の幅とかの属性が共通する何個かのグラフィックスを描画する場合に、その記述を簡潔にすることができます。

また、グラフィックスのグループには、名前を付けるということも可能です。グループに名前を付けておくと、その名前を指定することによってグループを描画することができますので、ひとつのグループを何回も使い回しするということが簡単にできるようになります。

1.5.2 グループの作り方

グラフィックスのグループを作りたいときは、g という要素型の要素を書きます。g 要素の書き方は、いたって簡単で、

```
<g>
  グループを構成するグラフィックスの記述
</g>
```

というように書けばいいだけです。g 要素の中にさらに g 要素を書いてもかまいません。

g 要素の開始タグの中には、stroke-width や stroke や fill など、グラフィックスの色とか線の幅とかの属性指定を書くことができます。そうしておくと、g 要素の中の要素でそれらの属性指定を省略した場合、g 要素の開始タグで設定された属性値が使われることになります。

SVG 文書の例 group.svg

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="320pt" height="224pt" viewBox="0 0 100 70"
    xmlns="http://www.w3.org/2000/svg">
  <g stroke-width="3" stroke="blueviolet" fill="none">
    <rect x="10" y="20" width="40" height="30"/>
    <rect x="20" y="10" width="70" height="30"/>
    <rect x="60" y="30" width="20" height="30"
        stroke="saddlebrown"/>
  </g>
</svg>
```

1.5.3 グループの名前

グラフィックスのグループに名前を付けたいときは、g 要素ではなく symbol という要素型の要素を書きます。symbol も、g と同じように、開始タグと終了タグのペアによって作られる要素です。そして、その中に書かれたグラフィックスから構成されるグループを作るという点も、g と同じです。

グループに付ける名前は、id という属性に対して、属性値として設定します。つまり、symbol 要素の開始タグを、

```
<symbol id="名前">
```

と書くことによって、その名前をグループに付けることができるわけです。たとえば、

```
<symbol id="cross">
  <rect x="10" y="17" width="20" height="6"/>
  <rect x="17" y="10" width="6" height="20"/>
</symbol>
```

という symbol 要素を書くことによって、2 個の長方形から構成されるグループを作って、そのグループに cross という名前を付けることができます。

g とは違って、symbol は、グループを作ってそれに名前を付けるだけの要素ですので、symbol 要素を書いてグループを作っただけだと、そのグループは描画されません。symbol で作られたグループを描画するためには、グループをその名前で参照する必要があります。

1.5.4 グループの参照

グループを名前で参照したいときは、use という要素型の要素を書きます。use は、空要素タグで作られる要素です。use を作る空要素タグの中に、

xlink:href="#名前"

という属性指定を書くと、その属性値として設定した名前を持つグループが描画されます（名前の左側にシャープ (#) を書く必要がありますので、注意してください)。

xlink というのは、XLink という言語の名前空間に結び付けられる標準的な名前空間接頭辞です[3]。XLink の名前空間名は、

http://www.w3.org/1999/xlink

ですので、

xmlns:xlink="http://www.w3.org/1999/xlink"

という名前空間宣言を書くことによって、それらの名前空間名と名前空間接頭辞とを結び付けることができます。

use 要素は、x と y という属性を持っています。これらの属性に対して x 座標と y 座標を設定すると、その座標で指定された位置を原点とする座標系を使ってグループが描画されます。

SVG 文書の例　use.svg

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="320pt" height="224pt" viewBox="0 0 100 70"
    xmlns="http://www.w3.org/2000/svg"
    xmlns:xlink="http://www.w3.org/1999/xlink">
  <symbol id="cross" fill="cadetblue">
    <rect x="10" y="17" width="20" height="6"/>
    <rect x="17" y="10" width="6" height="20"/>
  </symbol>
  <use xlink:href="#cross"/>
  <use xlink:href="#cross" x="20" y="30"/>
  <use xlink:href="#cross" x="60" y="10"/>
</svg>
```

1.6 リンク

1.6.1 ハイパーテキスト

直線的な順序で並べられたデータではなくて、「リンク」(link) と呼ばれる関連付けをたどることによって自由な順序で見ていくことを可能にしたデータのことを、「ハイパーテキスト」(hypertext) と言います。ウェブというのは、ハイパーテキストの一例です。

画面に表示されたテキストやグラフィックスなどの一部分で、別のデータへのリンクの出発点となっているもののことを、「アンカー」(anchor) と呼びます。ウェブのブラウザーは、アンカーがマウスでクリックされた場合、そのアンカーから出発したリンクの先にあるデータにアクセスします。

アンカーは、自分と関連付けられているデータを参照するための情報を持っている必要があります。ウェブの場合、その情報は、URI(uniform resource identifier) と呼ばれる形式で記述されます。URI というのは、URN(uniform resource name) と URL(uniform resource locator) の総称です。URI の仕様は、RFC2396 という RFC[4]の中で定義されています。

1.6.2 リンクを作る要素

SVG でも、キャンバスの上のグラフィックスをアンカーとする別のデータへのリンクを作ることができるようになっています。リンクを作りたいときは、a という要素型の要素を使います。

[3]XLink については、http://www.w3.org/TR/xlink/を参照してください。

[4]RFC(request for comment) というのは、IETF(Internet Engineering Task Force) という組織が公開している、標準規格を定めた文書のことです。

a 要素は、開始タグと終了タグのペアで作られる要素です。a 要素の内容として何らかのグラフィックスを描画する要素を書くと、そのグラフィックスをアンカーとするリンクが作られることになります。

a 要素は、xlink:href という属性を持っています。この属性は、第 1.5 節で use 要素を紹介したときに説明したように、XLink という仕様の中で定義されているものです。

a 要素の xlink:href 属性に対して URI を属性値として設定すると、その要素によって作られたリンクは、その URI によって記述されたデータを参照することになります。

次の二つの SVG 文書はリンクによって相互に関連付けられていますので、それぞれの中のグラフィックスをマウスでクリックすると、表示が相手側に切り換わります。

SVG 文書の例　linka.svg

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="320pt" height="224pt" viewBox="0 0 100 70"
    xmlns="http://www.w3.org/2000/svg"
    xmlns:xlink="http://www.w3.org/1999/xlink">
  <a xlink:href="linkb.svg">
    <rect x="20" y="25" width="60" height="20"
        fill="darkturquoise"/>
  </a>
</svg>
```

SVG 文書の例　linkb.svg

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="320pt" height="224pt" viewBox="0 0 100 70"
    xmlns="http://www.w3.org/2000/svg"
    xmlns:xlink="http://www.w3.org/1999/xlink">
  <a xlink:href="linka.svg">
    <rect x="20" y="10" width="60" height="20"
        fill="mediumseagreen"/>
  </a>
  <a xlink:href="http://www.example.org/">
    <rect x="20" y="40" width="60" height="20"
        fill="hotpink"/>
  </a>
</svg>
```

第 2 章　形状

2.1　基本的な形状

2.1.1　基本的な形状の基礎

SVG では、長方形、円、楕円、直線、折れ線、多角形という 6 種類の形状のことを、「基本的な形状」(basic shapes) と呼びます。

基本的な形状は、それぞれの形状に対応する、次のような要素を書くことによって描画することができます。

rect	長方形。
circle	円。
ellipse	楕円。
line	直線。
polyline	折れ線。
polygon	多角形。

2.1.2 長方形

第 1.2 節でも簡単に説明しましたが、長方形を描画したいときは、rect という要素型の要素を書きます。この要素は、角張った長方形だけではなくて、角が丸い長方形を描画することもできます。

長方形の位置、大きさ、角を丸くする指定は、次の 6 個の属性に設定します。

- x　左上の頂点の x 座標。
- y　左上の頂点の y 座標。
- width　横の長さ。
- height　縦の長さ。
- rx　角を丸くする楕円の x 軸方向の半径。
- ry　角を丸くする楕円の y 軸方向の半径。省略した場合は、rx で設定した半径と同じ長さになる。

SVG 文書の例　rect.svg

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="320pt" height="224pt" viewBox="0 0 100 70"
    xmlns="http://www.w3.org/2000/svg">
  <rect x="5" y="5" width="30" height="60" rx="5" ry="10"
      stroke-width="2" stroke="darkgreen" fill="palegreen"/>
  <rect x="15" y="20" width="70" height="30" rx="10"
      fill="limegreen"/>
  <rect x="65" y="5" width="30" height="60" rx="10" ry="5"
      stroke-width="2" stroke="darkolivegreen" fill="none"/>
</svg>
```

2.1.3 円

円を描画したいときは、circle という要素型の要素を書きます。円の位置と大きさは、次の 3 個の属性に設定します。

- cx　中心の x 座標。
- cy　中心の y 座標。
- r　半径。

SVG 文書の例　circle.svg

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="320pt" height="224pt" viewBox="0 0 100 70"
    xmlns="http://www.w3.org/2000/svg">
  <circle cx="30" cy="25" r="20" fill="orange"/>
  <circle cx="50" cy="35" r="20" stroke-width="4"
      stroke="brown" fill="coral"/>
  <circle cx="70" cy="45" r="20" stroke-width="4"
      stroke="chocolate" fill="none"/>
</svg>
```

2.1.4 楕円

楕円を描画したいときは、ellipse という要素型の要素を書きます。楕円の位置と大きさは、次の 4 個の属性に設定します。

- cx　中心の x 座標。
- cy　中心の y 座標。
- rx　x 軸方向の半径。
- ry　y 軸方向の半径。

SVG 文書の例　ellipse.svg

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="320pt" height="224pt" viewBox="0 0 100 70"
    xmlns="http://www.w3.org/2000/svg">
  <ellipse cx="20" cy="35" rx="14" ry="28"
      fill="greenyellow"/>
  <ellipse cx="50" cy="35" rx="30" ry="20" stroke-width="3"
      stroke="yellowgreen" fill="springgreen"/>
  <ellipse cx="78" cy="35" rx="14" ry="28" stroke-width="3"
      stroke="forestgreen" fill="none"/>
</svg>
```

2.1.5　直線

直線を描画したいときは、line という要素型の要素を書きます。次の 4 個の属性に 2 個の点の位置を設定すると、それらの点をつなぐ直線が描画されます。

x1　1 個目の点の x 座標。

y1　1 個目の点の y 座標。

x2　2 個目の点の x 座標。

y2　2 個目の点の y 座標。

line 要素を作るタグの中にも fill の属性指定を書くことは可能ですが、直線にはもともと内部というものがありませんので、それを書いたとしても、どこかが塗りつぶされるということはありません。

SVG 文書の例　line.svg

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="320pt" height="224pt" viewBox="0 0 100 70"
    xmlns="http://www.w3.org/2000/svg">
  <line x1="10" y1="10" x2="90" y2="60" stroke-width="10"
      stroke="crimson"/>
</svg>
```

2.1.6　折れ線

折れ線を描画したいときは、polyline という要素型の要素を書きます。polyline 要素が持っている points という属性に 2 個以上の点の位置を設定すると、それらの点をつないでいくことによってできる折れ線が描画されます。

points という属性に設定する属性値の中には、

数値 , 数値　数値 , 数値 ･･･

という形で、点の位置の x 座標と y 座標を並べていきます。たとえば、

```
points="10,20 40,30 70,10"
```

と書けば、1 個目の点が (10,20)、2 個目の点が (40,30)、3 個目の点が (70,10) という意味になります。

SVG 文書の例　polyline.svg

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="320pt" height="224pt" viewBox="0 0 100 70"
    xmlns="http://www.w3.org/2000/svg">
  <polyline points="20,50 10,40 10,20 20,10 40,10 50,20
      50,50 60,60 80,60 90,50 90,30 80,20 60,20 40,50"
      stroke-width="4" stroke="tomato" fill="khaki"/>
</svg>
```

2.1.7 多角形

多角形を描画したいときは、polygon という要素型の要素を書きます。polygon も、polyline と同じように points という属性を持っています。多角形の頂点の位置を points に設定すると、それらの頂点から構成される多角形が描画されます。

SVG 文書の例 polygon.svg

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="320pt" height="224pt" viewBox="0 0 100 70"
    xmlns="http://www.w3.org/2000/svg">
  <polygon points="20,50 10,40 10,20 20,10 40,10 50,20
      50,50 60,60 80,60 90,50 90,30 80,20 60,20 40,50"
      stroke-width="4" stroke="indianred" fill="mistyrose"/>
</svg>
```

2.2 パス

2.2.1 パスとは何か

直線または曲線から構成される形状のことを「パス」(path) と呼びます。

パスを作るためには、それを記述したデータが必要です。パスを記述したデータは、「パスデータ」(path data) と呼ばれます。パスデータは、直線または曲線の作り方に関する指示を書きあらわしたものです。

2.2.2 パスを描画する要素

パスは、path という要素型の要素を書くことによって、キャンバスの上に描画することができます。path 要素は、空要素タグを書くことによって作ります。

基本的な形状を描画する要素と同じように、path 要素も、次の属性を持っています。

fill	内部の色。
stroke-width	線の幅。
stroke	線の色。
opacity	内部の不透明度。
stroke-opacity	線の不透明度。

path 要素は、さらに、d という名前の属性を持っています。この属性にはパスデータを設定します。path 要素は、d 属性に設定されたパスデータを解釈して、その指示にしたがってパスを構築します。

さて、それでは、次の SVG 文書をウェブブラウザーに表示させてみてください（パスデータの書き方については、そのあとで説明します)。

SVG 文書の例 path.svg

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="320pt" height="224pt" viewBox="0 0 100 70"
    xmlns="http://www.w3.org/2000/svg">
  <path d="M10,10 L70,10 C140,80 -20,70 50,30"
      stroke-width="4" stroke="chocolate" fill="aquamarine"/>
</svg>
```

2.2.3 パスデータの基礎

パスデータは、「コマンド」(command) と呼ばれる記述を並べたものです。コマンドというのは、

コマンド名 引数 ・・・

という形の記述のことです。コマンドとコマンドとのあいだ、コマンド名と引数とのあいだ、引数と引数とのあいだには、任意の個数のホワイトスペースを挿入することができます。

引数と引数とのあいだを、ホワイトスペースではなくてコンマで区切ってもかまいません。ですから、引数として座標を書く場合は、x 座標と y 座標を、

$\boxed{x\text{ 座標}}$, $\boxed{y\text{ 座標}}$

というようにコンマで区切って並べると、コマンドが読みやすくなります。

コマンド名は、すべて、1 文字の英字で作られています。どのコマンドについても大文字と小文字のコマンド名がペアになっていて、大文字のコマンド名は絶対座標系のコマンドを、小文字のコマンド名は相対座標系のコマンドをあらわしています。相対座標系というのは、「カレントポイント」(current point) と呼ばれる点を原点とする座標系のことです。

2.2.4 カレントポイントの移動

最初のコマンドとして、M または m というコマンドについて説明しましょう。M(m) コマンドは、カレントポイントを移動させるという動作をあらわします。このコマンドに書く引数は、カレントポイントを移動させる移動先の座標です。たとえば、

```
M200,100
```

というコマンドは、カレントポイントを絶対座標系の (200,100) へ移動させます。同じように、

```
m200,-50
```

というコマンドは、カレントポイントを、x 軸の方向へ 200、y 軸の方向へ -50 だけ移動させます。

大文字の M は、カレントポイントが存在していないときに実行された場合は、指定された位置を持つ新しいカレントポイントを作ります。

パスデータは、先頭のコマンドから順番に実行されていくのですが、その先頭のコマンドが実行される直前では、まだカレントポイントが存在しない状態になっています。その状態で実行され得るコマンドは、大文字の M だけです。つまり、パスデータというものは、かならず M コマンドで始まる必要があるわけです。

2.2.5 直線の追加

L または l というコマンドは、パスに直線を追加するという動作をあらわします。L(l) コマンドに書く引数は、1 個の座標です。L(l) コマンドは、カレントポイントと引数とをつなぐ直線をパスに追加します。

L(l) コマンドは、直線をパスに追加したのち、追加した直線の終了点（つまり引数で指定された点）へカレントポイントを移動させます。

SVG 文書の例 lineto.svg

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="320pt" height="224pt" viewBox="0 0 100 70"
    xmlns="http://www.w3.org/2000/svg">
  <path d="M30,15 l40,0 l0,40 l-40,0 l0,-40"
      stroke-width="10" stroke="darkslateblue" fill="none"/>
</svg>
```

2.2.6 パスを閉じる

パスの開始点と終了点とを同一のものにして、それを単なる頂点のひとつにすることを、パスを「閉じる」(close) と言います。パスを閉じたいときは、Z または z というコマンドを書きます。

Z(z) コマンドには、引数をひとつも書きません。大文字の Z も小文字の z も動作はまったく同じで、カレントポイントとパスの開始点とをつなぐ直線をパスに追加して、パスを閉じます。

SVG 文書の例 close.svg

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="320pt" height="224pt" viewBox="0 0 100 70"
    xmlns="http://www.w3.org/2000/svg">
  <path d="M30,15 l40,0 l0,40 l-40,0 Z"
```

```
      stroke-width="10" stroke="darkslateblue" fill="none"/>
</svg>
```

2.2.7 曲線の追加

C または c というコマンドは、「ベジェ曲線」(Bézier curve) と呼ばれる曲線をパスに追加するという動作をあらわします。C(c) コマンドに書く引数は、3 個の座標です。

C(c) コマンドによってパスに追加される曲線は、「制御点」(control point) と呼ばれる 4 個の点によって決定されます。4 個の制御点のそれぞれを、制御点 1、制御点 2、制御点 3、制御点 4、と呼ぶことにしましょう。

制御点 1 というのは、カレントポイントです。そして、C(c) コマンドの引数として書いた 3 個の座標が、順番に、制御点 2、制御点 3、制御点 4 になります。C(c) コマンドによってパスに追加される曲線は、まず制御点 1 から出発して、制御点 2 へ向かいます。しかし、少しずつ方向を変えていって、制御点 3 と制御点 4 とをつなぐ直線に接する形で制御点 4 に到達して、そこで終了します。

C(c) コマンドは、曲線をパスに追加したのち、追加した曲線の終了点（つまり制御点 4）へカレントポイントを移動させます。

SVG 文書の例　curveto.svg

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="320pt" height="224pt" viewBox="0 0 100 70"
    xmlns="http://www.w3.org/2000/svg"
    xmlns:xlink="http://www.w3.org/1999/xlink">
  <symbol id="curve1" stroke-width="7" stroke="palegreen"
      fill="none">
    <path d="M20,50 C10,25 50,10 40,50"/>
  </symbol>
  <symbol id="curve2" stroke-width="7" stroke="palegreen"
      fill="none">
    <path d="M20,50 C10,25 40,50 50,10"/>
  </symbol>
  <symbol id="control" stroke-width="0.5" stroke="deeppink">
    <line x1="20" y1="50" x2="10" y2="25"/>
    <line x1="50" y1="10" x2="40" y2="50"/>
  </symbol>
  <use xlink:href="#curve1"/>
  <use xlink:href="#control"/>
  <use xlink:href="#curve2" x="40"/>
  <use xlink:href="#control" x="40"/>
</svg>
```

2.2.8 楕円弧の追加

A または a というコマンドは、楕円弧をパスに追加するという動作をあらわします。A(a) コマンドには、次の 7 個の引数を書きます。

1 個目　x 軸方向の楕円弧の半径。
2 個目　y 軸方向の楕円弧の半径。
3 個目　楕円の x 軸の回転角度。
4 個目　1 ならば長弧 (large arc)、0 ならば短弧 (small arc)。
5 個目　1 ならば時計回り (clockwise)、0 ならば反時計回り (counterclockwise)。
6 個目　楕円弧が終了する位置の x 座標。
7 個目　楕円弧が終了する位置の y 座標。

A(a) コマンドによってパスに追加される楕円弧は、カレントポイントから開始されて、6 個目と 7 個目の引数で指定された位置で終了します。そののち、カレントポイントは、楕円弧が終了した位置へ移動します。

二つの点をつなぐ楕円弧には、4 とおりの候補があります。そのうちのどれなのかということを指定するのが、4 個目と 5 個目の引数です。長弧というのは 180 度以上の楕円弧のことで、短弧というのは 180 度未満の楕円弧のことです。

SVG 文書の例　arcto.svg

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="320pt" height="224pt" viewBox="0 0 100 70"
    xmlns="http://www.w3.org/2000/svg">
  <g stroke-width="2" stroke="orangered" fill="none">
    <path d="M10,35 a20,15 0 1,1 30,0"/>
    <path d="M60,10 a20,15 0 1,0 30,0"/>
    <path d="M10,55 a20,15 0 0,1 30,0"/>
    <path d="M60,50 a20,15 0 0,0 30,0"/>
  </g>
</svg>
```

2.3　テキスト

2.3.1　テキストを描画する要素

テキストを描画したいときは、text という要素型の要素を書きます。text 要素は、空要素タグではなくて、開始タグと終了タグのペアを使って作ります。そうすると、text 要素の内容、つまり開始タグと終了タグとのあいだに書かれたテキストが描画されることになります。

基本的な形状を描画する要素や path 要素と同じように、text 要素も次の属性を持っています。

fill	内部の色。
stroke-width	線の幅。
stroke	線の色。
opacity	内部の不透明度。
stroke-opacity	線の不透明度。

描画されるテキストの位置や大きさは、text 要素が持っている次の属性に設定された属性値によって決定されます。

x	テキストの左端の x 座標。
y	テキストのベースラインの y 座標。
font-size	フォントの大きさ。数値で指定する。

SVG 文書の例　text.svg

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="320pt" height="224pt" viewBox="0 0 100 70"
    xmlns="http://www.w3.org/2000/svg">
  <text x="10" y="52" font-size="80"
      stroke-width="2" stroke="lightgreen" fill="aquamarine">
    xy
  </text>
  <text x="14" y="40" font-size="12" fill="forestgreen">
    I am a text.
  </text>
</svg>
```

2.3.2　フォントと装飾

text 要素が持っている属性のうちで、フォントと装飾に関連する属性としては、font-size のほかに次のようなものがあります。

font-family	フォントの名前。
font-weight	フォントの太さ。

font-style	フォントのスタイル。
text-decoration	テキストの装飾。

font-family 属性には、フォントの名前をホワイトスペースで区切って並べたものを設定します。

フォントに与えられた名前というのは、環境に依存します。しかし、「ジェネリックフォントファミリー名」(generic font family name) と呼ばれる、

serif sans-serif cursive fantasy monospace

というフォントの名前のいずれかを設定しておけば、どのような環境であっても最善のフォントが選択されます。

font-weight 属性には、normal や bold など、フォントの太さをあらわす記述を設定します。デフォルトは normal です。

font-style 属性には、normal 、italic 、oblique など、フォントのスタイルをあらわす記述を設定します。デフォルトは normal です。

text-decoration 属性には、テキストの装飾をあらわす次のような記述を設定します。

none	装飾なし。デフォルト。
underline	下線。
overline	上線。
line-through	取り消し線。

SVG 文書の例 font.svg

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="320pt" height="224pt" viewBox="0 0 100 70"
    xmlns="http://www.w3.org/2000/svg">
  <g font-size="8" font-family="serif" fill="dodgerblue">
    <text x="10" y="10">
      serif
    </text>
    <text x="10" y="20" font-family="monospace">
      monospace
    </text>
    <text x="10" y="30" font-weight="bold">
      serif bold
    </text>
    <text x="10" y="40" font-style="italic">
      serif italic
    </text>
    <text x="10" y="50" font-weight="bold"
        font-style="italic">
      serif bold italic
    </text>
    <text x="10" y="60" text-decoration="underline">
      serif underline
    </text>
  </g>
</svg>
```

2.3.3 単語の間隔と文字の間隔

text 要素が持っている次の属性に数値を設定することによって、単語の間隔と文字の間隔を指定することができます。マイナスの数値を設定することもできます。

word-spacing	単語の間隔。
letter-spacing	文字の間隔。

SVG 文書の例 space.svg

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
```

```
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="320pt" height="224pt" viewBox="0 0 100 70"
    xmlns="http://www.w3.org/2000/svg">
  <g fill="forestgreen" font-size="10" font-family="serif">
    <text x="10" y="14">I am a cat.</text>
    <text x="10" y="26" word-spacing="10">I am a cat.</text>
    <text x="10" y="38" word-spacing="-2">I am a cat.</text>
    <text x="10" y="50" letter-spacing="3">I am a cat.</text>
    <text x="10" y="62" letter-spacing="-1">I am a cat.</text>
  </g>
</svg>
```

2.3.4 テキストの一部分に対するスタイルの適用

tspan という要素型の要素を text 要素の子供として書くことによって、テキストの一部分に対して、その周囲とは異なるスタイル（色やフォントなど）を適用することができます。

tspan 要素は、開始タグと終了タグのペアで作ります。その要素の子供として書かれたテキストに対しては、その要素の属性に設定された属性値が適用されます。

SVG 文書の例 tspan.svg

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="320pt" height="224pt" viewBox="0 0 100 70"
    xmlns="http://www.w3.org/2000/svg">
  <g fill="navy" font-size="8" font-family="serif">
    <text x="10" y="10">
      I am <tspan fill="deeppink">not</tspan> a cat.
    </text>
    <text x="10" y="20">
      I am <tspan font-size="16">not</tspan> a cat.
    </text>
    <text x="10" y="30">
      I am <tspan font-family="san-serif">not</tspan> a cat.
    </text>
    <text x="10" y="40">
      I am <tspan font-weight="bold">not</tspan> a cat.
    </text>
    <text x="10" y="50">
      I am <tspan font-style="oblique">not</tspan> a cat.
    </text>
    <text x="10" y="60">
      I am <tspan text-decoration="underline">not</tspan>
      a cat.
    </text>
  </g>
</svg>
```

2.3.5 テキストの配置

デフォルトでは、text 要素の x 属性に設定された属性値は、テキストが開始される x 座標になります。ですから、何行かのテキストを左寄せにする場合は何の問題もないわけですが、それ以外の配置を指定するためには、そのための属性指定を書く必要があります。

テキストの配置は、text-anchor という属性に次のような単語を設定することによって指定することができます。

start	左寄せ。デフォルト。
middle	センタリング。
end	右寄せ。

SVG 文書の例 anchor.svg

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
```

```
<svg width="320pt" height="224pt" viewBox="0 0 100 70"
    xmlns="http://www.w3.org/2000/svg">
  <line x1="50" y1="5" x2="50" y2="65" stroke-width="0.2"
      stroke="red"/>
  <g fill="mediumblue" font-size="16" font-family="serif">
    <text x="50" y="20" text-anchor="start">start</text>
    <text x="50" y="40" text-anchor="middle">middle</text>
    <text x="50" y="60" text-anchor="end">end</text>
  </g>
</svg>
```

2.3.6 パスに沿ったテキスト

text 要素の子供として、textPath という要素型の要素を書くことによって、テキストをパスに沿って配置するということができます。

テキストをパスに沿って配置したいときは、まず、そのためのパスを path 要素を使って作ります。そのとき、path 要素の空要素タグの中に、

id="名前"

という属性指定を書くことによって、その path 要素に名前を付けておきます。

そして、textPath 要素の開始タグの中に、

xlink:href="#名前"

という属性指定で、path 要素の名前を xlink:href 属性に設定すると、それによって指定されたパスに沿ってテキストが配置されます。

パスに沿ったテキストを描画するだけで、パス自体を描画する必要はない、という場合は、

```
<defs>
  <path id="path1" d="M30,55 a35,23 0 1,1 40,0"/>
</defs>
```

というように、defs という要素の子供として path 要素を書きます。このように、defs 要素は、要素に名前を付けたいけれども、その要素の作用は抑制したい、というときに使います。

SVG 文書の例 textpath.svg

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="320pt" height="224pt" viewBox="0 0 100 70"
    xmlns="http://www.w3.org/2000/svg"
    xmlns:xlink="http://www.w3.org/1999/xlink">
  <defs>
    <path id="path1" d="M30,55 a35,23 0 1,1 40,0"/>
  </defs>
  <text font-size="9" font-family="serif" fill="cadetblue">
    <textPath xlink:href="#path1">
      Yoda: Do or do not. There is no try.
    </textPath>
  </text>
</svg>
```

2.4 座標系の変換

2.4.1 座標系の変換の基礎

SVG で形状を描画するときに使う座標系というのは、ひとつの SVG 文書の中ではひとつだけしか使えないというわけではなくて、必要に応じて変更を加えることができるようになっています。そのような変更のことを、座標系の「変換」(transformation) と呼びます。

座標系の変換には、移動 (translation)、拡大 (scaling)、回転 (rotation)、剪断 (せんだん) (skewing) などがあります。

座標系を変換したいときは、形状を描画する要素が持っている transform という属性に対して、どのような変換をしたいのかということを記述した属性値を設定します。たとえば、rect

要素を作る空要素タグの中に transform の属性指定を書けば、その長方形は、transform 属性に設定された記述によって変換された座標系を使って描画されることになります。

transform 属性の属性指定を書くことが可能な要素は、形状を描画する要素だけではなくて、g 要素や use 要素でも可能です。

2.4.2 座標系の移動

座標系の移動というのは、指定された距離だけ原点の位置を移動させるということです。

座標系を移動させたいときは、

translate(*tx*, *ty*)

という形の記述を、属性値として transform 属性に設定します。そうすると、x 軸方向に tx、y 軸方向に ty だけ原点を移動させた座標系が設定されます。たとえば、

```
transform="translate(40, 20)"
```

という属性指定は、座標系の原点を、x 軸方向に 40、y 軸方向に 20 だけ移動させるという意味になります。

SVG 文書の例 transla.svg

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="320pt" height="224pt" viewBox="0 0 100 70"
    xmlns="http://www.w3.org/2000/svg"
    xmlns:xlink="http://www.w3.org/1999/xlink">
  <symbol id="cross" stroke-width="3">
    <line x1="10" y1="20" x2="30" y2="20"/>
    <line x1="20" y1="10" x2="20" y2="30"/>
  </symbol>
  <use xlink:href="#cross" stroke="limegreen"/>
  <use xlink:href="#cross" transform="translate(20, 30)"
      stroke="navy"/>
  <g font-size="6" font-family="serif">
    <text x="24" y="28" fill="limegreen">original</text>
    <text x="44" y="58" fill="navy">translate(20, 30)</text>
  </g>
</svg>
```

2.4.3 座標系の拡大

座標系の拡大というのは、指定された拡大率で距離の単位を伸縮させるということです。

座標系を拡大したいときは、transform 属性に設定する属性値として、

scale(*sx*, *sy*)

という形のものを書きます。そうすると、距離の単位を x 軸方向に sx 倍、y 軸方向に sy 倍だけ拡大した座標系が設定されます。たとえば、

```
transform="scale(1.4, 0.8)"
```

という属性指定は、座標系の距離の単位を、x 軸方向に 1.4 倍、y 軸方向に 0.8 倍だけ拡大するという意味になります。

y 軸方向の拡大率を省略して、

scale(*sx*)

と書くと、y 軸方向の拡大率は x 軸方向と同じだと解釈されます。

SVG 文書の例 scale.svg

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="320pt" height="224pt" viewBox="0 0 100 70"
    xmlns="http://www.w3.org/2000/svg"
    xmlns:xlink="http://www.w3.org/1999/xlink">
  <symbol id="cross" stroke-width="3">
```

```
    <line x1="10" y1="50" x2="30" y2="50"/>
    <line x1="20" y1="40" x2="20" y2="60"/>
  </symbol>
  <use xlink:href="#cross" stroke="limegreen"/>
  <use xlink:href="#cross" transform="scale(3, 0.3)"
      stroke="navy"/>
  <g font-size="6" font-family="serif">
    <text x="24" y="58" fill="limegreen">original</text>
    <text x="60" y="24" fill="navy">scale(3, 0.3)</text>
  </g>
</svg>
```

2.4.4 座標系の回転

座標系の回転というのは、指定された点を中心にして、指定された角度だけ座標軸を回転させるということです。

座標系を回転させたいときは、transform 属性に設定する属性値として、

rotate(θ, cx, cy)

という形のものを書きます。そうすると、(cx, cy) という点を中心として、時計回りに θ 度だけ回転させた座標系が設定されます。たとえば、

```
transform="rotate(60, 40, 30)"
```

という属性指定は、(40,30) という点を中心として、時計回りに 60 度だけ座標系を回転させるという意味になります。

回転の角度をマイナスにすると、座標系は、時計回りではなく反時計回りで回転することになります。

SVG 文書の例 rotate.svg

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="320pt" height="224pt" viewBox="0 0 100 70"
    xmlns="http://www.w3.org/2000/svg"
    xmlns:xlink="http://www.w3.org/1999/xlink">
  <symbol id="cross" stroke-width="3">
    <line x1="20" y1="30" x2="60" y2="30"/>
    <line x1="40" y1="10" x2="40" y2="60"/>
  </symbol>
  <use xlink:href="#cross" stroke="limegreen"/>
  <use xlink:href="#cross" transform="rotate(60, 40, 30)"
      stroke="navy"/>
  <g font-size="6" font-family="serif">
    <text x="50" y="38" fill="limegreen">original</text>
    <text x="46" y="53" fill="navy">rotate(60, 40, 30)</text>
  </g>
</svg>
```

2.4.5 座標系の剪断

座標系の剪断というのは、指定された軸に沿って、指定された角度だけ座標系を傾けるということです。

x 軸に沿って座標系を剪断したいときは、transform 属性に設定する属性値として、

skewX(θ)

という形のものを書きます。そうすると、x 軸に沿って θ 度だけ剪断した座標系、つまり、y 軸を θ 度だけ傾けた座標系が設定されます。たとえば、

```
skewX(30)
```

という属性指定は、x 軸に沿って座標系を 30 度だけ剪断するという意味になります。同じように、

skewY(θ)

という形のものを書くことによって、y 軸に沿って θ 度だけ剪断した座標系、つまり、x 軸を θ 度だけ傾けた座標系を設定することができます。

SVG 文書の例 skew.svg

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="320pt" height="224pt" viewBox="0 0 100 70"
    xmlns="http://www.w3.org/2000/svg"
    xmlns:xlink="http://www.w3.org/1999/xlink">
  <symbol id="cross" stroke-width="3">
    <line x1="10" y1="20" x2="30" y2="20"/>
    <line x1="20" y1="10" x2="20" y2="30"/>
  </symbol>
  <use xlink:href="#cross" stroke="limegreen"/>
  <use xlink:href="#cross" transform="skewX(60)"
      stroke="navy"/>
  <use xlink:href="#cross" transform="skewY(50)"
      stroke="crimson"/>
  <g font-size="6" font-family="serif">
    <text x="24" y="28" fill="limegreen">original</text>
    <text x="55" y="15" fill="navy">skewX(60)</text>
    <text x="33" y="58" fill="crimson">skewY(50)</text>
  </g>
</svg>
```

2.4.6 変換の順序

transform 属性には、単独の変換の記述だけではなくて、いくつかの変換の記述をホワイトスペースで区切って並べたものを設定することも可能です。1 個の transform 属性に複数の変換の記述が設定されている場合、それぞれの記述は、入れ子になった複数の要素の transform 属性に設定されたものだとみなされます。たとえば、

```
<g transform="translate(50, 20) scale(0.2, 0.4)">
</g>
```

という記述があらわしている変換は、

```
<g transform="translate(50, 20)">
  <g transform="scale(0.2, 0.4)">
  </g>
</g>
```

という記述があらわしている変換と同じものになります。

ですから、1 個の transform 属性に設定されている複数の変換の記述は、右から左へ向かって順番に実行されると考えることができます。

SVG 文書の例 order.svg

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="320pt" height="224pt" viewBox="0 0 100 70"
    xmlns="http://www.w3.org/2000/svg"
    xmlns:xlink="http://www.w3.org/1999/xlink">
  <symbol id="cross" stroke-width="3">
    <line x1="10" y1="20" x2="30" y2="20"/>
    <line x1="20" y1="10" x2="20" y2="30"/>
  </symbol>
  <use xlink:href="#cross" stroke="limegreen"/>
  <use xlink:href="#cross" stroke="navy"
      transform="scale(3, 1) rotate(40, 20, 20)"/>
  <use xlink:href="#cross" stroke="crimson"
      transform="rotate(40, 20, 20) scale(3, 1)"/>
  <g font-size="4" font-family="serif">
    <text x="14" y="36" fill="limegreen">original</text>
```

```
    <text x="46" y="32" fill="navy">
      scale(3, 1) rotate(40, 20, 20)
    </text>
    <text x="12" y="60" fill="crimson">
      rotate(40, 20, 20) scale(3, 1)
    </text>
  </g>
</svg>
```

2.5 線の形状

2.5.1 端点の形状

線が始まる点と線が終わる点のことを、「端点」(end point) と呼びます。折れ線を描画した場合や、閉じていないパスの線を描画した場合、その両端は端点となります。

SVG では、stroke-linecap という属性に、端点の形状をあらわす記述を設定することによって、描画される端点の形状を指定することができるようになっています。

端点の形状をあらわす記述には、次の三つがあります。

butt	端点の位置で断ち切られた形。
round	端点の位置を中心として、線の幅の半分の半径を持つ円を描く形。
square	端点の位置から線の幅の半分だけはみ出した形。

SVG 文書の例 linecap.svg

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="320pt" height="224pt" viewBox="0 0 100 70"
    xmlns="http://www.w3.org/2000/svg"
    xmlns:xlink="http://www.w3.org/1999/xlink">
  <symbol id="hline" stroke-width="10" stroke="deepskyblue">
    <line x1="20" y1="15" x2="80" y2="15"/>
  </symbol>
  <g stroke-width="0.1" stroke="darkorange">
    <line x1="20" y1="5" x2="20" y2="65"/>
    <line x1="80" y1="5" x2="80" y2="65"/>
  </g>
  <use xlink:href="#hline" stroke-linecap="butt"/>
  <use xlink:href="#hline" y="20" stroke-linecap="round"/>
  <use xlink:href="#hline" y="40" stroke-linecap="square"/>
  <g font-size="6" font-family="serif" text-anchor="middle"
      fill="white">
    <text x="50" y="17">butt</text>
    <text x="50" y="37">round</text>
    <text x="50" y="57">square</text>
  </g>
</svg>
```

2.5.2 接続点の形状

線がそこで折れ曲がっている点のことを、「接続点」(join point) と呼びます。長方形や多角形の線を描画した場合や、閉じたパスの線を描画した場合、線が折れ曲がっている点は接続点となります。

SVG では、stroke-linejoin という属性に接続点の形状をあらわす記述を設定することによって、描画される接続点の形状を指定することができるようになっています。

接続点の形状をあらわす記述には、次の三つがあります。

miter	尖った形。
round	丸い形。
bevel	角を削った形。

SVG 文書の例 linejoin.svg

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="320pt" height="224pt" viewBox="0 0 100 70"
    xmlns="http://www.w3.org/2000/svg"
    xmlns:xlink="http://www.w3.org/1999/xlink">
  <symbol id="corner" stroke-width="8" stroke="greenyellow"
      fill="none">
    <polyline points="5,60 30,20 55,60"/>
  </symbol>
  <use xlink:href="#corner" stroke-linejoin="miter"/>
  <use xlink:href="#corner" x="20" stroke-linejoin="round"/>
  <use xlink:href="#corner" x="40" stroke-linejoin="bevel"/>
  <g font-size="6" font-family="serif" text-anchor="middle"
      fill="steelblue">
    <text x="30" y="10">miter</text>
    <text x="50" y="10">round</text>
    <text x="70" y="10">bevel</text>
  </g>
</svg>
```

2.5.3　破線

線と間隔が交互に繰り返されてできている線は、「破線」(dashed line) と呼ばれます。

SVG では、 stroke-dasharray という属性に、破線のパターンをあらわす記述を設定することによって、指定されたパターンで破線を描画することができるようになっています。

破線のパターンは、線の長さと間隔の長さを、コンマで区切って並べたものです。たとえば、

```
5, 2, 3, 4
```

という記述は、長さ 5 の線、長さ 2 の間隔、長さ 3 の線、長さ 4 の間隔、というパターンをあらわしています。

SVG 文書の例　dash.svg

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="320pt" height="224pt" viewBox="0 0 100 70"
    xmlns="http://www.w3.org/2000/svg"
    xmlns:xlink="http://www.w3.org/1999/xlink">
  <symbol id="stroke" stroke-width="5" stroke="midnightblue"
      fill="none">
    <line x1="10" y1="15" x2="90" y2="15"/>
  </symbol>
  <use xlink:href="#stroke"
      stroke-dasharray="2"/>
  <use xlink:href="#stroke" y="14"
      stroke-dasharray="2, 5"/>
  <use xlink:href="#stroke" y="28"
      stroke-dasharray="2, 5, 8"/>
  <use xlink:href="#stroke" y="42"
      stroke-dasharray="2, 5, 8, 3"/>
  <g font-size="6" font-family="serif" fill="coral">
    <text x="10" y="11">2</text>
    <text x="10" y="25">2, 5</text>
    <text x="10" y="39">2, 5, 8</text>
    <text x="10" y="53">2, 5, 8, 3</text>
  </g>
</svg>
```

第 3 章　装飾

3.1 グラディエント

3.1.1 グラディエントの基礎

線や領域に色を塗るときに、その全体を同じ色で塗るのではなくて、1 本のベクトルに沿っていくつかの色が連続的に変化するように塗ることを、「グラディエント」(gradient) と言います。グラディエントで使われる、連続的に変化する色が割り当てられたベクトルは、「グラディエントベクトル」(gradient vector) と呼ばれます。

SVG は、次の 2 種類のグラディエントを定義するための要素を持っています。

- 線形グラディエント (linear gradient)
- 放射状グラディエント (radial gradient)

線形グラディエントというのは、グラディエントベクトルを並行に移動させることによってできるグラディエントのことで、放射状グラディエントというのは、グラディエントベクトルを、その開始点を中心にして回転させることによってできるグラディエントのことです。

3.1.2 グラディエントの適用

SVG では、線や領域に対してどのように色を与えるかという方法のことを、「ペイントサーバー」(paint server) と呼びます。

線や領域に対してペイントサーバーを適用したいときは、`stroke` 属性または `fill` 属性に、ペイントサーバーを名前で参照する、

`url(#`名前`)`

という形の記述を設定します。たとえば、

```
fill="url(#paintserver)"
```

という属性指定を書くことによって、`paintserver` という名前を持つペイントサーバーを領域に適用することができます。

グラディエントもペイントサーバーの一種ですので、グラディエントを名前で参照する記述を `stroke` または `fill` に設定することによって、そのグラディエントを線または領域に適用することができます。

3.1.3 線形グラディエント

SVG では、線形グラディエントは、`linearGradient` という要素型の要素によって定義されます。`linearGradient` 要素を参照するための名前は、`id` 属性を使うことによって付けることができます。

グラディエントベクトルの上の位置に対する色の割り当てのことを、「グラディエントストップ」(gradient stop) と呼びます。

線形グラディエントのグラディエントストップは、`linearGradient` 要素の子供として `stop` という要素型の要素を書くことによって定義することができます。

グラディエントストップの位置と色は、`stop` 要素が持っている次の属性に設定します。

`offset`	パーセントで指定された位置。
`stop-color`	位置に設定する色。

たとえば、

```
<stop offset="30%" stop-color="blue"/>
```

という要素は、グラディエントベクトルの 30%という位置に青色を割り当てるという意味になります。

SVG 文書の例 `linear.svg`

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="320pt" height="224pt" viewBox="0 0 100 70"
    xmlns="http://www.w3.org/2000/svg">
  <linearGradient id="greenyellow">
    <stop offset="0%" stop-color="green"/>
```

```
    <stop offset="100%" stop-color="yellow"/>
  </linearGradient>
  <rect x="5" y="5" width="90" height="60"
     fill="url(#greenyellow)"/>
</svg>
```

3.1.4 線形グラディエントのグラディエントベクトル

グラディエントベクトルは、x1、y1、x2、y2、という 4 個の属性に設定された属性値によって決定されます。x1 と y1 が開始点の座標で、x2 と y2 が終了点の座標です。これらの座標としては、グラディエントが適用される線または領域に対するパーセントを設定することができます。

グラディエントベクトルのデフォルトは、x1 が 0%、y1 が 0%、x2 が 100%、y2 が 0%、つまり、左から右へ向かう水平なベクトルです。

SVG 文書の例 vector.svg

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="320pt" height="224pt" viewBox="0 0 100 70"
    xmlns="http://www.w3.org/2000/svg">
  <linearGradient id="blueaqua"
     x1="0%" y1="100%" x2="100%" y2="0%">
    <stop offset="0%" stop-color="blue"/>
    <stop offset="100%" stop-color="aqua"/>
  </linearGradient>
  <rect x="5" y="5" width="90" height="60"
     fill="url(#blueaqua)"/>
</svg>
```

3.1.5 放射状グラディエント

SVG では、放射状グラディエントは、radialGradient という要素型の要素によって定義されます。線形グラディエントの場合と同じように、id 属性に設定された名前によって、その要素を参照することができます。

グラディエントストップを定義する方法も、線形グラディエントの場合と同じです。放射状グラディエントを定義する要素の子供として stop 要素を書けばいいわけです。

SVG 文書の例 radial.svg

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="320pt" height="224pt" viewBox="0 0 100 70"
    xmlns="http://www.w3.org/2000/svg">
  <radialGradient id="lightdarkgreen">
    <stop offset="0%" stop-color="lightgreen"/>
    <stop offset="100%" stop-color="darkgreen"/>
  </radialGradient>
  <rect x="5" y="5" width="90" height="60"
     fill="url(#lightdarkgreen)"/>
</svg>
```

3.1.6 放射状グラディエントの中心と半径

放射状グラディエントの中心と半径は、次の属性に設定された属性値によって決定されます。

cx 中心の x 座標。デフォルトは 50%。

cy 中心の y 座標。デフォルトは 50%。

r 半径。デフォルトは 50%。

SVG 文書の例 center.svg

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="320pt" height="224pt" viewBox="0 0 100 70"
```

```
    xmlns="http://www.w3.org/2000/svg">
  <radialGradient id="whitenavy"
      cx="50%" cy="100%" r="100%">
    <stop offset="0%" stop-color="white"/>
    <stop offset="100%" stop-color="navy"/>
  </radialGradient>
  <rect x="5" y="5" width="90" height="60"
      fill="url(#whitenavy)"/>
</svg>
```

3.1.7 グラディエントの継承

グラディエントの要素（つまり linearGradient と radialGradient）は、xlink:href 属性を使って別のグラディエントの要素をその名前で参照することによって、それが持っている属性とグラディエントストップをそこから継承することができます。

SVG 文書の例 inherite.svg

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="320pt" height="224pt" viewBox="0 0 100 70"
    xmlns="http://www.w3.org/2000/svg"
    xmlns:xlink="http://www.w3.org/1999/xlink">
  <linearGradient id="yellowredmagenta"
      x1="0%" y1="0%" x2="0%" y2="100%">
    <stop offset="0%" stop-color="yellow"/>
    <stop offset="50%" stop-color="red"/>
    <stop offset="100%" stop-color="magenta"/>
  </linearGradient>
  <linearGradient id="west"
      xlink:href="#yellowredmagenta" x2="100%" y2="0%"/>
  <linearGradient id="north"
      xlink:href="#yellowredmagenta"/>
  <linearGradient id="northwest"
      xlink:href="#yellowredmagenta" x2="100%"/>
  <linearGradient id="southwest"
      xlink:href="#yellowredmagenta"
      y1="100%" x2="100%" y2="0%"/>
  <symbol id="rect">
    <rect x="5" y="5" width="42.5" height="27.5"/>
  </symbol>
  <use xlink:href="#rect" fill="url(#west)"/>
  <use xlink:href="#rect" x="47.5" fill="url(#north)"/>
  <use xlink:href="#rect" y="32.5" fill="url(#northwest)"/>
  <use xlink:href="#rect" x="47.5" y="32.5"
      fill="url(#southwest)"/>
</svg>
```

3.1.8 不透明度のグラディエント

色だけではなくて、不透明度を連続的に変化させること、つまり不透明度のグラディエントを作ることも可能です。

stop 要素は、stop-opacity という属性を持っています。この属性に不透明度を設定すると、その値が、offset 属性で指定された位置の不透明度になります。

SVG 文書の例 opagra.svg

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="320pt" height="224pt" viewBox="0 0 100 70"
    xmlns="http://www.w3.org/2000/svg">
  <linearGradient id="opacitygradient">
    <stop offset="0%" stop-color="navy" stop-opacity="1"/>
    <stop offset="100%" stop-color="navy" stop-opacity="0"/>
  </linearGradient>
```

```
  <rect x="10" y="20" width="80" height="30" fill="aqua"/>
  <rect x="5" y="5" width="90" height="60"
      fill="url(#opacitygradient)"/>
</svg>
```

3.2 パターン

3.2.1 パターンとは何か

SVG では、一定のグラフィックスを縦方向と横方向に並べることによって領域を埋め尽くすということが可能です。そのとき、領域の中に並べられるグラフィックスは、「パターン」(pattern) と呼ばれます。

3.2.2 パターンの適用

パターンは、グラディエントと同じようにペイントサーバーの一種です。ですから、グラディエントの場合と同じように、stroke 属性または fill 属性に、パターンを名前で参照する、

url(# 名前)

という形の記述を設定することによって、その名前を持つパターンを線または領域に適用することができます。

3.2.3 パターンの定義

SVG では、パターンは、pattern という要素型の要素によって定義されます。pattern 要素の子供として、何らかの描画を実行する要素を書くと、それらの要素によって描画されるグラフィックスが、パターンとして定義されることになります。

パターンを定義する場合は、pattern 要素の次の属性に属性値を設定する必要があります。

width	横方向のパターンの間隔。
height	縦方向のパターンの間隔。

width と height は、デフォルトの状態では、パターンを適用する線または領域の大きさに対するパーセンテージで指定します。

SVG 文書の例 pattern.svg

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="320pt" height="224pt" viewBox="0 0 100 70"
    xmlns="http://www.w3.org/2000/svg">
  <pattern id="circlepattern" width="20%" height="25%">
    <circle cx="4" cy="4" r="3" fill="skyblue"/>
  </pattern>
  <rect x="5" y="5" width="55" height="30"
      fill="url(#circlepattern)"/>
  <rect x="65" y="5" width="30" height="60"
      fill="url(#circlepattern)"/>
</svg>
```

3.2.4 パターンの敷き詰め

パターンを並べる間隔を指定する方法は、デフォルトでは、それを適用する線または領域の大きさに対するパーセンテージで指定することになっているわけですが、そうではなくて、パターンの大きさを設定しておいて、そのパターンを隙間なく敷き詰めるという方法で指定することも可能です。

大きさが設定されたパターンを線または領域に敷き詰めたいときは、pattern 要素が持っている patternUnits という属性に userSpaceOnUse という属性値を設定して、width と height のそれぞれにパターンの大きさを設定します。

SVG 文書の例 userspc.svg

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
```

```
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="320pt" height="224pt" viewBox="0 0 100 70"
    xmlns="http://www.w3.org/2000/svg">
  <pattern id="circlepattern" patternUnits="userSpaceOnUse"
      width="8" height="8">
    <circle cx="4" cy="4" r="3" fill="turquoise"/>
  </pattern>
  <rect x="5" y="5" width="55" height="30"
      fill="url(#circlepattern)"/>
  <rect x="65" y="5" width="30" height="60"
      fill="url(#circlepattern)"/>
</svg>
```

3.2.5 パターンの入れ子

パターンは、入れ子にすることも可能です。つまり、パターンが適用された線または領域を持つグラフィックスを、さらにパターンとして線または領域に適用する、ということが可能だということです。

SVG 文書の例 nested.svg

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="320pt" height="224pt" viewBox="0 0 100 70"
    xmlns="http://www.w3.org/2000/svg">
  <pattern id="stripe" patternUnits="userSpaceOnUse"
      width="2" height="20">
    <line x1="1" y1="0" x2="1" y2="20"
        stroke-width="0.3" stroke="mediumblue"/>
  </pattern>
  <pattern id="circlepattern" width="33.3%" height="50%">
    <circle cx="15" cy="15" r="14" fill="url(#stripe)"/>
  </pattern>
  <rect x="5" y="5" width="90" height="60"
      fill="url(#circlepattern)"/>
</svg>
```

3.3 クリッピング

3.3.1 クリッピングとは何か

SVG は、「クリッピング」(clipping) と呼ばれる機能を持っています。

クリッピングというのは、何らかの形状を持つ枠を作っておいて、その枠の中だけで描画を実行するということです。クリッピングのために使われる形状は、「クリッピングパス」(clipping path) と呼ばれます。

3.3.2 クリッピングパスの定義

クリッピングパスは、clipPath という要素型の要素を書くことによって定義することができます。

clipPath 要素の子供として、グラフィックスを描画する要素を書くと、そのグラフィックスがクリッピングパスとして定義されます。clipPath 要素が持っている id という属性に名前を設定すると、それがクリッピングパスの名前になります。たとえば、

```
<clipPath id="circleclip">
  <circle cx="50" cy="35" r="20"/>
</clipPath>
```

という clipPath 要素を書くことによって、circleclip という名前を持つ、円の形のクリッピングパスを定義することができます。

3.3.3 クリッピングパスの適用

グラフィックスを描画する要素、または g 要素が持っている clip-path という属性に、

url(#名前)

という形の記述を設定すると、その中の名前で指定されたクリッピングパスがグラフィックスに適用されます。

SVG 文書の例 clipping.svg

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="320pt" height="224pt" viewBox="0 0 100 70"
    xmlns="http://www.w3.org/2000/svg">
  <clipPath id="ellipseclip">
    <ellipse cx="50" cy="35" rx="45" ry="30"/>
  </clipPath>
  <rect x="0" y="0" width="100" height="70" fill="navy"/>
  <g clip-path="url(#ellipseclip)">
    <circle cx="0" cy="100" r="80" fill="lavender"/>
    <circle cx="20" cy="30" r="18" fill="lightsalmon"/>
    <circle cx="50" cy="5" r="16" fill="plum"/>
    <circle cx="120" cy="0" r="70" fill="lightyellow"/>
    <circle cx="80" cy="60" r="20" fill="crimson"/>
  </g>
</svg>
```

3.4 画像

3.4.1 画像を描画する要素

SVG は、PNG や JPEG などのラスター画像を長方形の領域に描画することができるという機能を持っています。

画像を描画したいときは、image という要素型の要素を書きます。この要素は次のような属性を持っています。

xlink:href	画像のファイルを識別する URI。
x	領域の左上の点の x 座標。
y	領域の左上の点の y 座標。
width	領域の横の長さ。
height	領域の縦の長さ。

SVG 文書の例 image.svg

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="320pt" height="224pt" viewBox="0 0 100 70"
    xmlns="http://www.w3.org/2000/svg"
    xmlns:xlink="http://www.w3.org/1999/xlink">
  <image xlink:href="sample.jpg" x="5" y="5"
      width="90" height="60"/>
</svg>
```

3.4.2 SVG 文書の参照

image 要素は、ラスター画像だけではなくて、外部に存在している SVG 文書を参照して、それが記述しているグラフィックスを描画する、ということも可能です。

SVG 文書の例 inner.svg

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="320pt" height="224pt" viewBox="0 0 100 70"
    xmlns="http://www.w3.org/2000/svg">
  <rect x="5" y="5" width="70" height="50" fill="orangered"/>
  <ellipse cx="60" cy="40" rx="35" ry="25" fill="royalblue"/>
```

```
</svg>
```

SVG 文書の例　refsvg.svg

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="320pt" height="224pt" viewBox="0 0 100 70"
    xmlns="http://www.w3.org/2000/svg"
    xmlns:xlink="http://www.w3.org/1999/xlink">
  <image xlink:href="inner.svg" x="5" y="5"
      width="40" height="30"/>
  <image xlink:href="inner.svg" x="25" y="20"
      width="70" height="50"/>
</svg>
```

第 4 章　フィルター

4.1　フィルターの基礎

4.1.1　フィルターとは何か

SVG では、グラフィックスに対する演算のことを「フィルター」(filter) と呼びます。

SVG は、フィルターを定義して、それをグラフィックスに適用することができる、という機能を持っています。この機能を使うことによって、グラフィックスの輪郭をぼかしたり、照明効果を加えたりすることができます。

4.1.2　フィルターの定義

フィルターを使うためには、まず第一に、それを定義する必要があります。フィルターを定義したいときは、filter という要素型の要素を書きます。その要素の id 属性に設定された名前が、定義されるフィルターの名前になります。たとえば、

```
<filter id="donothing"/>
```

という filter 要素を書くことによって、 donothing という名前を持つ、何もしないフィルターを定義することができます。

4.1.3　原始フィルター

何らかの動作をするフィルターを定義するためには、それがどのような演算であるかということを、 filter 要素の子供として記述する必要があります。 filter 要素の子供としては、「原始フィルター」(filter primitive) と呼ばれるものを指定する要素を書きます。原始フィルターというのは、SVG によって定義されているさまざまなフィルターのことです。

たとえば、SVG では、 feGaussianBlur という要素型の要素によって指定される原始フィルターが定義されています。この原始フィルターは、「ガウシアンブラー」(Gaussian blur) と呼ばれる、輪郭をぼかす演算を実行します。

feGaussianBlur を使う場合には、 stdDeviation という属性に、ぼかしの程度をあらわす数値を設定する必要があります。

たとえば、

```
<filter id="blur">
  <feGaussianBlur stdDeviation="2"/>
</filter>
```

という filter 要素を書くことによって、ガウシアンブラーを実行する、 blur という名前のフィルターを定義することができます。

4.1.4　フィルターの適用

フィルターをグラフィックスに適用したいときは、そのグラフィックスを描画する要素、または g 要素が持っている filter という属性に対して、

```
filter="url(#名前)"
```

という形式で、フィルターの名前を設定します。

SVG 文書の例　filter.svg

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="320pt" height="224pt" viewBox="0 0 100 70"
    xmlns="http://www.w3.org/2000/svg">
  <filter id="blur">
    <feGaussianBlur stdDeviation="2"/>
  </filter>
  <ellipse cx="50" cy="35" rx="40" ry="25" fill="limegreen"
      filter="url(#blur)"/>
</svg>
```

4.1.5　移動の原始フィルター

feOffset という原始フィルターは、グラフィックスの移動という演算を実行します。

feOffset 要素は、dx と dy という属性を持っています。dx には x 軸方向の移動量、dy には y 軸方向の移動量を設定します。

なお、feOffset を使う場合は、filter 要素の開始タグの中に、

```
filterUnits="userSpaceOnUse"
```

という属性指定を書いておく必要があります。その理由は、filterUnits 属性がデフォルトのままだと、フィルターを適用した結果を、フィルターを適用する対象となるグラフィックスを囲む長方形の外側に描画することができないからです。

SVG 文書の例　offset.svg

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="320pt" height="224pt" viewBox="0 0 100 70"
    xmlns="http://www.w3.org/2000/svg">
  <filter id="offset" filterUnits="userSpaceOnUse">
    <feOffset dx="20" dy="10"/>
  </filter>
  <ellipse cx="40" cy="30" rx="35" ry="25" fill="lime"/>
  <ellipse cx="40" cy="30" rx="35" ry="25" fill="green"
      filter="url(#offset)"/>
</svg>
```

4.1.6　原始フィルターの接続

フィルターは、複数個の原始フィルターを組み合わせることによって定義することも可能です。

filter 要素の子供として、複数個の原始フィルターの要素を書くと、それらの原始フィルターを組み合わせたフィルターが定義されます。フィルターが適用されたグラフィックスは、原則として、原始フィルターの要素が書かれている順序のとおりに、上から下へ、それぞれの原始フィルターの中を通過していきます。つまり、原始フィルターの入力は、要素が直前に書かれている原始フィルターの出力に接続される、ということです。

次の SVG 文書の中で定義されている bluroffset というフィルターは、まずガウシアンブラーを実行したのち、その出力に対して移動を実行します。

SVG 文書の例　connect.svg

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="320pt" height="224pt" viewBox="0 0 100 70"
    xmlns="http://www.w3.org/2000/svg">
  <filter id="bluroffset" filterUnits="userSpaceOnUse">
    <feGaussianBlur stdDeviation="2"/>
    <feOffset dx="20" dy="10"/>
```

```
  </filter>
  <ellipse cx="40" cy="30" rx="35" ry="25" fill="crimson"/>
  <ellipse cx="40" cy="30" rx="35" ry="25" fill="orange"
      filter="url(#bluroffset)"/>
</svg>
```

原始フィルターの要素は、 in という属性を持っています。この属性に対して属性値を設定することによって、原始フィルターの入力を明示的に何かに接続する、ということができます。

たとえば、原始フィルターの in 属性に対して SourceGraphic という属性値を設定すると、その原始フィルターの入力は、フィルターが適用されたグラフィックスに明示的に接続されます。

4.1.7 アルファチャンネル

グラフィックスは、形状、色、不透明度から構成されていると考えることができます。グラフィックスから色を取り除いて、形状と不透明度だけを残したものは、グラフィックスの「アルファチャンネル」(alpha channel) と呼ばれます。

フィルターをグラフィックスに適用すると、デフォルトでは、形状と色と不透明度を持つグラフィックスがその入力になります。しかし、フィルターの目的によっては、アルファチャンネルが必要になる場合もあります。

原始フィルターの in 属性に対して SourceAlpha という属性値を設定すると、その原始フィルターの入力は、フィルターが適用されたグラフィックスのアルファチャンネルに接続されます。

SVG 文書の例 alpha.svg

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="320pt" height="224pt" viewBox="0 0 100 70"
    xmlns="http://www.w3.org/2000/svg">
  <linearGradient id="opacitygradient">
    <stop offset="0%" stop-color="lime" stop-opacity="0"/>
    <stop offset="100%" stop-color="lime" stop-opacity="1"/>
  </linearGradient>
  <filter id="offset" filterUnits="userSpaceOnUse">
    <feOffset dx="20" dy="10" in="SourceAlpha"/>
  </filter>
  <g fill="url(#opacitygradient)">
    <ellipse cx="40" cy="30" rx="35" ry="25"/>
    <ellipse cx="40" cy="30" rx="35" ry="25"
        filter="url(#offset)"/>
  </g>
</svg>
```

4.2 併合

4.2.1 名前による原始フィルターの接続

原始フィルターの出力には、名前を与えることができます。原始フィルターの出力に名前を与えておくと、複数の原始フィルターを組み合わせることによってフィルターを定義する場合に、その名前を使うことによって、それらの原始フィルターの出力と入力とを明示的に接続することができます。

原始フィルターの要素は、 result という属性を持っています。この属性に名前を設定すると、その名前は、その原始フィルターの出力に対して与えられます。

原始フィルターの要素が持っている in 属性に対して、原始フィルターの出力に与えられている名前を設定すると、それらの原始フィルターの出力と入力は、その名前によって明示的に接続されることになります。

4.2.2 併合の原始フィルター

複数の入力をひとつにまとめて出力することを、入力を「併合する」(merge) と言います。

feMerge という原始フィルターは、複数の入力を併合した結果を出力します。

feMerge 要素は、任意の個数の feMergeNode 要素を子供として持つことができます。それぞ

れの feMergeNode 要素が持っている in 属性に対して、併合したいものの名前を設定すると、feMerge は、それらを併合した結果を出力します。

SVG 文書の例　merge.svg

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="320pt" height="224pt" viewBox="0 0 100 70"
    xmlns="http://www.w3.org/2000/svg">
  <filter id="merge" filterUnits="userSpaceOnUse">
    <feOffset dx="20" dy="10" in="SourceGraphic" result="a"/>
    <feOffset dx="-20" dy="10" in="SourceGraphic" result="b"/>
    <feOffset dx="0" dy="-10" in="SourceGraphic" result="c"/>
    <feMerge>
      <feMergeNode in="a"/>
      <feMergeNode in="b"/>
      <feMergeNode in="c"/>
    </feMerge>
  </filter>
  <ellipse cx="50" cy="35" rx="25" ry="20" fill="greenyellow"
      filter="url(#merge)"/>
</svg>
```

4.2.3　影を付けるフィルター

グラフィックスに適用すると、そのグラフィックスの影を右下に付加するフィルターを作ってみましょう。

影は、適用したグラフィックスのアルファチャンネルにガウシアンブラーを実行して、さらに右下へ移動させることによって作ることができます。そして、もとのグラフィックスと影とを併合すれば、影の付いたグラフィックスが得られます。

SVG 文書の例　shadow.svg

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="320pt" height="224pt" viewBox="0 0 100 70"
    xmlns="http://www.w3.org/2000/svg">
  <filter id="shadow" filterUnits="userSpaceOnUse">
    <feGaussianBlur stdDeviation="0.7"
        in="SourceAlpha" result="blur"/>
    <feOffset dx="1.5" dy="1" in="blur" result="offset"/>
    <feMerge>
      <feMergeNode in="offset"/>
      <feMergeNode in="SourceGraphic"/>
    </feMerge>
  </filter>
  <text x="10" y="42" font-size="26" font-family="serif"
      fill="springgreen" filter="url(#shadow)">
    shadow
  </text>
</svg>
```

4.3　合成

4.3.1　合成の原始フィルター

二つのグラフィックスに対する演算を実行して、その結果を出力することを、入力を「合成する」(composite) と言います。

feComposite という原始フィルターは、二つの入力を合成した結果を出力します。

feComposite に与える二つの入力は、ひとつは in という属性に設定して、もうひとつは in2 という属性に設定します。

4.3.2　合成演算

feComposite が実行することのできる演算は、「合成演算」(compositing operation) と呼ばれます。どの合成演算を実行したいのかということは、その名前を operator という属性に設定することによって指定することができます。合成演算は次の 6 種類です。

over	in2 の上に in を重ねたものを出力する。
in	in のうちで in2 と重なっている部分だけを出力する。
out	in のうちで in2 と重なっていない部分だけを出力する。
atop	in のうちで in2 と重なっている部分を in2 の上に重ねたものを出力する。
xor	in と in2 から、それらが重なっている部分を取り除いたものを出力する。
arithmetic	ピクセルごとに演算を実行した結果を出力する。

arithmetic を実行する場合は、k1、k2、k3、k4 という属性のそれぞれに数値を設定する必要があります。そうすると、ピクセルごとに、

$$(\mathtt{k1} \times \mathtt{in} \times \mathtt{in2}) + (\mathtt{k2} \times \mathtt{in}) + (\mathtt{k3} \times \mathtt{in2}) + \mathtt{k4}$$

という計算が実行されて、その結果が出力のピクセルになります。

SVG 文書の例　compo.svg

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="320pt" height="224pt" viewBox="0 0 100 70"
    xmlns="http://www.w3.org/2000/svg"
    xmlns:xlink="http://www.w3.org/1999/xlink">
  <filter id="over" filterUnits="userSpaceOnUse">
    <feOffset dx="5" dy="3" in="SourceAlpha" result="offset"/>
    <feComposite operator="over"
        in="SourceGraphic" in2="offset"/>
  </filter>
  <filter id="in" filterUnits="userSpaceOnUse">
    <feOffset dx="5" dy="3" in="SourceAlpha" result="offset"/>
    <feComposite operator="in"
        in="SourceGraphic" in2="offset"/>
  </filter>
  <filter id="out" filterUnits="userSpaceOnUse">
    <feOffset dx="5" dy="3" in="SourceAlpha" result="offset"/>
    <feComposite operator="out"
        in="SourceGraphic" in2="offset"/>
  </filter>
  <filter id="atop" filterUnits="userSpaceOnUse">
    <feOffset dx="5" dy="3" in="SourceAlpha" result="offset"/>
    <feComposite operator="atop"
        in="SourceGraphic" in2="offset"/>
  </filter>
  <filter id="xor" filterUnits="userSpaceOnUse">
    <feOffset dx="5" dy="3" in="SourceAlpha" result="offset"/>
    <feComposite operator="xor"
        in="SourceGraphic" in2="offset"/>
  </filter>
  <filter id="arithmetic" filterUnits="userSpaceOnUse">
    <feOffset dx="5" dy="3" in="SourceAlpha" result="offset"/>
    <feComposite operator="arithmetic"
        k1="0" k2="0.8" k3="0.2" k4="0"
        in="SourceGraphic" in2="offset"/>
  </filter>
  <symbol id="rect">
    <rect x="20" y="7" width="20" height="14" fill="coral"/>
  </symbol>
  <g font-size="4" font-family="serif" fill="forestgreen">
    <use xlink:href="#rect" filter="url(#over)"/>
    <text x="10" y="14">over</text>
    <use xlink:href="#rect" y="20" filter="url(#in)"/>
    <text x="10" y="34">in</text>
    <use xlink:href="#rect" y="40" filter="url(#out)"/>
```

```
    <text x="10" y="54">out</text>
    <use xlink:href="#rect" x="45" filter="url(#atop)"/>
    <text x="55" y="14">atop</text>
    <use xlink:href="#rect" x="45" y="20" filter="url(#xor)"/>
    <text x="55" y="34">xor</text>
    <use xlink:href="#rect" x="45" y="40"
        filter="url(#arithmetic)"/>
    <text x="55" y="54">arithmetic</text>
  </g>
</svg>
```

4.4　ライティング

4.4.1　ライティングの基礎

入力のグラフィックスに光を当てて、その光が反射した結果を求める演算は、「ライティング」(lighting) と呼ばれます。グラフィックスにライティングを適用することによって、そのグラフィックスを立体的に見せることができます。

次の二つの原始フィルターは、ライティングを実行します。

`feDiffuseLighting`　拡散反射のライティング。

`feSpecularLighting`　鏡面反射のライティング。

ライティングについて考える場合には、平面上の位置を指定するための x 軸と y 軸だけではなくて、z 軸と呼ばれる第 3 の軸が必要となります。z 軸は、xy 平面と垂直に交わっていて、グラフィックスを見る人間の目が存在する方向を向いています。

グラフィックスに対してライティングを適用する場合、そのグラフィックスは、z 軸方向の凹凸を持っている必要があります。ライティングにおいては、グラフィックスの z 軸方向の凹凸は、不透明度によってあらわされているとみなされます。

4.4.2　光源

ライティングを実行するためには、そのための光源を作る必要があります。光源は、ライティングを実行する原始フィルターの要素の子供として、光源を作る要素を書くことによって作られます。

`feDistantLight` という要素型の要素は、無限遠光源、つまり無限に遠い位置にある光源を作ります。光源の方向は、次の属性に数値を設定することによって指定します。

`azimuth`　xy 平面での光源の方向をあらわす角度。右が 0 度で、時計回り。

`elevation`　光源の z 軸方向の高さをあらわす角度。

4.4.3　拡散反射

物体に当たった光の反射のうちで、様々な方向に散乱するものは、「拡散反射」(diffuse reflection) と呼ばれます。

拡散反射のライティングは、`feDiffuseLighting` という要素型の要素によって実行されます。この要素は、次のような属性を持っています。

`surfaceScale`　不透明度が 1 のときの z 軸方向の高さ。

`diffuseConstant`　拡散反射定数。

`lighting-color`　光源の色。

4.4.4　ライティングの適用

グラフィックスに対してライティングを適用したいときは、次のような手順を実行するフィルターを定義する必要があります。

(1) ガウシアンブラーを使って不透明度の差を作る（つまり、z 軸方向の凹凸を作る）。

(2) その結果に対してライティングを実行する。

(3) `feComposite` の `in` を使って、ライティングの結果を、もとのグラフィックスの領域の中だけにする。

(4) feComposite の arithmetic を使って、ライティングの結果ともとのグラフィックスとを合成する。

SVG 文書の例 diffuse.svg

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="320pt" height="224pt" viewBox="0 0 100 70"
    xmlns="http://www.w3.org/2000/svg">
  <filter id="difflight" filterUnits="userSpaceOnUse">
    <feGaussianBlur stdDeviation="4"
        in="SourceAlpha" result="blur"/>
    <feDiffuseLighting surfaceScale="20" diffuseConstant="0.4"
        lighting-color="white" in="blur" result="diff">
      <feDistantLight azimuth="-150" elevation="30"/>
    </feDiffuseLighting>
    <feComposite operator="in"
        in="diff" in2="SourceAlpha" result="compoin"/>
    <feComposite operator="arithmetic"
        k1="0" k2="1" k3="1" k4="0"
        in="SourceGraphic" in2="compoin"/>
  </filter>
  <ellipse cx="50" cy="35" rx="35" ry="20"
      stroke-width="20" stroke="mediumblue" fill="none"
      filter="url(#difflight)"/>
</svg>
```

4.4.5 鏡面反射

物体に当たった光の反射のうちで、特定の方向へ向かうものは、「鏡面反射」(specular reflection) と呼ばれます。

鏡面反射のライティングは、feSpecularLighting という要素型の要素によって実行されます。この要素は、次のような属性を持っています。

surfaceScale	不透明度が 1 のときの z 軸方向の高さ。
specularConstant	鏡面反射定数。
specularExponent	鏡面指数。1 から 128 まで。この数値が大きいほど鏡面らしくなる。
lighting-color	光源の色。

SVG 文書の例 specular.svg

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="320pt" height="224pt" viewBox="0 0 100 70"
    xmlns="http://www.w3.org/2000/svg">
  <filter id="speclight" filterUnits="userSpaceOnUse">
    <feGaussianBlur stdDeviation="4"
        in="SourceAlpha" result="blur"/>
    <feSpecularLighting surfaceScale="20"
        specularConstant="0.7" specularExponent="8"
        lighting-color="white" in="blur" result="spec">
      <feDistantLight azimuth="-150" elevation="30"/>
    </feSpecularLighting>
    <feComposite operator="in"
        in="spec" in2="SourceAlpha" result="compoin"/>
    <feComposite operator="arithmetic"
        k1="0" k2="1" k3="1" k4="0"
        in="SourceGraphic" in2="compoin"/>
  </filter>
  <ellipse cx="50" cy="35" rx="35" ry="20"
      stroke-width="20" stroke="mediumblue" fill="none"
      filter="url(#speclight)"/>
</svg>
```

第 5 章　アニメーション

5.1　アニメーションの基礎

5.1.1　アニメーションの基礎の基礎

時間の経過とともに変化するグラフィックスは、「アニメーション」(animation) と呼ばれます。

SVG は、アニメーションを作る機能を持っています。SVG でアニメーションを作る方法としては、そのための要素を書く方法と、「スクリプト」と呼ばれるものを書く方法とがあります。

この章では、SVG でアニメーションを作るための方法のうちで、そのための要素を利用するものについて説明したいと思います。

5.1.2　アニメーション要素

SVG で定義されている、アニメーションを作るための要素は、「アニメーション要素」(animation element) と呼ばれます。

アニメーション要素には、次の五つのものがあります。

`animate`	要素の属性を変化させる。
`animateColor`	グラフィックスの色を変化させる。
`animateTransform`	座標系を変化させる。
`animateMotion`	パスに沿って座標系を移動させる。
`set`	指定された時間だけ属性に値を設定する。

5.1.3　アニメーション要素の属性

アニメーション要素は、アニメーションを作るための属性として、次のようなものを持っています。

`xlink:href`	アニメーションが変化させる対象となる要素。この属性に対して属性値が設定されていない場合は、アニメーション要素の親になっている要素が変化の対象となる。
`attributeName`	変化の対象となる属性の名前。`animateMotion` 要素を使う場合は不要。
`from`	開始値 (starting value)、すなわち、アニメーションが開始される時点での属性値。
`to`	終了値 (ending value)、すなわち、アニメーションが終了する時点での属性値。
`dur`	持続時間 (duration)、すなわち、アニメーションが開始されてから終了するまでの時間。数値の右側に時間の単位を書いたものを設定する。時間の単位は、時間 (`h`)、分 (`min`)、秒 (`s`)、ミリ秒 (`ms`) のいずれか。
`repeatCount`	繰り返しの回数。無限に繰り返したい場合は `indefinite` を設定する。
`fill`	アニメーションが終了したのち、属性値をどうするか。アニメーションによって設定された属性値のままで凍結させたい場合は `freeze` を設定して、それを破棄して本来の属性値に戻したい場合は `remove` を設定する。

5.2　属性のアニメーション

5.2.1　属性値を変化させる要素

`animate` というアニメーション要素は、要素の属性を変化させることによってアニメーションを作ります。

次の SVG 文書の中の `animate` 要素は、円を描画する `circle` 要素が持っている `cx` という属性を変化させることによって、円を左から右へ移動させるアニメーションを作ります。

SVG 文書の例　`attricx.svg`

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="320pt" height="224pt" viewBox="0 0 100 70"
    xmlns="http://www.w3.org/2000/svg">
```

```
  <circle cy="35" r="5" fill="lightseagreen">
    <animate attributeName="cx" from="15" to="85"
        dur="10s" repeatCount="3" fill="freeze"/>
  </circle>
</svg>
```

同じように、次の SVG 文書の中の animate 要素は、長方形を描画する rect 要素が持っている opacity という属性を変化させることによって、不透明度のアニメーションを作ります。

SVG 文書の例 attrio.svg

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="320pt" height="224pt" viewBox="0 0 100 70"
    xmlns="http://www.w3.org/2000/svg">
  <ellipse cx="50" cy="35" rx="40" ry="25" fill="dodgerblue"/>
  <rect x="5" y="5" width="90" height="60" fill="mediumblue">
    <animate attributeName="opacity" from="1" to="0"
        dur="10s" repeatCount="indefinite"/>
  </rect>
</svg>
```

5.2.2 複数の属性のアニメーション

ひとつの要素の子供としてはひとつのアニメーション要素しか書くことができない、という制約は存在しません。ですから、ひとつの要素の子供として複数の animate 要素を書くことによって、ひとつの要素の複数の属性を同時に変化させるアニメーションを作るということも可能です。

次の SVG 文書は、 rect 要素の子供として書かれている二つの animate 要素によって、長方形の横の長さと縦の長さが同時に変化します。

SVG 文書の例 attriwh.svg

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="320pt" height="224pt" viewBox="0 0 100 70"
    xmlns="http://www.w3.org/2000/svg">
  <rect x="10" y="10" fill="darkcyan">
    <animate attributeName="width" from="10" to="80"
        dur="10s" repeatCount="indefinite"/>
    <animate attributeName="height" from="50" to="10"
        dur="10s" repeatCount="indefinite"/>
  </rect>
</svg>
```

5.2.3 色の変化

fill や stroke のような、色が設定される属性を変化させるアニメーションを作りたいときは、animate 要素ではなくて animateColor というアニメーション要素を使います。この要素の使い方は、 animate 要素の使い方と同じです。

次の SVG 文書の中の animateColor 要素は、楕円を描画する ellipse 要素が持っている fill という属性を変化させることによって、楕円の色を空色から黄緑へ変化させるアニメーションを作ります。

SVG 文書の例 attrif.svg

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="320pt" height="224pt" viewBox="0 0 100 70"
    xmlns="http://www.w3.org/2000/svg">
  <ellipse cx="50" cy="35" rx="40" ry="25">
    <animateColor attributeName="fill"
        from="skyblue" to="yellowgreen"
        dur="5s" repeatCount="indefinite"/>
  </ellipse>
```

```
</svg>
```

5.3　座標系の変換のアニメーション

5.3.1　座標系を変化させる要素

animateTransform というアニメーション要素は、座標系を変化させることによってアニメーションを作ります。

animateTransform 要素が持っている attributeName 属性には、transform という属性の名前を設定します。

animateTransform 要素は、type という属性を持っています。この属性には、座標系の変化のタイプをあらわす名前を設定します。設定することのできる変化のタイプは、次の五つです。

translate	移動
scale	拡大
rotate	回転
skewX	x 軸に沿った剪断
skewY	y 軸に沿った剪断

アニメーションの開始値と終了値は、ほかのアニメーション要素と同じように、from 属性と to 属性に設定します。

5.3.2　移動のアニメーション

animateTransform 要素を使って移動のアニメーションを作る場合、from 属性と to 属性には、それぞれ、

x, y

という形式の属性値を設定します。x は x 軸方向への移動距離で、y は y 軸方向への移動距離です。

SVG 文書の例　animtra.svg

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="320pt" height="224pt" viewBox="0 0 100 70"
    xmlns="http://www.w3.org/2000/svg">
  <g fill="chocolate">
    <circle cx="0" cy="0" r="5"/>
    <circle cx="5" cy="0" r="5"/>
    <animateTransform attributeName="transform"
        type="translate" from="0, 0" to="100, 70"
        dur="5s" repeatCount="indefinite"/>
  </g>
</svg>
```

5.3.3　拡大のアニメーション

animateTransform 要素を使って拡大のアニメーションを作る場合、from 属性と to 属性には、それぞれ、

x, y

という形式の属性値を設定します。x は x 軸方向の拡大率で、y は y 軸方向の拡大率です（コンマと y を省略して x だけを書いた場合、y 軸方向の拡大率は x 軸方向と同じだと解釈されます）。

SVG 文書の例　animsca.svg

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="320pt" height="224pt" viewBox="0 0 100 70"
    xmlns="http://www.w3.org/2000/svg">
  <g transform="translate(50, 60)">
```

```
    <text x="-40" y="0" font-size="10" font-family="serif"
        fill="darkgreen">
        animateTransform
      <animateTransform attributeName="transform"
          type="scale" from="1" to="20"
          dur="10s" repeatCount="indefinite"/>
    </text>
  </g>
</svg>
```

5.3.4 回転のアニメーション

animateTransform 要素を使って回転のアニメーションを作る場合、from 属性と to 属性には、それぞれ、

θ, x, y

という形式の属性値を設定します。θ は回転の角度で、x と y は回転の中心の座標です。

SVG 文書の例 animrot.svg

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="320pt" height="224pt" viewBox="0 0 100 70"
    xmlns="http://www.w3.org/2000/svg">
  <g>
    <g stroke-width="2" stroke="steelblue">
      <line x1="40" y1="35" x2="60" y2="35"/>
      <line x1="50" y1="25" x2="50" y2="45"/>
    </g>
    <text x="55" y="45" font-size="14" font-family="serif"
        fill="cadetblue">
      rotate
    </text>
    <animateTransform attributeName="transform"
        type="rotate" from="0, 50, 35" to="360, 50, 35"
        dur="5s" repeatCount="indefinite"/>
  </g>
</svg>
```

5.3.5 剪断のアニメーション

animateTransform 要素を使って剪断のアニメーションを作る場合、from 属性と to 属性には、それぞれ、剪断の角度を設定します。

SVG 文書の例 animske.svg

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="320pt" height="224pt" viewBox="0 0 100 70"
    xmlns="http://www.w3.org/2000/svg"
    xmlns:xlink="http://www.w3.org/1999/xlink">
  <symbol id="cross" stroke-width="3">
    <line x1="10" y1="20" x2="30" y2="20"/>
    <line x1="20" y1="10" x2="20" y2="30"/>
  </symbol>
  <use xlink:href="#cross" stroke="navy">
    <animateTransform attributeName="transform"
        type="skewX" from="0" to="70"
        dur="5s" repeatCount="indefinite"/>
  </use>
  <use xlink:href="#cross" stroke="crimson">
    <animateTransform attributeName="transform"
        type="skewY" from="0" to="70"
        dur="5s" repeatCount="indefinite"/>
  </use>
  <use xlink:href="#cross" stroke="limegreen"/>
```

```
</svg>
```

5.4　パスに沿った移動のアニメーション

5.4.1　パスに沿ってグラフィックスを移動させる要素

animateMotion というアニメーション要素は、パスに沿って座標系を移動させることによってアニメーションを作ります。

animateMotion 要素が持っている attributeName 属性には、属性値を何も設定する必要はありません。

座標系がそれに沿って移動するパスを指定する方法は、二つあります。

ひとつは、animateMotion 要素が持っている path という属性に対してパスデータを設定するという方法です。そしてもうひとつは、path 要素に名前を付けておいて、それを名前で参照するという方法です。

path 要素を名前で参照したいときは、まず、path 要素が持っている id という属性に名前を設定することによって、それに名前を与えます。そのとき、path 要素を defs という要素型の要素の子供にしておくと、パスを描画しないで path 要素に名前を与えることができます。

path 要素を名前で参照したいときは、animateMotion 要素の子供として mpath という要素型の要素を書きます。mpath 要素のタグの中には、

```
<mpath xlink:href="#名前"/>
```

というように、path 要素を参照する URI を xlink:href 属性に設定する属性指定を書きます。

SVG 文書の例　animoti.svg

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="320pt" height="224pt" viewBox="0 0 100 70"
    xmlns="http://www.w3.org/2000/svg"
    xmlns:xlink="http://www.w3.org/1999/xlink">
  <defs>
    <path id="path1"
        d="M65,10 a25,25 0 1,1 0,50 l-30,0
           a25,25 0 1,1 0,-50 z"/>
  </defs>
  <use xlink:href="#path1"
      stroke-width="0.3" stroke="blue" fill="none"/>
  <rect x="-20" y="0" width="20" height="5" fill="tomato">
    <animateMotion dur="20s" repeatCount="indefinite">
      <mpath xlink:href="#path1"/>
    </animateMotion>
  </rect>
</svg>
```

5.4.2　パスの向きに応じた角度の変化

animateMotion 要素は、rotate という属性を持っています。これは、グラフィックスを回転させる角度を設定する属性なのですが、この属性に auto という属性値を設定すると、グラフィックスの角度が、パスの向きに応じて自動的に変化するようになります。

SVG 文書の例　autoro.svg

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="320pt" height="224pt" viewBox="0 0 100 70"
    xmlns="http://www.w3.org/2000/svg"
    xmlns:xlink="http://www.w3.org/1999/xlink">
  <defs>
    <path id="path1"
        d="M65,10 a25,25 0 1,1 0,50 l-30,0
           a25,25 0 1,1 0,-50 z"/>
```

```
  </defs>
  <use xlink:href="#path1"
      stroke-width="0.3" stroke="blue" fill="none"/>
  <rect x="-20" y="0" width="20" height="5" fill="tomato">
    <animateMotion dur="20s" repeatCount="indefinite"
        rotate="auto">
      <mpath xlink:href="#path1"/>
    </animateMotion>
  </rect>
</svg>
```

第 6 章　CSS

6.1　CSS の基礎

6.1.1　CSS の基礎の基礎

XML 文書を表示するために使われる、色、線、フォントなどの属性は、総称して「スタイル」(style) と呼ばれます。XML 文書のスタイルは、CSS(Cascading Style Sheets) などの言語を使うことによって記述することができます。CSS は、W3C によって勧告されている標準規格のひとつです。

SVG によって記述されるグラフィックスは、そのスタイルについても SVG 自身で記述することが可能です。しかし、CSS を使ってグラフィックスのスタイルを記述するということも可能です。CSS を使うことによって、スタイルの記述を SVG 文書から分離することができますので、いくつかの SVG 文書のスタイルを一箇所にまとめて記述するということが可能になります。

スタイルを記述した文書は、「スタイルシート」(style sheet) と呼ばれます。そして、スタイルシートが格納されているファイルは、「スタイルシートファイル」(style sheet file) と呼ばれます。CSS で書かれたスタイルシートが格納されているスタイルシートファイルのファイル名には、通常、 `.css` という拡張子を付けます。

6.1.2　スタイルシート処理命令

スタイルシートによって記述されたスタイルを SVG 文書に適用したいときは、XML 宣言とルート要素とのあいだに、「スタイルシート処理命令」(style sheet processing instruction) と呼ばれるものを書きます。

スタイルシート処理命令というのは、

```
<?xml-stylesheet type="MIME タイプ" href="URI"?>
```

という構文を持つ文字列のことです。この中に書く「MIME タイプ」(MIME type) というのは、言語またはデータ形式を示すために使われる、

トップレベルタイプ名 / サブタイプ名

という形式の文字列のことです。CSS の MIME タイプは、 `text/css` という文字列です。

スタイルシート処理命令の中の「URI」のところには、スタイルシートファイルの URI を書きます。SVG 文書のファイルとスタイルシートファイルとが同じディレクトリの中にある場合は、スタイルシートファイルのファイル名が URI になります。

たとえば、CSS で書かれたスタイルシートが、 `rect.css` というファイルに格納されていて、そのファイルが SVG 文書と同じディレクトリにあるとすれば、その SVG 文書の XML 宣言とルート要素とのあいだに、

```
<?xml-stylesheet type="text/css" href="rect.css"?>
```

というスタイルシート処理命令を書くことによって、そのスタイルシートによって記述されたスタイルをその SVG 文書に適用することができます。

次のスタイルシートは、青色で塗りつぶすというスタイルを長方形に適用する、ということを CSS で記述したものです。

スタイルシートの例　`rect.css`

```
rect { fill: blue; }
```

次の SVG 文書によって表示される長方形に対しては、上のスタイルシートによって記述されたスタイルが適用されます。

SVG 文書の例 sspi.svg

```
<?xml version="1.0" standalone="no"?>
<?xml-stylesheet type="text/css" href="rect.css"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="320pt" height="224pt" viewBox="0 0 100 70"
    xmlns="http://www.w3.org/2000/svg">
  <rect x="10" y="10" width="80" height="50"/>
</svg>
```

6.1.3 スタイルシートを書くための要素

スタイルシートは、SVG 文書の中に埋め込むことも可能です。

SVG は、スタイルシートを書くための要素型として、style という要素型を定義しています。スタイルシートは、style 要素の子供として書くことができます。

style 要素を書くときは、それが持っている type という属性に、スタイルシートを書くために使われている言語の MIME タイプを設定する必要があります。すでに述べたとおり、CSS の MIME タイプは、text/css です。

6.1.4 CDATA セクション

XML 文書の中に XML ではない文書を埋め込む場合は、通常、「CDATA セクション」(CDATA section) というものが使われます。

CDATA セクションというのは、<![CDATA[という文字列で始まって、]]> という文字列で終わる文字列のことです。それらの文字列に囲まれた部分には、どんな文字列を書いてもかまいません（ただし、]]> という文字列だけは書けません）。

XML 文書の中に書かれた CDATA セクションは、XML の文字列として解釈されるのではなくて、純粋に文字列として解釈されます。ですから、そこにどんな文字列を書いたとしても、XML 文書の構造に影響を与えることはありません。

style 要素の子供としてスタイルシートを書く場合も、通常、CDATA セクションを使って次のように書きます。

```
<style type="text/css"><![CDATA[
  スタイルシート
]]></style>
```

次の SVG 文書は、長方形を緑色で塗りつぶすというスタイルを、style 要素の中で CSS を使って記述しています。

SVG 文書の例 style.svg

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="320pt" height="224pt" viewBox="0 0 100 70"
    xmlns="http://www.w3.org/2000/svg">
  <style type="text/css"><![CDATA[
    rect { fill: green; }
  ]]></style>
  <rect x="10" y="10" width="80" height="50"/>
</svg>
```

6.2 ルール

6.2.1 ルールの基礎

スタイルシートというのは、基本的には、「ルール」(rule) と呼ばれる記述がいくつか並んだものだと考えることができます。

ルールというのは、「どのようなスタイルをどのような部分に適用するのか」ということを記述したもののことです。たとえば、

```
rect { fill: blue; }
```

というルールを書くことによって、「青色で塗りつぶすというスタイルを長方形に適用する」ということを記述することができます。

すべてのルールは、

セレクター　宣言ブロック

という構文にしたがって書かれます。たとえば、

```
rect { fill: blue; }
```

というルールの場合は、`rect` という部分がセレクターで、

```
{ fill: blue; }
```

という部分が宣言ブロックです。

6.2.2 セレクター

「セレクター」(selector) というのは、スタイルを適用する対象を記述したもののことです。セレクターとして要素型名を書くと、その要素型を持つすべての要素に対して、宣言ブロックで記述されたスタイルが適用されます。たとえば、

```
circle { fill: red; }
```

というルールを書くことによって、すべての `circle` 要素のグラフィックス（つまり円）を赤色で塗りつぶすことができます。

6.2.3 宣言ブロック

宣言ブロックは、かならず、左中括弧 (`{`) で始まって、右中括弧 (`}`) で終わります。そして、その中に、「宣言」(declaration) と呼ばれるものを書きます。宣言は、ひとつの宣言ブロックの中に何個でも好きなだけ書くことができます。たとえば、

```
{ fill: green; stroke: aqua; }
```

という宣言ブロックの中には、「緑色で塗りつぶす」という宣言と「線の色を水色にする」という宣言とが含まれています。

6.2.4 宣言

宣言というのは、「スタイルの特定の種類に対して、それをどうするのか」ということを記述したもののことです。すべての宣言は、

プロパティー : 値 ;

という構文にしたがって書かれます。たとえば、

```
fill: green;
```

という宣言の場合は、`fill` という部分がプロパティーで、`green` という部分が値です。

「プロパティー」(property) というのは、スタイルの種類をあらわしている名前のことです。たとえば、`fill` というのは塗りつぶしの色というスタイルの種類をあらわしているプロパティーで、`stroke` というのは線の色というスタイルの種類をあらわしているプロパティーです。

「値」(value) というのは、プロパティーで指定された種類のスタイルをどうするのかという具体的な記述のことです。

6.2.5 ホワイトスペースと注釈

CSS では、単語の前後に任意の個数のホワイトスペースを挿入することができます（ただし、コロン (`:`) の左側にホワイトスペースを挿入することはできません）。そして、ホワイトスペースの有無はスタイルシートの意味を変化させません。

CSS では、`/*` で始まって `*/` で終わる文字列は注釈とみなされます。注釈は、途中に改行を含んでいてもかまいません。

6.3 セレクター

6.3.1 クラスセレクター

SVG の主要な要素は、class という属性を持っています。これは、「クラス名」(class name) と呼ばれる名前を設定する属性です。いくつかの要素の class 属性に対して同一のクラス名を設定することによって、それらの要素がひとつのクラスに所属しているということを示すことができます。

クラスを指定するセレクターを書くことによって、ひとつのクラスに所属する複数の要素に対して同一のスタイルを適用する、ということが可能です。クラスを指定するセレクターは、「クラスセレクター」(class selector) と呼ばれます。

クラスセレクターは、要素型名の右側にドット (dot) を書いて、そのさらに右側にクラス名を書いたもの、つまり、

要素型名 . クラス名

という形の記述です。このようなセレクターを書くことによって、ドットの左に書かれた要素型名で指定された要素のうちで、ドットの右に書かれたクラス名が class 属性に設定されているものだけに対してスタイルを適用することができます。たとえば、

```
rect.special
```

というセレクターを書くことによって、special というクラス名が class 属性に設定されている rect 要素だけに対してスタイルを適用することができます。

クラスセレクターは、要素型名を省略して、

. クラス名

という形のものを書くこともできます。この場合は、要素型を問わず、記述されたクラス名が設定されているすべての要素に対してスタイルが適用されることになります。

スタイルシートの例 class.css

```
rect.normal { fill: blue; }
rect.special { fill: red; }
.deluxe { fill: green; }
```

SVG 文書の例 class.svg

```
<?xml version="1.0" standalone="no"?>
<?xml-stylesheet type="text/css" href="class.css"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="320pt" height="224pt" viewBox="0 0 100 70"
    xmlns="http://www.w3.org/2000/svg">
  <rect class="normal" x="5" y="5" width="42" height="28"/>
  <rect class="special" x="53" y="5" width="42" height="28"/>
  <rect class="deluxe" x="5" y="37" width="42" height="28"/>
  <ellipse class="deluxe" cx="74" cy="51" rx="21" ry="14"/>
</svg>
```

6.3.2 ID セレクター

スタイルを適用する対象を、要素の id 属性に設定された属性値、つまり要素の名前で指定するセレクターを書くことも可能です。そのようなセレクターは、「ID セレクター」(ID selector) と呼ばれます。

ID セレクターは、シャープ (sharp) の右側に名前を書いたもの、つまり、

名前

という形の記述です。このようなセレクターを書くことによって、シャープの右側に書かれた名前が id 属性に設定されている要素に対してスタイルを適用することができます。たとえば、

```
#id000
```

というセレクターを書くことによって、id000という名前がid属性に設定されている要素に対してスタイルを適用することができます。

スタイルシートの例 id.css

```
#id000 { fill: teal; }
#id001 { fill: orange; }
#id002 { fill: crimson; }
#id003 { fill: palegreen; }
```

SVG 文書の例 id.svg

```
<?xml version="1.0" standalone="no"?>
<?xml-stylesheet type="text/css" href="id.css"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="320pt" height="224pt" viewBox="0 0 100 70"
    xmlns="http://www.w3.org/2000/svg">
  <rect id="id000" x="5" y="5" width="42" height="28"/>
  <rect id="id001" x="53" y="5" width="42" height="28"/>
  <ellipse id="id002" cx="26" cy="51" rx="21" ry="14"/>
  <ellipse id="id003" cx="74" cy="51" rx="21" ry="14"/>
</svg>
```

6.3.3 子孫セレクター

XML 文書は、さまざまな要素を組み合わせることによって、木構造を構成しています。CSSでは、指定された要素の子孫になっている要素だけに対してスタイルを適用する、ということができるようになっています。スタイルを適用する対象として要素の子孫を指定するセレクターは、「子孫セレクター」(descendant selector) と呼ばれます。

子孫セレクターは、セレクターの右側に空白を書いて、その右側にさらにセレクターを書いたもの、つまり、

セレクター セレクター

という形の記述です。このようなセレクターを書くことによって、左に書かれたセレクターによって指定された要素の子孫になっている要素のうちで、右に書かれたセレクターによって指定される要素に対して、スタイルが適用されることになります。たとえば、

```
#id000 rect.normal
```

というセレクターを書くことによって、#id000というセレクターによって指定される要素の子孫になっている要素のうちで、rect.normalというセレクターによって指定される要素だけに対してスタイルを適用することができます。

子孫セレクターの全体もひとつのセレクターになりますので、セレクターは、

セレクター セレクター ・・・ セレクター

というように、いくつでも並べることができます。そうすることによって、○○の子孫のうちの○○の子孫のうちの……というように、スタイルを適用する要素を何重にも限定することができます。

スタイルシートの例 descend.css

```
#id000 rect { fill: skyblue; }
#id000 ellipse { fill: greenyellow; }
#id001 rect { fill: indigo; }
#id001 ellipse { fill: chocolate; }
```

SVG 文書の例 descend.svg

```
<?xml version="1.0" standalone="no"?>
<?xml-stylesheet type="text/css" href="descend.css"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="320pt" height="224pt" viewBox="0 0 100 70"
    xmlns="http://www.w3.org/2000/svg">
```

```
  <g id="id000">
    <rect x="5" y="5" width="42" height="28"/>
    <ellipse cx="74" cy="19" rx="21" ry="14"/>
  </g>
  <g id="id001">
    <rect x="5" y="37" width="42" height="28"/>
    <ellipse cx="74" cy="51" rx="21" ry="14"/>
  </g>
</svg>
```

6.3.4 グループ化

同一のスタイルを異なる対象に適用する複数のルールは、それらの対象を指定するひとつのセレクターを書くことによって、それらのルールをひとつにまとめることができます。同一のスタイルを対象に適用する複数のルールをひとつにまとめることを、「グループ化」(grouping) と呼びます。

複数の対象を指定するセレクターというのは、セレクターをコンマ (comma) で区切って並べたもののことです。たとえば、

```
rect, circle, polygon { fill: brown; }
```

というルールを書くことによって、長方形と円と多角形を茶色にすることができます。

スタイルシートの例 grouping.css

```
rect, .deluxe, #id000 { fill: orange; }
```

SVG 文書の例 grouping.svg

```
<?xml version="1.0" standalone="no"?>
<?xml-stylesheet type="text/css" href="grouping.css"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="320pt" height="224pt" viewBox="0 0 100 70"
    xmlns="http://www.w3.org/2000/svg">
  <rect x="5" y="5" width="60" height="30"/>
  <circle class="deluxe" cx="80" cy="20" r="15"/>
  <ellipse id="id000" cx="50" cy="50" rx="45" ry="15"/>
</svg>
```

6.4 プロパティー

6.4.1 色と不透明度に関するプロパティー

この節では、グラフィックスのスタイルを CSS で記述するときに使うことのできる主要なプロパティーを紹介したいと思います。

色と不透明度に関するプロパティーとしては、次のようなものがあります。色の値は第 1.4 節で説明した色名、16 進数、10 進数のいずれかで記述して、不透明度の値は 0 から 1 までの数値で記述します。

`fill`	塗りつぶしの色。
`stroke`	線の色。
`opacity`	グラフィックスの内部の不透明度。
`stroke-opacity`	線の不透明度。

スタイルシートの例 color.css

```
#id000 {
    fill: lightyellow;
    stroke: olive;
}

#id001 {
    fill: lightcyan;
    stroke: slateblue;
```

```
    opacity: 0.8;
    stroke-opacity: 0.6;
}
```

SVG 文書の例 color.svg

```
<?xml version="1.0" standalone="no"?>
<?xml-stylesheet type="text/css" href="color.css"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="320pt" height="224pt" viewBox="0 0 100 70"
    xmlns="http://www.w3.org/2000/svg">
  <g stroke-width="5">
    <ellipse id="id000" cx="35" cy="25" rx="30" ry="20"/>
    <ellipse id="id001" cx="65" cy="45" rx="30" ry="20"/>
  </g>
</svg>
```

6.4.2 線に関するプロパティー

線に関するプロパティーとしては、次のようなものがあります。線のスタイルについての詳細は、第 2.5 節を参照してください。

stroke-width	線の幅。
stroke-linecap	端点の形状。値は、 butt 、 round 、 square のいずれか。
stroke-linejoin	接続点の形状。値は、 miter 、 round 、 bevel のいずれか。
stroke-dasharray	破線のパターン。値は、線の長さと間隔の長さを、コンマで区切って並べたもの。

スタイルシートの例 stroke.css

```
line, polyline {
    fill: none;
    stroke: deepskyblue;
    stroke-width: 10;
}

#id000 {
    stroke-linecap: butt;
    stroke-linejoin: bevel;
}

#id001 {
    stroke-linecap: round;
    stroke-linejoin: round;
}

#id002 { stroke-dasharray: 10, 3, 2, 3; }
```

SVG 文書の例 stroke.svg

```
<?xml version="1.0" standalone="no"?>
<?xml-stylesheet type="text/css" href="stroke.css"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="320pt" height="224pt" viewBox="0 0 100 70"
    xmlns="http://www.w3.org/2000/svg">
  <polyline id="id000" points="10,45 27,10 44,45"/>
  <polyline id="id001" points="56,45 73,10 90,45"/>
  <line id="id002" x1="10" y1="60" x2="90" y2="60"/>
</svg>
```

6.4.3 テキストに関するプロパティー

テキストに関するプロパティーとしては、次のようなものがあります。テキストのスタイルについての詳細は、第 2.3 節を参照してください。

font-size	フォントの大きさ。
font-family	フォントの名前。
font-weight	フォントの太さ。値は、 normal や bold など。
font-style	フォントのスタイル。値は、 normal 、 italic 、 oblique など。
text-decoration	テキストの装飾。値は、 none 、 underline 、 overline 、 line-through など。
word-spacing	単語の間隔。
letter-spacing	文字の間隔。
text-anchor	テキストの配置。値は、 start 、 middle 、 end のいずれか。

スタイルシートの例　csstext.css

```
text {
    fill: orangered;
    font-size: 12;
    font-family: serif;
    word-spacing: 2;
    letter-spacing: 0.4;
    text-anchor: middle;
}

.emphasis {
    font-family: sans-serif;
    font-weight: bold;
    font-style: italic;
    text-decoration: underline;
}
```

SVG 文書の例　csstext.svg

```
<?xml version="1.0" standalone="no"?>
<?xml-stylesheet type="text/css" href="csstext.css"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="320pt" height="224pt" viewBox="0 0 100 70"
    xmlns="http://www.w3.org/2000/svg">
  <text x="50" y="25">normal text</text>
  <text class="emphasis" x="50" y="50">emphatic text</text>
</svg>
```

第 7 章　スクリプトの基礎

7.1　スクリプトの基礎の基礎

7.1.1　スクリプトとは何か

SVG 文書の中には、「スクリプト」(script) と呼ばれる記述を書くことができます。

スクリプトというのは、動作をあらわしている記述のことです。ウェブブラウザーは、SVG 文書の中に書かれたスクリプトによってあらわされている動作を実行することができます。

7.1.2　JavaScript とは何か

SVG 文書の中に書くスクリプトの記述には、通常、JavaScript というプログラミング言語が使われます。

JavaScript というのは、ECMA(European Computer Manufacturers Association) という標準化団体が策定した ECMA-262 という文書で定義されているプログラミング言語のことです。その文書は、自分が定義しているプログラミング言語を ECMAScript と呼んでいますので、JavaScript は通称で、ECMAScript が正式名称ということになります。

このチュートリアルのここから先の部分は、JavaScript がすでに使えるようになっている人を読者として想定して書かれています。ですので、まだ JavaScript が使えない読者は、この先の部分を読む前に JavaScript について勉強する必要があります。

JavaScript については、付録 B でも簡単に説明しています。ただし、その付録は、すでに何らかの言語でプログラムが書ける人を読者として想定して書かれていますので、プログラミング自体が初めての人は、JavaScript の入門書を読むほうがいいでしょう。

7.1.3 スクリプトを書くための要素

スクリプトは、SVG 文書の中のどこに書いてもいいというわけではありません。

スクリプトを書くことのできる場所は、 `script` という要素型の要素を書くことによって作ることができます。`script` 要素の子供としてスクリプトを書いておくと、そのスクリプトは、ウェブブラウザーが SVG 文書を読み込んだ時点で実行されます。

`script` 要素を書くときは、それが持っている `type` という属性に、スクリプトを書くために使われている言語の MIME タイプを設定する必要があります。

JavaScript の MIME タイプは `text/ecmascript` ですので、言語として JavaScript を使う場合、 `script` 要素は、

```
<script type="text/ecmascript">スクリプト</script>
```

と書くことになります。

なお、 `text/ecmascript` は、以前から使われてきた JavaScript の MIME タイプですが、現在では、RFC4329 で規定されているように、

```
application/javascript または application/ecmascript
```

というのが JavaScript の正式な MIME タイプです。しかし、あらゆるウェブブラウザーが正式な MIME タイプを認識することができるようになるのは当分先のことになると思われますので、このチュートリアルでは、以前から使われてきた MIME タイプを使うことにします。

スクリプトは、通常、XML 応用言語ではない言語を使って書かれます。ですから、スクリプトは普通、

```
<![CDATA[スクリプト]]>
```

というように、CDATA セクションの中に書かれます。

SVG 文書の例 `script.svg`

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="320pt" height="224pt" viewBox="0 0 100 70"
    xmlns="http://www.w3.org/2000/svg">
  <script type="text/ecmascript"><![CDATA[
    alert('Hello, world!');
  ]]></script>
</svg>
```

この SVG 文書のスクリプトの中で使われている `alert` というのは、引数として渡されたテキストをダイアログボックスで表示するメソッドです。テキストではないデータまたはオブジェクトが引数として渡された場合は、それをテキストに変換して表示します。

`alert` は、 `window` という名前のオブジェクトが持っているメソッドですので、本来は、

```
window.alert(式)
```

という式で呼び出さなければならないのですが、 `window.` は、省略することができます。

7.1.4 スクリプトファイル

スクリプトは、SVG 文書の中に書くことができるだけではなくて、SVG 文書から独立した文書として書いておいて、それを SVG 文書から参照することも可能です。

スクリプトだけから構成されている文書が格納されているファイルは、「スクリプトファイル」(script file) と呼ばれます。

スクリプトファイルのファイル名には、そこに格納されているスクリプトを書くために使われているプログラミング言語に応じた拡張子を付けます。スクリプトが JavaScript を使って書かれている場合の拡張子は、 `.js` です。

スクリプトファイルに格納されているスクリプトを SVG 文書から参照したいときは、script 要素の xlink:href 属性に、そのファイルを指定する URI を設定します。

それでは、次のスクリプトと SVG 文書を入力して、ブラウザーに SVG 文書を表示させてみてください。

スクリプトの例　hello.js

```
alert('Hello, world!');
```

SVG 文書の例　script2.svg

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="320pt" height="224pt" viewBox="0 0 100 70"
    xmlns="http://www.w3.org/2000/svg"
    xmlns:xlink="http://www.w3.org/1999/xlink">
  <script type="text/ecmascript" xlink:href="hello.js"/>
</svg>
```

7.2　イベント

7.2.1　イベントとは何か

人間がマウスなどを操作することによって発生する出来事は、「イベント」(event) と呼ばれます。SVG は、イベントが発生したときに、そのイベントに対応するスクリプトを実行することができる、という機能を持っています。

イベントが発生したときにスクリプトによって実行される動作は、「イベント処理」(event processing) と呼ばれます。

7.2.2　イベント属性

SVG で定義されている要素の多くは、「イベント属性」(event attribute) と呼ばれるいくつかの種類の属性を持っています。イベント属性に対してスクリプトを属性値として設定しておくと、そのスクリプトは、何らかのイベントが発生したときに実行されます。

イベント属性の種類は、イベントの種類に対応しています。ですから、イベント処理が実行されるようにしたいときは、どのような種類のイベントが発生したときにスクリプトを実行するのかということに応じて、その種類に対応するイベント属性にスクリプトを設定しておく必要があります。

マウス（一般的に言えばポインティング・デバイス）によるイベントに対応するイベント属性としては、次のようなものがあります。

onmouseover	グラフィックスの上に重なった。
onmouseout	グラフィックスの上から出ていった。
onmousedown	グラフィックスの上でボタンが押された。
onmouseup	グラフィックスの上でボタンが離された。
onmousemove	グラフィックスの上で移動した。
onclick	グラフィックスの上でクリックされた。

グラフィックスを描画する要素が持っているイベント属性に対してスクリプトを設定しておくと、そのスクリプトは、その要素によって描画されたグラフィックスの上でイベントが発生したときに実行されます。

SVG 文書の例　eventat.svg

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="320pt" height="224pt" viewBox="0 0 100 70"
    xmlns="http://www.w3.org/2000/svg">
  <ellipse cx="50" cy="20" rx="30" ry="10" fill="springgreen"
      onmouseover="alert('I am an ellipse.')"/>
  <rect x="20" y="40" width="60" height="20" fill="limegreen"
```

```
      onclick="alert('I am a rectangle.')"/>
</svg>
```

7.2.3 グループのイベント

グラフィックスのグループを作る要素が持っているイベント属性に対してスクリプトを設定しておくと、そのグループに属するどのグラフィックスについても、その上でイベントが発生した場合には、そのスクリプトが実行されることになります。

SVG 文書の例 grevent.svg

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="320pt" height="224pt" viewBox="0 0 100 70"
    xmlns="http://www.w3.org/2000/svg">
  <g fill="darkslateblue"
      onclick="alert('I am a group of graphics.')">
    <ellipse cx="50" cy="20" rx="30" ry="10"/>
    <rect x="20" y="40" width="60" height="20"/>
  </g>
</svg>
```

7.2.4 イベントハンドラー

script 要素の子供として書かれたスクリプトの中で定義されたものの名前は、SVG 文書の全体が有効範囲になります。ですから、script 要素の子供として書かれたスクリプトの中で定義されている関数やメソッドを呼び出すスクリプトをイベント属性に設定しておくと、イベントが発生したときにその関数やメソッドが呼び出されることになります。

イベント属性に設定されたスクリプトから呼び出される関数やメソッドは、「イベントハンドラー」(event handler) と呼ばれます。

SVG 文書の例 handler.svg

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="320pt" height="224pt" viewBox="0 0 100 70"
    xmlns="http://www.w3.org/2000/svg">
  <g fill="forestgreen" onclick="showDateAndTime()">
    <rect x="30" y="20" width="40" height="20"/>
    <text x="50" y="50" text-anchor="middle" font-size="8"
        font-family="serif">
      present date and time
    </text>
  </g>
  <script type="text/ecmascript"><![CDATA[
    function showDateAndTime() {
        alert((new Date()).toLocaleString());
    }
  ]]></script>
</svg>
```

7.3 イベントオブジェクト

7.3.1 イベントオブジェクトの基礎

SVG のスクリプトの中では、「イベントオブジェクト」(event object) と呼ばれるオブジェクトを扱うことができます。イベントオブジェクトというのは、発生したイベントをあらわしているオブジェクトのことです。

SVG のスクリプトの中では、evt という名前で、イベントオブジェクトを参照することができます。

イベントオブジェクトを扱うイベントハンドラーは、かならず、イベントオブジェクトを引数として受け取るように定義する必要があります。そして、イベントハンドラーを呼び出して、evt

から取得したイベントオブジェクトをイベントハンドラーに引数として渡すスクリプトを、イベント属性に設定しないといけません。そのようにしなければならない理由は、もしも、イベントオブジェクトを引数で受け取るのではなくて、イベントハンドラーが呼び出されたのちに evt から取得するように定義したとすると、取得までのタイムラグのあいだに発生した別のイベントを処理してしまう可能性があるからです。

SVG 文書の例 eventob.svg

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="320pt" height="224pt" viewBox="0 0 100 70"
    xmlns="http://www.w3.org/2000/svg">
  <g fill="cadetblue" onclick="showEventObject(evt)">
    <rect x="30" y="20" width="40" height="20"/>
    <text x="50" y="50" text-anchor="middle" font-size="8"
        font-family="serif">
      show event object
    </text>
  </g>
  <script type="text/ecmascript"><![CDATA[
    function showEventObject(evt) {
        alert(evt);
    }
  ]]></script>
</svg>
```

7.3.2 マウスポインターの座標

何らかのイベントが発生すると、イベントオブジェクトのプロパティーに、発生したイベントに関するさまざまなデータやオブジェクトが設定されます。たとえば、offsetX と offsetY というプロパティーには、イベントが発生したときのマウスポインターの座標（原点は、イベントが発生した要素の左上の隅）が設定されます。

イベントオブジェクトに設定される座標の単位は、viewBox 属性で設定した単位ではなくて、物理的なピクセルです。

SVG 文書の例 position.svg

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="320pt" height="224pt" viewBox="0 0 100 70"
    xmlns="http://www.w3.org/2000/svg">
  <rect x="0" y="0" width="100" height="70"
      fill="sandybrown" onclick="showPosition(evt)"/>
  <script type="text/ecmascript"><![CDATA[
    function showPosition(evt) {
        alert("offsetX = " + evt.offsetX + "\n" +
              "offsetY = " + evt.offsetY);
    }
  ]]></script>
</svg>
```

7.3.3 要素オブジェクト

XML の要素をあらわしているオブジェクトは、「要素オブジェクト」(element object) と呼ばれます。

イベントオブジェクトが持っている target というプロパティーには、イベントが発生したグラフィックスの要素をあらわしている要素オブジェクトが設定されます。

SVG 文書の例 elemobj.svg

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="320pt" height="224pt" viewBox="0 0 100 70"
    xmlns="http://www.w3.org/2000/svg">
```

```
  <rect x="10" y="10" width="30" height="20"
      fill="olivedrab" onclick="showElementObject(evt)"/>
  <ellipse cx="70" cy="20" rx="20" ry="10"
      fill="darkred" onclick="showElementObject(evt)"/>
  <polygon points="25,40 40,45 40,55 25,60 10,55 10,45"
      fill="darkslateblue" onclick="showElementObject(evt)"/>
  <path d="M 60,60 C 10,0 120,90 80,40"
      fill="darkmagenta" onclick="showElementObject(evt)"/>
  <script type="text/ecmascript"><![CDATA[
    function showElementObject(evt) {
        alert(evt.target);
    }
  ]]></script>
</svg>
```

第 8 章　DOM

8.1　DOM の基礎

8.1.1　DOM とは何か

プログラムやスクリプトは、通常、HTML 文書や XML 文書を操作する際に DOM(Document Object Model) と呼ばれるものを使います。

DOM というのは、W3C によって勧告されている標準規格のひとつで、HTML 文書や XML 文書を操作するプログラムやスクリプトのための API(application programming interface) を定めたものです。つまり、プログラムやスクリプトはどのような方法で HTML 文書や XML 文書にアクセスすればよいのか、ということを定めたものが DOM なのです。

DOM においては、HTML 文書や XML 文書を構成している要素や属性に対して、それらを追加したり削除したり変更したりする方法などが定められています。

8.1.2　バインディング

DOM 自体は、プログラミング言語に依存しない抽象的な仕様です。それに対して、特定のプログラミング言語を使って HTML 文書や XML 文書を操作するプログラムやスクリプトを書く際には、その言語に対応した具体的な仕様を使うことになります。

そこで必要になるのが、「バインディング」(binding) と呼ばれるものです。バインディングというのは、抽象的な仕様と具体的な言語とを結び付ける仕様のことです。

DOM とバインディングとは基本的には別のものですが、Java と JavaScript(ECMAScript) の二つの言語に関しては、DOM の仕様書の中にそれらの言語のバインディングも含まれています。

8.1.3　文書オブジェクト

DOM では、HTML 文書または XML 文書の全体をあらわすオブジェクトのことを「文書オブジェクト」(document object) と呼びます。

スクリプトの中では、`document` という名前で、文書オブジェクトを参照することができます。

8.1.4　ノード

DOM では、HTML 文書や XML 文書を、「ノード」(node) と呼ばれるものから構成される木構造として扱います。ノードとして扱われるのは、XML 文書を構成する要素と属性とテキストで、それぞれ、「要素ノード」(element node)、「属性ノード」(attribute node)、「テキストノード」(text node) と呼ばれます。

要素ノードをあらわすオブジェクトは、第 7.3 節ですでに説明したように、「要素オブジェクト」(element object) と呼ばれます。同じように、属性ノードをあらわすオブジェクトは「属性オブジェクト」(attribute object) と呼ばれ、テキストノードをあらわすオブジェクトは「テキストオブジェクト」(text object) と呼ばれます。

8.2 属性の操作

8.2.1 属性値の取得

要素オブジェクトは、属性に設定されている属性値を取り出すメソッドや、属性に属性値を設定するメソッドを持っています。ですから、それらのメソッドを使うことによって、スクリプトで要素の属性を操作することができます。

要素オブジェクトが持っている getAttributeNS というメソッドを呼び出すことによって、要素オブジェクトから、属性に設定されている属性値を取り出すことができます。

getAttributeNS は、2 個の引数を受け取ります。1 個目は名前空間名で、2 個目は属性名です。このメソッドは、1 個目の引数で指定された名前空間の中で、2 個目の引数で指定された属性に設定されている属性値を取り出して、それを戻り値として返します。

たとえば、ele という変数に要素オブジェクトが設定されているとするとき、

```
ele.getAttributeNS(null, "stroke-width")
```

という式で getAttributeNS を呼び出すことによって、stroke-width という属性に設定されている属性値を取得することができます。

この例のように、SVG の要素オブジェクトから属性値を取得する場合は、getAttributeNS の 1 個目の引数として、SVG の名前空間名ではなくて、null を渡す必要があります。

SVG 文書の例 getattr.svg

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="320pt" height="224pt" viewBox="0 0 100 70"
    xmlns="http://www.w3.org/2000/svg">
  <rect x="10" y="10" width="40" height="30"
      fill="aquamarine" onclick="showAttribute(evt)"/>
  <rect x="60" y="10" width="30" height="50"
      fill="midnightblue" onclick="showAttribute(evt)"/>
  <rect x="20" y="30" width="50" height="20"
      fill="plum" onclick="showAttribute(evt)"/>
  <rect x="30" y="20" width="10" height="40"
      fill="tomato" onclick="showAttribute(evt)"/>
  <script type="text/ecmascript"><![CDATA[
    function attributeValue(evt, attname) {
        return attname + ": " +
            evt.target.getAttributeNS(null, attname) + "\n";
    }

    function showAttribute(evt) {
        alert(attributeValue(evt, "x") +
              attributeValue(evt, "y") +
              attributeValue(evt, "width") +
              attributeValue(evt, "height") +
              attributeValue(evt, "fill"));
    }
  ]]></script>
</svg>
```

8.2.2 属性値の設定

要素オブジェクトが持っている setAttributeNS というメソッドを呼び出すことによって、要素オブジェクトに対して属性値を設定することができます。

setAttributeNS は、3 個の引数を受け取ります。1 個目は名前空間名、2 個目は属性名、3 個目は任意のデータです。このメソッドは、1 個目の引数で指定された名前空間の中で、2 個目の引数で指定された属性に対して、3 個目の引数を属性値として設定します（3 個目の引数が文字列ではない場合は、暗黙のうちに文字列に変換されます）。

たとえば、ele という変数に要素オブジェクトが設定されているとするとするとき、

```
ele.setAttributeNS(null, "height", 30)
```

という式で setAttributeNS を呼び出すことによって、height という属性に対して、30 という

文字列を属性値として設定することができます。

getAttributeNS と同じように、setAttributeNS の場合も、SVG の要素オブジェクトに属性値を設定するときは、1 個目の引数として、SVG の名前空間名ではなくて、null を渡す必要があります。

次の SVG 文書によって表示される楕円は、マウスポインターがその上に重なっているときだけ色が変化します。

SVG 文書の例　setattr.svg

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="320pt" height="224pt" viewBox="0 0 100 70"
    xmlns="http://www.w3.org/2000/svg">
  <ellipse cx="50" cy="35" rx="40" ry="20" fill="darkgreen"
      onmouseover="setOverColor(evt)"
      onmouseout="setOutColor(evt)"/>
  <script type="text/ecmascript"><![CDATA[
    function setOverColor(evt) {
        evt.target.setAttributeNS(null, "fill", "palegreen");
    }

    function setOutColor(evt) {
        evt.target.setAttributeNS(null, "fill", "darkgreen");
    }
  ]]></script>
</svg>
```

getAttributeNS の戻り値は常に文字列ですので、それを数値として扱いたい場合は、明示的に数値に変換する必要があります。JavaScript では、文字列から 0 を減算することによって、その文字列を数値に変換することができます。

次の SVG 文書によって表示される円は、マウスでクリックされると、半径が 1 だけ大きくなります。

SVG 文書の例　growth.svg

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="320pt" height="224pt" viewBox="0 0 100 70"
    xmlns="http://www.w3.org/2000/svg">
  <circle cx="50" cy="35" r="5" fill="forestgreen"
      onclick="growth(evt)"/>
  <script type="text/ecmascript"><![CDATA[
    var step = 1;

    function growth(evt) {
        var ele = evt.target;
        var radius = ele.getAttributeNS(null, "r") - 0;
        var newradius = radius + step;
        ele.setAttributeNS(null, "r", newradius);
    }
  ]]></script>
</svg>
```

8.3　テキストオブジェクト

8.3.1　テキストオブジェクトの取得

この節では、text 要素によって描画されるテキストを操作する方法について説明したいと思います。

テキストを操作するためには、まず最初に、text 要素からテキストオブジェクトを取得する必要があります。

要素オブジェクトは、firstChild というプロパティーを持っています。このプロパティーに

は、その要素オブジェクトがあらわしている要素ノードの最初の子供が設定されています。要素ノードの最初の子供がテキストノードの場合、このプロパティーに設定されているのは、そのテキストノードをあらわしているテキストオブジェクトです。

ですから、text 要素でイベントが発生したときに、

```
evt.target.firstChild
```

という式を評価すると、その text 要素の最初の子供になっているテキストをあらわしているテキストオブジェクトが値として得られます。

次の SVG 文書によって描画されるテキストをマウスでクリックすると、そのテキストをあらわしているテキストオブジェクトがダイアログボックスで表示されます。

SVG 文書の例 textobj.svg

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="320pt" height="224pt" viewBox="0 0 100 70"
    xmlns="http://www.w3.org/2000/svg">
  <text x="10" y="36" font-size="12" font-family="serif"
      fill="darkolivegreen"
      onclick="showFirstChild(evt)">I am a text.</text>
  <script type="text/ecmascript"><![CDATA[
    function showFirstChild(evt) {
        alert(evt.target.firstChild);
    }
  ]]></script>
</svg>
```

8.3.2 テキストの取得

テキストオブジェクトは、テキストそのものではありません。テキストそのものを処理するためには、テキストオブジェクトからテキストそのものを取得する必要があります。

テキストオブジェクトからテキストを取得したいときは、テキストオブジェクトが持っている data というプロパティーを使います。このプロパティーにはテキストそのものが設定されています。ですから、text 要素でイベントが発生したときに、

```
evt.target.firstChild.data
```

という式を評価すると、その text 要素の最初の子供になっているテキストが値として得られます。

次の SVG 文書によって描画されるテキストをマウスでクリックすると、そのテキストがダイアログボックスで表示されます。

SVG 文書の例 getdata.svg

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="320pt" height="224pt" viewBox="0 0 100 70"
    xmlns="http://www.w3.org/2000/svg">
  <g font-size="16" font-family="serif">
    <text x="10" y="20" fill="cornflowerblue"
        onclick="showText(evt)">hotokenoza</text>
    <text x="10" y="40" fill="dodgerblue"
        onclick="showText(evt)">tsuyukusa</text>
    <text x="10" y="60" fill="powderblue"
        onclick="showText(evt)">katabami</text>
  </g>
  <script type="text/ecmascript"><![CDATA[
    function showText(evt) {
        alert(evt.target.firstChild.data);
    }
  ]]></script>
</svg>
```

8.3.3 テキストの設定

テキストオブジェクトが持っている data プロパティーは、そこからテキストを取得することができるだけではなくて、そこにテキストを設定することも可能です。

たとえば、 text 要素でイベントが発生したときに、

```
evt.target.firstChild.data = "nazuna";
```

という式文を実行したとすると、その text 要素の最初の子供になっているテキストが nazuna というテキストに置き換わります。

次の SVG 文書によって描画されるテキストは、マウスでクリックしたときに表示されるダイアログボックスにテキストを入力することによって、自由に変更することができます。

SVG 文書の例 setdata.svg

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="320pt" height="224pt" viewBox="0 0 100 70"
    xmlns="http://www.w3.org/2000/svg">
  <text x="10" y="36" font-size="18" font-family="serif"
      fill="midnightblue"
      onclick="setText(evt)">default</text>
  <script type="text/ecmascript"><![CDATA[
    function setText(evt) {
        var text = evt.target.firstChild;
        text.data = prompt("new text: ", text.data);
    }
  ]]></script>
</svg>
```

この SVG 文書のスクリプトの中で使われている prompt というのは、テキストを読み込むためのダイアログボックスを表示するメソッドで、2 個の引数を受け取ります。引数の 1 個目はプロンプトとして表示するテキストで、2 個目は読み込むテキストのデフォルトです。戻り値として、読み込んだテキストを返します。 alert と同じように、 window というオブジェクトが持っているメソッドです。

8.4 名前による要素オブジェクトの取得

8.4.1 イベントが発生していない要素に対する処理

スクリプトが要素を処理するためには、その要素をあらわす要素オブジェクトを取得する必要があります。

要素オブジェクトを取得する方法としては、第 7.3 節で、イベントオブジェクトが持っている target プロパティーを使うという方法を紹介しましたが、この方法で取得することができるのは、イベントが発生した要素をあらわす要素オブジェクトだけです。イベントが発生していない要素をあらわす要素オブジェクトを取得したい場合、この方法は使えません。

この節では、イベントが発生していない要素をあらわす要素オブジェクトを取得する方法として、その要素の名前を使うというものを紹介したいと思います。

8.4.2 要素の名前

要素の名前を使って要素オブジェクトを取得するためには、前提として、その要素に名前が与えられている必要があります。

要素に名前を与えたいときは、その要素が持っている id という属性に属性値を設定します。そうすると、その属性値が要素の名前になります。たとえば、

```
<circle id="moon" cx="50" cy="35" r="20"/>
```

という要素は、 moon という名前を持つことになります。

8.4.3 名前によって要素オブジェクトを取得するメソッド

第 8.1 節で説明したように、スクリプトの中では、 document という名前で、文書オブジェクト (document object)、つまり XML 文書の全体をあらわしているオブジェクトを取得すること

ができます。

文書オブジェクトは、getElementById というメソッドを持っています。このメソッドは、引数として文字列を受け取って、その文字列を名前として持つ要素をあらわす要素オブジェクトを戻り値として返します。たとえば、

```
document.getElementById("moon")
```

という式を評価すると、その値として、moon という名前を持つ要素をあらわす要素オブジェクトが得られます。

次の SVG 文書は、長方形をクリックすることによって円を左右に移動させることができます。

SVG 文書の例 movelr.svg

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="320pt" height="224pt" viewBox="0 0 100 70"
    xmlns="http://www.w3.org/2000/svg">
  <circle id="ball" cx="50" cy="20" r="5" fill="seagreen"/>
  <g font-size="8" font-family="serif" text-anchor="middle">
    <g fill="plum" onclick="moveLeftRight(-1)">
      <rect x="20" y="40" width="20" height="15"/>
      <text x="30" y="62">left</text>
    </g>
    <g fill="turquoise" onclick="moveLeftRight(1)">
      <rect x="60" y="40" width="20" height="15"/>
      <text x="70" y="62" font-size="8">right</text>
    </g>
  </g>
  <script type="text/ecmascript"><![CDATA[
    var STEP = 3;
    var ballobj = document.getElementById("ball");
    var x = 50;

    function moveLeftRight(direction) {
        ballobj.setAttributeNS(null, "cx",
            x += direction*STEP);
    }
  ]]></script>
</svg>
```

同じように、次の SVG 文書は、長方形をクリックすることによって数字を増やしていくことができます。

SVG 文書の例 countup.svg

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="320pt" height="224pt" viewBox="0 0 100 70"
    xmlns="http://www.w3.org/2000/svg">
  <text id="number" x="10" y="30" font-size="30"
      font-family="serif" fill="orangered">0</text>
  <g font-size="8" font-family="serif" fill="indigo"
      onclick="countup()">
    <rect x="30" y="40" width="40" height="15"/>
    <text x="50" y="62" text-anchor="middle">countup</text>
  </g>
  <script type="text/ecmascript"><![CDATA[
    var numberobj =
        document.getElementById("number").firstChild;
    var count = 0;

    function countup() {
        numberobj.data = ++count;
    }
  ]]></script>
</svg>
```

同じように、次の SVG 文書は、マウスポインターの座標を表示します。

SVG 文書の例 mocoord.svg

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="320pt" height="224pt" viewBox="0 0 100 70"
    xmlns="http://www.w3.org/2000/svg">
  <text id="coordinate" x="10" y="40"
      font-size="17" font-family="serif"
      fill="darkturquoise">(0,0)</text>
  <rect x="0" y="0" width="100" height="70" opacity="0"
      onmousemove="setCoordinate(evt)"/>
  <script type="text/ecmascript"><![CDATA[
    var coordinateobj =
        document.getElementById("coordinate").firstChild;

    function setCoordinate(evt) {
        coordinateobj.data =
            "(" + evt.offsetX + "," + evt.offsetY + ")";
    }
  ]]></script>
</svg>
```

8.5 ルート要素オブジェクト

8.5.1 ルート要素オブジェクトとは何か

1.1 で説明したように、XML 文書の本体に相当する部分は、「ルート要素」(root element) と呼ばれます。

ルート要素というのもひとつの要素ですから、DOM では、ルート要素も要素オブジェクトによってあらわされます。ルート要素をあらわす要素オブジェクトは、「ルート要素オブジェクト」(root element object) と呼ばれます。

8.1 で説明したように、HTML 文書または XML 文書の全体をあらわすオブジェクトは、「文書オブジェクト」(document object) と呼ばれます。スクリプトの中では、document という名前で、文書オブジェクトを参照することができます。

ルート要素というのは文書の全体ではありませんので、ルート要素オブジェクトと文書オブジェクトとは、同じものではありません。

8.5.2 ルート要素オブジェクトの取得

ところで、ルート要素オブジェクトを取得したいときは、どうすればいいのでしょうか。

getElementById を使うことによってルート要素オブジェクトを取得するということも可能ですが、それよりももっと手軽な方法があります。それは、ルート要素オブジェクトが設定されているプロパティーを使うという方法です。

文書オブジェクトは、documentElement というプロパティーを持っていて、そこにはルート要素オブジェクトが設定されています。ですから、このプロパティーを参照するだけで、ルート要素オブジェクトを取得することができます。

次の SVG 文書は、長方形をクリックすると、ルート要素が持っている属性に設定されている属性値を表示します。

SVG 文書の例 root.svg

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="320pt" height="224pt" viewBox="0 0 100 70"
    xmlns="http://www.w3.org/2000/svg">
  <g font-size="8" font-family="serif" fill="yellowgreen"
      onclick="showRootAttributes()">
    <rect x="30" y="20" width="40" height="15"/>
    <text x="50" y="42" text-anchor="middle">
```

```
      show root attributes
    </text>
  </g>
  <script type="text/ecmascript"><![CDATA[
    var rootobj = document.documentElement;
    var width = rootobj.getAttributeNS(null, "width");
    var height = rootobj.getAttributeNS(null, "height");
    var viewBox = rootobj.getAttributeNS(null, "viewBox");

    function showRootAttributes() {
        alert("width: " + width + "\n" +
              "height: " + height + "\n" +
              "viewBox: " + viewBox);
    }
  ]]></script>
</svg>
```

8.6 要素の追加と削除

8.6.1 要素オブジェクトの生成

この節では、SVG 文書に要素を追加したり削除したりする方法について説明したいと思っているわけですが、その説明を始める前に、まず、新しい要素オブジェクトを生成する方法について説明しておきたいと思います。

文書オブジェクトは、createElementNS というメソッドを持っています。これは、新しい要素オブジェクトを生成するメソッドです。

createElementNS は、2 個の引数を受け取ります。1 個目は名前空間名で、2 個目は要素型名です。このメソッドは、1 個目の引数で指定された名前空間の中で、2 個目の引数で指定された要素型の要素をあらわす要素オブジェクトを生成して、それを戻り値として返します。たとえば、SVG という変数に SVG の名前空間名が設定されているとするとき、

```
document.createElementNS(SVG, "rect")
```

という式で createElementNS を呼び出すことによって、SVG の rect 要素をあらわす要素オブジェクトを生成することができます。

createElementNS で新しく生成された直後の要素オブジェクトは、属性が何も設定されていない状態になっています。要素オブジェクトに対して属性を設定したいときは、第 8.2 節で紹介した setAttributeNS を使います。たとえば、

```
ele.setAttributeNS(null, "width", "30")
```

という式を評価することによって、ele という変数に設定されている要素オブジェクトの width という属性に対して 30 という属性値を設定することができます。

8.6.2 要素の追加

要素オブジェクトは、appendChild というメソッドを持っています。これは、要素に対してその子供を追加するメソッドです。このメソッドは、要素オブジェクトを引数として受け取って、それがあらわしている要素を最後の子供として要素に追加します。たとえば、a と b が要素オブジェクトだとするとき、

```
a.appendChild(b);
```

という式文を実行することによって、a に対して、その最後の子供として b を追加することができます。

次の SVG 文書は、長方形をクリックすると、ランダムな半径と色を持つ円をランダムな位置に追加します。

SVG 文書の例 append.svg

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="320pt" height="224pt" viewBox="0 0 100 70"
    xmlns="http://www.w3.org/2000/svg">
```

```
  <g fill="mediumslateblue" onclick="appendCircle()">
    <rect x="10" y="55" width="40" height="10"/>
    <text x="55" y="63" font-size="12">append</text>
  </g>
  <script type="text/ecmascript"><![CDATA[
    var SVG = "http://www.w3.org/2000/svg";
    var rootobj = document.documentElement;

    function randomInt(range) {
        return Math.floor(Math.random()*range);
    }

    function randomColor() {
        return "rgb(" + randomInt(255) + ","
            + randomInt(255) + "," + randomInt(255) + ")";
    }

    function appendCircle() {
        var circle = document.createElementNS(SVG, "circle");
        circle.setAttributeNS(null, "cx", randomInt(100));
        circle.setAttributeNS(null, "cy", randomInt(45));
        circle.setAttributeNS(null, "r", randomInt(8)+2);
        circle.setAttributeNS(null, "fill", randomColor());
        rootobj.appendChild(circle);
    }
  ]]></script>
</svg>
```

このSVG文書のスクリプトの中で使われているMath.randomというのは、0から1までの範囲で擬似乱数を発生させるメソッドで、発生させた乱数を戻り値として返します。

また、Math.floorというのは、引数として受け取った数値を超えない最大の整数を戻り値として返すメソッドです。

8.6.3 テキストオブジェクトの生成

文書オブジェクトは、createTextNodeというメソッドを持っています。これは、新しいテキストオブジェクトを生成するメソッドです。このメソッドは、テキストを引数として受け取って、そのテキストが設定されたテキストオブジェクトを生成して、それを戻り値として返します。たとえば、

```
document.createTextNode("eris")
```

という式を評価することによって、erisというテキストが設定されたテキストオブジェクトを生成することができます。

テキストオブジェクトを要素オブジェクトの子供として追加したいときは、要素オブジェクトを追加する場合と同じように、要素オブジェクトが持っているappendChildを呼び出して、引数としてテキストオブジェクトを渡します。

次のSVG文書は、長方形をクリックすると、ランダムな大きさと色を持つランダムな文字をランダムな位置に追加します。

SVG文書の例 cretext.svg

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="320pt" height="224pt" viewBox="0 0 100 70"
    xmlns="http://www.w3.org/2000/svg">
  <g fill="darkcyan" onclick="appendText()">
    <rect x="10" y="55" width="40" height="10"/>
    <text x="55" y="63" font-size="12">append</text>
  </g>
  <script type="text/ecmascript"><![CDATA[
    var SVG = "http://www.w3.org/2000/svg";
    var rootobj = document.documentElement;

    function randomInt(range) {
```

```
        return Math.floor(Math.random()*range);
    }

    function randomChar() {
        return String.fromCharCode(randomInt(93)+33);
    }

    function randomColor() {
        return "rgb(" + randomInt(255) + ","
            + randomInt(255) + "," + randomInt(255) + ")";
    }

    function appendText() {
        var textobj = document.createTextNode(randomChar());
        var textele = document.createElementNS(SVG, "text");
        textele.appendChild(textobj);
        textele.setAttributeNS(null, "x", randomInt(100));
        textele.setAttributeNS(null, "y", randomInt(50));
        textele.setAttributeNS(null, "font-family", "serif");
        textele.setAttributeNS(null, "font-size",
            randomInt(32)+8);
        textele.setAttributeNS(null, "fill", randomColor());
        rootobj.appendChild(textele);
    }
  ]]></script>
</svg>
```

この SVG 文書のスクリプトの中で使われている `String.fromCharCode` というのは、引数として整数を受け取って、その整数を文字コードとする文字から構成される文字列を返すメソッドです。この SVG 文書では引数を 1 個だけしか渡していませんが、引数は何個でも渡すことができます。たとえば、

```
String.fromCharCode(110, 97, 109, 97, 107, 111)
```

という式を評価すると、値として `namako` という文字列が得られます。

8.6.4 要素の削除

要素オブジェクトは、`removeChild` というメソッドを持っています。これは、要素からその子供を削除するメソッドです。このメソッドは、要素オブジェクトを引数として受け取って、それがあらわしている要素を削除します。たとえば、a と b が要素オブジェクトで、b が a の子供だとするとき、

```
a.removeChild(b);
```

という式文を実行することによって、a から b を削除することができます。

次の SVG 文書は、円をクリックすることによって、その円を削除することができます。

SVG 文書の例 `remove.svg`

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="320pt" height="224pt" viewBox="0 0 100 70"
    xmlns="http://www.w3.org/2000/svg">
  <script type="text/ecmascript"><![CDATA[
    var SVG = "http://www.w3.org/2000/svg";
    var rootobj = document.documentElement;
    initialize();

    function removeCircle(evt) {
        rootobj.removeChild(evt.target);
    }

    function appendCircle(cx, cy, r, fill, onclick) {
        var circle = document.createElementNS(SVG, "circle");
        circle.setAttributeNS(null, "cx", cx);
        circle.setAttributeNS(null, "cy", cy);
```

```
        circle.setAttributeNS(null, "r", r);
        circle.setAttributeNS(null, "fill", fill);
        circle.setAttributeNS(null, "onclick", onclick);
        rootobj.appendChild(circle);
    }

    function initialize() {
        for (var x = 10; x <= 90; x += 5)
            for (var y = 10; y <= 60; y += 5)
                appendCircle(x, y, 3, "darkslateblue",
                    "removeCircle(evt)");
    }
  ]]></script>
</svg>
```

8.7 DOM によるイベント処理

8.7.1 DOM によるイベント処理の基礎

イベント処理については、第 7 章で、イベント属性を使う方法について説明しました。しかし、イベントを処理する方法は、それだけではありません。もうひとつ、DOM を使ってイベントを処理するという方法もあります。

DOM を使ってイベントを処理したいときは、「イベントリスナー」(event listener) と呼ばれるオブジェクトを要素オブジェクトに設定しておきます。イベントリスナーは、自分の中に関数またはメソッドを持っていて、イベントが発生すると、自分が持っている関数またはメソッドを呼び出して、発生したイベントについての情報を持っているオブジェクトを引数として渡します。

イベントリスナーが持っている関数またはメソッドは、「イベントハンドラー」(event handler) と呼ばれます。そして、イベントハンドラーが受け取る引数は、「イベントオブジェクト」(event object) と呼ばれます。

8.7.2 イベントリスナーを設定するメソッド

要素オブジェクトは、`addEventListener` というメソッドを持っています。このメソッドを使うことによって、イベントリスナーを要素オブジェクトに設定することができます。

`addEventListener` は、3 個の引数を受け取ります。

1 個目の引数は、イベントの種類を指定する、「イベント名」(event name) と呼ばれる文字列です。イベント名には、次のようなものがあります。

`click`	マウスでクリックされた。
`mousedown`	マウスのボタンが押された。
`mouseup`	マウスのボタンが離された。
`mouseover`	マウスポインターが重なった。
`mouseout`	マウスポインターが外へ出た。
`mousemove`	マウスが移動した。
`keydown`	キーボードのキーが押された。
`keyup`	キーボードのキーが離された。

2 個目の引数は、イベントが発生したときに呼び出されるイベントハンドラーです。式として、イベントハンドラーの名前または関数式を書きます。

3 個目の引数は、イベントを伝達する方向を示す真偽値です。真偽値のそれぞれは、次のような意味に解釈されます。

`true`	ルート要素からイベントが発生した要素へ
`false`	イベントが発生した要素からルート要素へ

8.7.3 マウスによるイベント

それでは、マウスによるクリックのイベントを処理する SVG 文書を、DOM を使って書いてみましょう。

マウスによるイベントが発生したときのマウスポインターの座標は、イベントオブジェクトが持っている offsetX と offsetY というプロパティーに設定されます。

次の SVG 文書は、マウスでクリックされると、そのときのマウスポインターの座標をダイアログボックスで表示します。

SVG 文書の例　click.svg

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="320pt" height="224pt" viewBox="0 0 100 70"
    xmlns="http://www.w3.org/2000/svg">
  <rect id="rect" x="0" y="0" width="100" height="70"
      fill="springgreen"/>
  <script type="text/ecmascript"><![CDATA[
    var rectobj = document.getElementById("rect");
    rectobj.addEventListener("click", function(event) {
        alert("offsetX = " + event.offsetX + "\n" +
              "offsetY = " + event.offsetY);
    }, false);
  ]]></script>
</svg>
```

8.7.4　キーボードによるイベント

次に、キーボードによるイベントを処理する SVG 文書を、DOM を使って書いてみましょう。

キーボードによるイベントを処理するためのイベントリスナーは、通常、XML 文書のルート要素のオブジェクトに設定します。

キーボードによるイベントが発生した場合、イベントオブジェクトが持っている keyCode というプロパティーには、イベントを発生させたキーを識別する、「キーコード」(key code) と呼ばれる整数が設定されます。

次の SVG 文書は、キーボードのキーが押されると、そのキーのキーコードを表示します。

SVG 文書の例　keydown.svg

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="320pt" height="224pt" viewBox="0 0 100 70"
    xmlns="http://www.w3.org/2000/svg">
  <text id="text" x="10" y="36" font-size="12"
      font-family="serif" fill="chocolate">keyCode = </text>
  <script type="text/ecmascript"><![CDATA[
    var text = document.getElementById("text").firstChild;
    document.documentElement.addEventListener("keydown",
        function(event) {
            text.data = "keyCode = " + event.keyCode;
        }, false);
  ]]></script>
</svg>
```

第 9 章　スクリプトとアニメーション

9.1　アニメーションの制御

9.1.1　アニメーションの開始

この節では、スクリプトを使ってアニメーションを制御する方法について説明したいと思います。

まず最初に、スクリプトでアニメーションを開始させる方法を紹介しましょう。

アニメーション要素の要素オブジェクトは、beginElement というメソッドを持っています。このメソッドを呼び出すことによって（引数は不要です）、アニメーションを開始させることができます。ただし、このメソッドでアニメーションを開始させるためには、あらかじめ、アニメー

ション要素が持っている begin という属性に、 indefinite という属性値を設定しておく必要があります。

たとえば、次の SVG 文書は、円をクリックすることによって、長方形の高さを変化させるアニメーションを開始させることができます。

SVG 文書の例 begin.svg

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="320pt" height="224pt" viewBox="0 0 100 70"
    xmlns="http://www.w3.org/2000/svg">
  <rect x="10" y="10" width="80" height="40" fill="violet">
    <animate id="animate" begin="indefinite"
        attributeName="height" from="0" to="40" dur="5s"
        repeatCount="1" fill="freeze"/>
  </rect>
  <g fill="darkorchid" onclick="begin()">
    <circle cx="30" cy="60" r="5"/>
    <text x="40" y="64" font-size="12">begin</text>
  </g>
  <script type="text/ecmascript"><![CDATA[
    var animateobj = document.getElementById("animate");

    function begin() {
        animateobj.beginElement();
    }
  ]]></script>
</svg>
```

9.1.2 アニメーションの終了

次に、スクリプトでアニメーションを終了させる方法を紹介しましょう。

アニメーション要素の要素オブジェクトは、 endElement というメソッドを持っています。このメソッドを呼び出すことによって（引数は不要です）、アニメーションを終了させることができます。ただし、このメソッドでアニメーションを終了させるためには、あらかじめ、アニメーション要素が持っている end という属性に、 indefinite という属性値を設定しておく必要があります。

デフォルトでは、アニメーションの動作中に beginElement が呼び出されると、その時点でアニメーションは最初の状態に戻って繰り返されます。

しかし、アニメーション要素が持っている restart という属性に、 whenNotActive という属性値を設定しておくことによって、アニメーションの動作中に beginElement が呼び出されたとしても、それを無視してアニメーションを続行させることができます。

たとえば、次の SVG 文書は、円をクリックすることによって、パスに沿って座標系を移動させるアニメーションを、開始させたり終了させたりすることができます。

SVG 文書の例 end.svg

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="320pt" height="224pt" viewBox="0 0 100 70"
    xmlns="http://www.w3.org/2000/svg"
    xmlns:xlink="http://www.w3.org/1999/xlink">
  <defs>
    <path id="path1"
        d="M0,0 a25,25 0 1,1 0,50 l-30,0
           a25,25 0 1,1 0,-50 z"/>
  </defs>
  <circle cx="65" cy="10" r="5" fill="limegreen">
    <animateMotion id="animoti" dur="10s" begin="indefinite"
        end="indefinite" repeatCount="indefinite"
        restart="whenNotActive">
      <mpath xlink:href="#path1"/>
    </animateMotion>
  </circle>
```

```
  <g font-size="12" font-family="serif">
    <g fill="deepskyblue" onclick="begin()">
      <circle cx="35" cy="25" r="5"/>
      <text x="45" y="28">begin</text>
    </g>
    <g fill="orangered" onclick="end()">
      <circle cx="35" cy="45" r="5"/>
      <text x="45" y="48">end</text>
    </g>
  </g>
  <script type="text/ecmascript"><![CDATA[
    var animotiobj = document.getElementById("animoti");

    function begin() {
        animotiobj.beginElement();
    }

    function end() {
        animotiobj.endElement();
    }
  ]]></script>
</svg>
```

9.1.3　アニメーションイベント属性

アニメーション要素は、「アニメーションイベント属性」(animation event attribute) と呼ばれるいくつかの属性を持っています。それらの属性にスクリプトを設定しておくと、そのスクリプトは、アニメーションに関連するイベントが発生したときに実行されます。

アニメーションイベント属性としては、次のようなものがあります。

`onbegin`	アニメーションが始まった。
`onend`	アニメーションが終了した。
`onrepeat`	アニメーションの繰り返しが始まった。

たとえば、次の SVG 文書は、アニメーションの繰り返しが始まるたびに、数字を 1 ずつ大きくしていきます。

SVG 文書の例　`onrepeat.svg`

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="320pt" height="224pt" viewBox="0 0 100 70"
    xmlns="http://www.w3.org/2000/svg">
  <line x1="50" y1="40" x2="50" y2="5" stroke-width="2"
      stroke="cornflowerblue">
    <animateTransform attributeName="transform" type="rotate"
        from="0,50,35" to="360,50,35" dur="1s"
        repeatCount="indefinite" onrepeat="countup()"/>
  </line>
  <text id="times" x="10" y="30" font-size="24"
      font-family="serif" fill="olive">0</text>
  <script type="text/ecmascript"><![CDATA[
    var count = 0;
    var timesobj =
        document.getElementById("times").firstChild;

    function countup() {
        timesobj.data = ++count;
    }
  ]]></script>
</svg>
```

9.2 スクリプトによるアニメーション

9.2.1 この節について

アニメーションは、animate 要素などのアニメーション要素ではなくて、スクリプトを使って作ることも可能です。また、スクリプトを使うことによって、アニメーション要素では作ることのできないようなアニメーションも可能になります。

この節では、スクリプトを使ってアニメーションを作る方法について説明したいと思います。

9.2.2 バックグラウンド処理

SVG のスクリプトは、基本的には、何らかのイベントが発生したときに実行されます。しかし、場合によっては、イベントの発生とは無関係に何らかの処理を実行することが必要になることもあります。

イベントの発生とは無関係に実行される処理は、「バックグラウンド処理」(background processing) と呼ばれます。アニメーションも、バックグラウンド処理を必要とします。

バックグラウンド処理は、通常、何らかの動作を一定の間隔で実行することによって実現されます。

9.2.3 バックグラウンド処理の設定

バックグラウンド処理を実行するためには、どのような動作を実行するのかということと、それを実行する時間の間隔を設定する必要があります。

JavaScript では、setInterval というメソッドを呼び出すことによって、バックグラウンド処理の設定をすることができます。

setInterval を呼び出す式は、

setInterval(関数 , 時間間隔)

と書きます。「関数」のところには、バックグラウンド処理として実行したい関数を値とする式（関数名または関数式）を書いて、「時間間隔」のところには、そのスクリプトを実行する時間間隔（整数で、単位はミリ秒）を値とする式を書きます。たとえば、

```
setInterval(namako, 1000)
```

という式で setInterval を呼び出すことによって、1 秒ごとに namako という関数を呼び出すというバックグラウンド処理を設定することができます。

次の SVG 文書は、バックグラウンド処理で円を左右に移動させます。

SVG 文書の例　setint.svg

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="320pt" height="224pt" viewBox="0 0 100 70"
    xmlns="http://www.w3.org/2000/svg">
  <circle id="ball" cx="5" cy="35" r="5" fill="darkcyan"/>
  <script type="text/ecmascript"><![CDATA[
    var ballobj = document.getElementById("ball");
    var x = 5;
    var direction = "right";
    setInterval(function() {
        if (direction == "right") {
            if (x >= 95)
                direction = "left";
            else
                ballobj.setAttributeNS(null, "cx", x++);
        } else {
            if (x <= 5)
                direction = "right";
            else
                ballobj.setAttributeNS(null, "cx", x--);
        }
    }, 10);
  ]]></script>
</svg>
```

次の SVG 文書は、現在の時刻を表示し続けます。

SVG 文書の例　time.svg

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="320pt" height="224pt" viewBox="0 0 100 70"
    xmlns="http://www.w3.org/2000/svg">
  <text id="time" x="50" y="42" font-size="22"
      font-family="serif" fill="navy" text-anchor="middle">
  </text>
  <script type="text/ecmascript"><![CDATA[
    var timeobj = document.getElementById("time").firstChild;
    setInterval(function() {
        var now = new Date();
        timeobj.data = now.toTimeString().substring(0, 8);
    }, 100);
  ]]></script>
</svg>
```

9.2.4　バックグラウンド処理の設定の解除

バックグラウンド処理の設定は、解除することも可能です。ただし、バックグラウンド処理の設定を解除するためには、そのバックグラウンド処理を識別する ID が必要になります。

setInterval は、自分が設定したバックグラウンド処理を識別するための ID を戻り値として返します。ですから、その戻り値を変数に設定しておけば、それを使ってその設定を解除することができます。

バックグラウンド処理の設定を解除したいときは、clearInterval というメソッドを呼び出します。このメソッドは、バックグラウンド処理を識別する ID を引数として受け取って、その ID によって識別されるバックグラウンド処理の設定を解除します。

次の SVG 文書は、長方形をクリックすることによって、数字の増加を開始させたり中断させたり停止させたりすることができます。

SVG 文書の例　watch.svg

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="320pt" height="224pt" viewBox="0 0 100 70"
    xmlns="http://www.w3.org/2000/svg">
  <text id="time" x="10" y="34" font-size="32"
      font-family="serif" fill="coral">0</text>
  <g font-size="8" font-family="serif" text-anchor="middle">
    <g fill="hotpink" onclick="start()">
      <rect x="10" y="45" width="20" height="10"/>
      <text x="20" y="62">start</text>
    </g>
    <g fill="forestgreen" onclick="lap()">
      <rect x="40" y="45" width="20" height="10"/>
      <text x="50" y="62">lap</text>
    </g>
    <g fill="cornflowerblue" onclick="stop()">
      <rect x="70" y="45" width="20" height="10"/>
      <text x="80" y="62">stop</text>
    </g>
  </g>
  <script type="text/ecmascript"><![CDATA[
    var timeobj =
        document.getElementById("time").firstChild;
    var id = null;
    var time = 0;
    var display = true;

    function start() {
        if (id == null) {
```

```
            id = setInterval(function() {
                time++;
                if (display)
                    timeobj.data = time;
            }, 10);
            time = 0;
            display = true;
        }
    }

    function lap() {
        display = ! display;
    }

    function stop() {
        if (id != null) {
            clearInterval(id);
            id = null;
        }
    }
  ]]></script>
</svg>
```

第 10 章　Raphaël

10.1　Raphaël の基礎

10.1.1　Raphaël とは何か

この章では、Raphaël というものについて解説したいと思います。

Raphaël というのは、Dmitry Baranovskiy さんという人が開発した JavaScript のライブラリーです。このライブラリーを使うことによって、SVG をベースにしたベクターグラフィックスを簡単に生成することができます。

10.1.2　Raphaël を使うための準備

Raphaël を使うためには、まず、それを入手する必要があります。Raphaël は、

Raphaël—JavaScript Library　`http://raphaeljs.com/`

という、Raphaël の公式サイトからダウンロードすることができます。ライセンスは MIT license です。

Raphaël のメソッドは、HTML 文書の中のスクリプトから呼び出すことができます。ただし、そのためには、その HTML 文書の中に、Raphaël を読み込む `script` 要素を書く必要があります。Raphaël のファイルが HTML 文書と同じディレクトリにあって、そのファイル名が `raphael.js` だとするならば、`script` 要素は次のようになります。

```
<script type="text/javascript" src="raphael.js"></script>
```

これを書く場所は、`head` 要素の中でも、`body` 要素の中でも、どちらでもかまいません。ただし、Raphaël を使うスクリプトの `script` 要素は、Raphaël を読み込む `script` 要素よりも下に書かないといけません。

10.1.3　キャンバスオブジェクト

Raphaël を使ってグラフィックスを描画するためには、まず最初に、「キャンバスオブジェクト」(canvas object) と呼ばれるものを生成する必要があります。

Raphaël では、グラフィックスがその上に描画される領域のことを「キャンバス」(canvas) と呼びます。キャンバスオブジェクトは、キャンバスをあらわすオブジェクトだと考えることができます。さらに、キャンバスオブジェクトは、グラフィックスの描画に関するさまざまな状態を保持していて、形状を描画するメソッドも持っています。

キャンバスオブジェクトは、`Raphael` という関数を呼び出すことによって生成することができます。この関数には、キャンバスの位置と大きさを示す、次の 4 個の引数を渡します（単位はピ

クセルです)。

1 個目　左上の隅の x 座標。

2 個目　左上の隅の y 座標。

3 個目　横の長さ。

4 個目　縦の長さ。

たとえば、次のようなコードを書くことによって、ページの左上に接する位置に、横の長さが 400 ピクセル、縦の長さが 300 ピクセルのキャンバスをあらわすキャンバスオブジェクトを生成して、それを paper という変数に設定することができます。

```
var paper = Raphael(0, 0, 400, 300);
```

10.1.4 形状を描画するメソッド

Raphaël では、キャンバスオブジェクトが持っているメソッドを呼び出すことによって、キャンバスの上に形状を描画することができます。形状を描画するメソッドの名前は、その形状に対応する SVG の要素の名前と同じです。

たとえば、楕円は、 ellipse というメソッドを呼び出すことによって描画することができます。このメソッドは、次の 4 個の引数を受け取ります。

1 個目　中心の x 座標。

2 個目　中心の y 座標。

3 個目　x 軸方向の半径。

4 個目　y 軸方向の半径。

HTML 文書の例　ellipse.html

```
<!DOCTYPE html>
<html>
<head><title>ellipse</title></head>
<body>
<script type="text/javascript" src="raphael.js"></script>
<script type="text/javascript">
var paper = Raphael(0, 0, 400, 300);
paper.ellipse(200, 150, 180, 130);
</script>
</body>
</html>
```

10.1.5 HTML の要素のキャンバス化

Raphael 関数には、次のような 3 個の引数を渡すこともできます。

1 個目　HTML の要素の名前（ id 属性の値)。

2 個目　横の長さ。

3 個目　縦の長さ。

この場合は、1 個目の引数として渡した名前を持つ HTML の要素がキャンバスになります。

HTML 文書の例　element.html

```
<!DOCTYPE html>
<html>
<head>
<title>element</title>
<style type="text/css">
#div1 {
    color: #f00;
    background-color: #9ff;
}
#div2 {
    color: #00f;
    background-color: #ff9;
}
```

```
</style>
</head>
<body>
<div id="div1"></div><div id="div2"></div>
<script type="text/javascript" src="raphael.js"></script>
<script type="text/javascript">
var paper1 = Raphael("div1", 200, 100);
paper1.ellipse(100, 50, 90, 40);
var paper2 = Raphael("div2", 100, 200);
paper2.ellipse(50, 100, 40, 90);
</script>
</body>
</html>
```

10.1.6 属性の設定

ellipse のような、形状を描画するメソッドは、SVG の要素をあらわすオブジェクトを生成して、それを戻り値として返します。

SVG の要素をあらわすオブジェクトは、attr というメソッドを持っています。このメソッドは、SVG の要素が持っている属性に対して属性値を設定します。

attr に渡す引数は、オブジェクトです。attr は、引数として受け取ったオブジェクトのプロパティーを属性名、そのプロパティーの値を属性値とみなして、SVG の要素の属性に属性値を設定します（マイナス (-) を含んでいる属性名は識別子にすることができませんので、文字列リテラルを書く必要があります)。たとえば、

```
paper.ellipse(200, 150, 180, 130).attr({
    "stroke-width": 30,
    stroke: "forestgreen",
    fill: "palegreen"
});
```

と書くことによって、線の幅が 30 で、線の色が forestgreen で、塗りつぶしの色が palegreen の楕円を描画することができます。

attr は、SVG の要素をあらわすオブジェクトを戻り値として返しますので、メソッドチェーンをうしろにつなぐことが可能です。

HTML 文書の例　attr.html

```
<!DOCTYPE html>
<html>
<head><title>attr</title></head>
<body>
<script type="text/javascript" src="raphael.js"></script>
<script type="text/javascript">
var paper = Raphael(0, 0, 400, 300);
var ellipse1 = paper.ellipse(160, 100, 140, 80);
var ellipse2 = paper.ellipse(240, 200, 140, 80);
ellipse1.attr({
    "stroke-width": 10,
    stroke: "navy",
    fill: "palegreen"
});
ellipse2.attr({
    "stroke-width": 20,
    stroke: "maroon",
    fill: "none"
});
</script>
</body>
</html>
```

attr は、引数として属性名（文字列）が渡された場合は、その属性の属性値を戻り値として返します。また、引数として属性名の配列が渡された場合は、それらの属性の属性値から構成される配列を戻り値として返します。

10.1.7 集合オブジェクト

Raphaël では、形状のグループを作りたいときは、「集合オブジェクト」(set object) と呼ばれるオブジェクトを使います。

集合オブジェクトは、キャンバスオブジェクトが持っている set というメソッドを呼び出すことによって作ることができます。そして、集合オブジェクトに形状を追加したいときは、その集合オブジェクトが持っている push というメソッドを呼び出して、追加したい形状を引数として渡します。引数は、何個でも好きなだけ渡すことができます。

集合オブジェクトも、SVG の要素をあらわすオブジェクトと同じように、attr というメソッドを持っています。このメソッドを使うことによって、集合オブジェクトを構成しているすべての形状の属性に対して、同じ属性値を設定することができます。

HTML 文書の例 set.html

```
<!DOCTYPE html>
<html>
<head><title>set</title></head>
<body>
<script type="text/javascript" src="raphael.js"></script>
<script type="text/javascript">
var paper = Raphael(0, 0, 400, 300);
var st = paper.set();
st.push(
    paper.ellipse(160, 100, 140, 80).attr({ fill: "coral" }),
    paper.ellipse(240, 200, 140, 80).attr({ fill: "khaki" })
).attr({
    "stroke-width": 10,
    stroke: "crimson",
});
</script>
</body>
</html>
```

10.2 形状の描画

10.2.1 この節について

第 10.1 節で説明したように、Raphaël では、キャンバスオブジェクトが持っているメソッドを呼び出すことによって、形状を描画することができます。

この節では、さまざまな形状について、それを描画するメソッドを紹介したいと思います。ただし、楕円を描画するメソッドについては、すでに第 10.1 節で紹介していますので、そちらを参照してください。

10.2.2 長方形

長方形は、rect というメソッドを呼び出すことによって描画することができます。このメソッドは、次の 4 個の引数を受け取ります。

1 個目　左上の頂点の x 座標。

2 個目　左上の頂点の y 座標。

3 個目　横の長さ。

4 個目　縦の長さ。

角を丸くする円の半径を、5 個目の引数として渡すことによって、角の丸い長方形を描画することもできます。

HTML 文書の例 rect.html

```
<!DOCTYPE html>
<html>
<head><title>rect</title></head>
<body>
<script type="text/javascript" src="raphael.js"></script>
<script type="text/javascript">
var paper = Raphael(0, 0, 400, 300);
```

```
paper.rect(20, 20, 280, 230).attr({ fill: "skyblue" });
paper.rect(100, 50, 280, 230, 40).attr({ fill: "navy" });
</script>
</body>
</html>
```

10.2.3 円

円は、circle というメソッドを呼び出すことによって描画することができます。このメソッドは、次の 3 個の引数を受け取ります。

1 個目　中心の x 座標。

2 個目　中心の y 座標。

3 個目　半径。

HTML 文書の例　circle.html

```
<!DOCTYPE html>
<html>
<head><title>circle</title></head>
<body>
<script type="text/javascript" src="raphael.js"></script>
<script type="text/javascript">
var paper = Raphael(0, 0, 400, 300);
paper.circle(200, 150, 140).attr({ fill: "lawngreen" });
</script>
</body>
</html>
```

10.2.4 パス

パスは、path というメソッドを呼び出すことによって描画することができます。このメソッドは、引数としてパスデータを受け取ります。パスデータの書き方は、SVG と同じです。

Raphaël は、line、polyline、polygon というメソッドを定義していません。ですから、Raphaël で直線を描画したいときは、path メソッドを使うことになります。

破線を作るための stroke-dasharray という属性に設定する値は、SVG と Raphaël とで書き方が違いますので、注意が必要です。Raphaël では、マイナス (-) とドット (.) という 2 種類の文字を並べた文字列で破線の形状を指定します。たとえば、 -.. という文字列を設定することによって、二点鎖線を作ることができます。

HTML 文書の例　path.html

```
<!DOCTYPE html>
<html>
<head><title>path</title></head>
<body>
<script type="text/javascript" src="raphael.js"></script>
<script type="text/javascript">
var data = "l80,0 a80,60 0 1,1 100,0 l80,0";
var paper = Raphael(0, 0, 400, 300);
paper.path("M50,150" + data).attr({
    "stroke-width": 30,
    stroke: "slateblue",
    "stroke-linecap": "round",
    "stroke-linejoin": "round"
});
paper.path("M50,280" + data).attr({
    "stroke-width": 5,
    stroke: "indigo",
    "stroke-dasharray": "-.."
});
</script>
</body>
</html>
```

path 要素のオブジェクトを生成したのちに、その要素が持っているパスデータを変更する、ということも可能です。パスデータを変更したいときは、path 要素のオブジェクトが持っている path という属性に対して新しいパスデータを設定します。

引数を何も渡さないで path を呼び出すと、空のパスを描画する path 要素が生成されます。

HTML 文書の例 pathbyattr.html

```
<!DOCTYPE html>
<html>
<head><title>path by attribute</title></head>
<body>
<script type="text/javascript" src="raphael.js"></script>
<script type="text/javascript">
var data = "M50,60 l120,30 a100,100 0 1,1 0,120 l-120,30 Z";
var paper = Raphael(0, 0, 400, 300);
paper.path().attr({
    path: data,
    fill: "forestgreen"
});
</script>
</body>
</html>
```

10.2.5 テキスト

テキストは、text というメソッドを呼び出すことによって描画することができます。このメソッドは、次の 3 個の引数を受け取ります。

1 個目　テキストの左端の x 座標。

2 個目　テキストのベースラインの y 座標。

3 個目　描画するテキスト。

HTML 文書の例 text.html

```
<!DOCTYPE html>
<html>
<head><title>text</title></head>
<body>
<script type="text/javascript" src="raphael.js"></script>
<script type="text/javascript">
var paper = Raphael(0, 0, 400, 300);
paper.set().push(
    paper.text(200, 70, "serif")
         .attr({ "font-family": "serif" }),
    paper.text(200, 120, "sans-serif")
         .attr({ "font-family": "sans-serif" }),
    paper.text(200, 170, "serif italic").attr({
        "font-family": "serif",
        "font-style": "italic"
    }),
    paper.text(200, 220, "serif bold").attr({
        "font-family": "serif",
        "font-weight": "bold"
    }),
    paper.text(200, 270, "serif italic bold").attr({
        "font-family": "serif",
        "font-style": "italic",
        "font-weight": "bold"
    })
).attr({
    fill: "chocolate",
    "font-size": 50,
    "text-anchor": "middle"
});
</script>
</body>
</html>
```

10.3 イベント処理

10.3.1 Raphaël でのイベント処理の基礎

Raphaël では、SVG の要素をあらわすオブジェクトに対して、イベントが発生したときに呼び出される関数を設定することによって、イベント処理を実装することができます。

SVG の要素をあらわすオブジェクトは、マウスによって発生するイベントの種類ごとに、そのイベントが発生したときに呼び出される関数を設定する、次のようなメソッドを持っています。

`mouseover`	グラフィックスの上に重なった。
`mouseout`	グラフィックスの上から出ていった。
`mousedown`	グラフィックスの上でボタンが押された。
`mouseup`	グラフィックスの上でボタンが離された。
`mousemove`	グラフィックスの上で移動した。
`click`	グラフィックスの上でクリックされた。
`dblclick`	グラフィックスの上でダブルクリックされた。

イベントが発生したときに呼び出される関数は、発生したイベントをあらわすイベントオブジェクトを引数として受け取ります。そして、SVG の要素をあらわすオブジェクトを戻り値として返します。

イベントが発生したときに呼び出される関数の中では、イベントが発生した SVG の要素をあらわすオブジェクトを、`this` で求めることができます。

HTML 文書の例 `mouse.html`

```
<!DOCTYPE html>
<html>
<head><title>mouse</title></head>
<body>
<script type="text/javascript" src="raphael.js"></script>
<script type="text/javascript">
var paper = Raphael(0, 0, 400, 300);
paper.ellipse(200, 150, 180, 130).attr({
    fill: "firebrick"
}).mouseover(function(event) {
    this.attr({ fill: "deeppink" });
}).mouseout(function(event) {
    this.attr({ fill: "firebrick" });
});
</script>
</body>
</html>
```

10.3.2 イベントオブジェクト

先ほど説明したように、イベントが発生したときに呼び出される関数は、発生したイベントをあらわすイベントオブジェクトを引数として受け取ります。

マウスによるイベントが発生したときのマウスポインターの座標は、イベントオブジェクトが持っている `offsetX` と `offsetY` というプロパティーに設定されます。

HTML 文書の例 `position.html`

```
<!DOCTYPE html>
<html>
<head><title>position</title></head>
<body>
<script type="text/javascript" src="raphael.js"></script>
<script type="text/javascript">
var paper = Raphael(0, 0, 400, 300);
paper.rect(0, 0, 400, 300).attr({ fill: "lemonchiffon" })
     .click(function(event) {
         paper.circle(event.offsetX, event.offsetY, 10)
              .attr({ fill: "crimson" });
```

```
    });
</script>
</body>
</html>
```

10.4 アニメーション

10.4.1 Raphaël でのアニメーションの基礎

Raphaël では、SVG の要素のオブジェクトが持っている animate というメソッドを使うことによって、その SVG の要素の属性を変化させるアニメーションを作ることができます。このメソッドは、次の 2 個の引数を受け取ります。

1 個目　アニメーションが終了した時点での属性を指定するオブジェクト（プロパティーが属性名、そのプロパティーの値が属性値になる）。

2 個目　アニメーションが始まってから終わるまでの時間（単位はミリ秒）。

HTML 文書の例　animate.html

```
<!DOCTYPE html>
<html>
<head><title>animate</title></head>
<body>
<script type="text/javascript" src="raphael.js"></script>
<script type="text/javascript">
var paper = Raphael(0, 0, 400, 300);
paper.circle(20, 150, 10).attr({ fill: "blue" })
     .animate({
         cx: 280,
         r: 100,
         fill: "white"
     }, 4000);
</script>
</body>
</html>
```

10.4.2 アニメーション終了後の処理

animate メソッドには、3 個目の引数として関数を渡すことも可能です。3 個目の引数として渡された関数は、アニメーションが終了した時点で呼び出されます。ですから、別のアニメーションを実行する関数を 3 個目の引数として animate に渡すことによって、さらに複雑なアニメーションを作ることができます。

HTML 文書の例　repeat.html

```
<!DOCTYPE html>
<html>
<head><title>repeat</title></head>
<body>
<script type="text/javascript" src="raphael.js"></script>
<script type="text/javascript">
var paper = Raphael(0, 0, 400, 300);
var c = paper.circle(200, 150, 10).attr({ fill: "lime" });
inflate();

function inflate() {
    c.animate({ r: 130 }, 4000, deflate);
}

function deflate() {
    c.animate({ r: 10 }, 4000, inflate);
}
</script>
</body>
</html>
```

10.4.3 パスに沿った移動のアニメーション

SVG の要素のオブジェクトが持っている animateAlong というメソッドを呼び出すことによって、その SVG の要素をパスに沿って移動させるアニメーションを作ることができます。このメソッドは、次の 2 個の引数を受け取ります。

1 個目　グラフィックスをそれに沿って移動させるパスのパスデータ。

2 個目　アニメーションが始まってから終わるまでの時間（単位はミリ秒）。

animate メソッドと同じように、 animateAlong メソッドにも、3 個目の引数として関数を渡すことができて、その関数は、アニメーションが終了した時点で呼び出されます。

HTML 文書の例　along.html

```
<!DOCTYPE html>
<html>
<head><title>along</title></head>
<body>
<script type="text/javascript" src="raphael.js"></script>
<script type="text/javascript">
var paper = Raphael(0, 0, 400, 300);
var data = "M150,30 l80,0 a120,120 0 1,1 0,240 " +
           "l-80,0 a120,120 0 1,1 0,-240 Z";
paper.path(data);
var r = paper.rect(150, 30, 40, 15).attr({ fill: "indigo" });
goAround();

function goAround() {
    r.animateAlong(data, 10000, goAround);
}
</script>
</body>
</html>
```

10.4.4 アニメーションの追尾

Raphaël では、SVG の要素のオブジェクトが持っている onAnimation というメソッドを使うことによって、アニメーションを追尾して何らかの処理を実行する、ということができます。

SVG の要素のオブジェクトが持っている onAnimation メソッドを呼び出して、引数として関数を渡すと、その関数は、アニメーションが実行されているあいだ、連続的に呼び出されます。

HTML 文書の例　onanimation.html

```
<!DOCTYPE html>
<html>
<head><title>onAnimation</title></head>
<body>
<script type="text/javascript" src="raphael.js"></script>
<script type="text/javascript">
var paper = Raphael(0, 0, 400, 300);
var data = "M150,30 l80,0 a120,120 0 1,1 0,240 " +
           "l-80,0 a120,120 0 1,1 0,-240 Z";
paper.path(data);
var line = paper.path().attr({
    "stroke-width": 8,
    stroke: "orangered"
});
var r = paper.circle(150, 30, 10).attr({ fill: "limegreen" });
goAround();

function goAround() {
    r.animateAlong(data, 10000, goAround)
     .onAnimation(function() {
          line.attr({
              path: "M200,150 L" +
                    r.attr("cx") + "," + r.attr("cy")
          });
     });
```

```
}
</script>
</body>
</html>
```

10.5 プラグイン

10.5.1 Raphaël のプラグインの基礎

Raphaël の機能を拡張する JavaScript のスクリプトは、Raphaël の「プラグイン」(plugin) と呼ばれます。

Raphaël のプラグインは、キャンバスオブジェクトまたは SVG の要素をあらわすオブジェクトにメソッドを追加することによって作ることができます。

この節では、キャンバスオブジェクトにメソッドを追加する方法と、SVG の要素をあらわすオブジェクトにメソッドを追加する方法について説明したいと思います。

10.5.2 キャンバスオブジェクトにメソッドを追加する方法

キャンバスオブジェクトにメソッドを追加したいときは、`Raphael.fn` という名前のオブジェクトにプロパティーを追加して、そのプロパティーに関数を設定します。そうすると、その関数が、キャンバスオブジェクトにメソッドとして追加されます。メソッドの名前は、追加したプロパティーの名前と同じです。たとえば、

```
Raphael.fn.namako = function { ... }
```

と書くことによって、`namako` という名前のメソッドがキャンバスオブジェクトに追加されます。

プラグインの例 `packman.js`

```
Raphael.fn.packman = function(cx, cy, r) {
    var dx = r * Math.cos(Math.PI/6);
    var dy = r * Math.sin(Math.PI/6);
    return paper.path(
        "M" + cx + "," + cy + " l" + dx + "," + -dy +
        " a" + r + "," + r + " 0 1,0 0," + dy*2 + " Z");
}
```

このプラグインは、パックマンを描画する `packman` というメソッドをキャンバスオブジェクトに追加します。このメソッドは、次の 3 個の引数を受け取ります。

1 個目　中心の x 座標。

2 個目　中心の y 座標。

3 個目　半径。

HTML 文書の例 `packman.html`

```
<!DOCTYPE html>
<html>
<head><title>packman</title></head>
<body>
<script type="text/javascript" src="raphael.js"></script>
<script type="text/javascript" src="packman.js"></script>
<script type="text/javascript">
var paper = Raphael(0, 0, 400, 300);
paper.packman(200, 150, 120).attr({ fill: "yellow" });
</script>
</body>
</html>
```

10.5.3 SVG の要素をあらわすオブジェクトにメソッドを追加する方法

SVG の要素をあらわすオブジェクトにメソッドを追加したいときは、`Raphael.el` という名前のオブジェクトにプロパティーを追加して、そのプロパティーに関数を設定します。そうすると、その関数が、SVG の要素をあらわすオブジェクトにメソッドとして追加されます。メソッドの名前は、追加したプロパティーの名前と同じです。たとえば、

```
Raphael.el.namako = function { ... }
```

と書くことによって、namako という名前のメソッドが、SVG の要素をあらわすオブジェクトに追加されます。

プラグインの例　href.js

```
Raphael.el.href = function(url) {
    this.attr({ href: url });
    return this;
}
```

このプラグインは、SVG の要素をアンカーにする href というメソッドを、SVG の要素をあらわすオブジェクトに追加します。このメソッドは、アンカーに設定する URI を引数として受け取ります。

このプラグインから分かるとおり、Raphaël では、attr メソッドを使って href という属性に URI を設定することによって、SVG の要素をアンカーにすることができます。

HTML 文書の例　href.html

```
<!DOCTYPE html>
<html>
<head><title>href</title></head>
<body>
<script type="text/javascript" src="raphael.js"></script>
<script type="text/javascript" src="href.js"></script>
<script type="text/javascript">
var paper = Raphael(0, 0, 400, 300);
paper.circle(200, 150, 120)
     .href("http://www.example.org/")
     .attr({ fill: "dodgerblue" });
</script>
</body>
</html>
```

付録 A　応用的な SVG 文書

A.1　アナログ時計

A.1.1　この付録について

このチュートリアルの本文で紹介されている SVG 文書は、特定の機能について理解してもらうことを目的として書かれたものですので、あまり意味のないものが多かったわけですが、この付録では、それらよりも少しだけ応用的な SVG 文書をいくつか紹介したいと思います。

A.1.2　アナログ時計の SVG 文書

応用的な SVG 文書としてまず最初に紹介するのは、アナログ時計 (analog clock) の SVG 文書です。

SVG 文書の例　clock.svg

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="400px" height="400px" viewBox="0 0 400 400"
    xmlns="http://www.w3.org/2000/svg">
  <g font-size="48" font-family="serif" fill="navy"
     text-anchor="middle">
    <text x="268" y="94">1</text>
    <text x="320" y="144">2</text>
    <text x="344" y="218">3</text>
    <text x="320" y="288">4</text>
    <text x="268" y="338">5</text>
    <text x="200" y="355">6</text>
    <text x="134" y="338">7</text>
    <text x="80" y="288">8</text>
```

```
    <text x="58" y="218">9</text>
    <text x="88" y="150">10</text>
    <text x="136" y="100">11</text>
    <text x="200" y="78">12</text>
  </g>
  <path id="hour" fill="forestgreen"
      d="M200,80 l15,90 l-11,0 l0,10
                 a20,20 0 1,1 -8,0
                 l0,-10 l-11,0 Z"/>
  <path id="minute" fill="mediumblue"
      d="M200,40 l10,135 l-8,0 l0,10
                 a15,15 0 1,1 -4,0
                 l0,-10 l-8,0 Z"/>
  <g id="second" stroke="crimson" fill="crimson">
    <line x1="200" y1="44" x2="200" y2="240"
         stroke-width="3"/>
    <circle cx="200" cy="200" r="10"/>
  </g>
  <script type="text/ecmascript"><![CDATA[
    var SVG = "http://www.w3.org/2000/svg";
    var rootobj = document.documentElement;
    drawScale();
    var hour = new Hour();
    var minute = new Minute();
    var second = new Second();
    var last = 0;
    setInterval(function() {
        var now = new Date();
        var h = now.getHours();
        var m = now.getMinutes();
        var s = now.getSeconds();
        if (s != last) {
            hour.rotate(h, m, s);
            minute.rotate(m, s);
            second.rotate(s);
            last = s;
        }
    }, 100);

    function drawLine(length, width, angle) {
        var y1 = 10;
        var lineobj = document.createElementNS(SVG, "line");
        lineobj.setAttributeNS(null, "x1", 200);
        lineobj.setAttributeNS(null, "y1", y1);
        lineobj.setAttributeNS(null, "x2", 200);
        lineobj.setAttributeNS(null, "y2", length-y1);
        lineobj.setAttributeNS(null, "stroke-width", width);
        lineobj.setAttributeNS(null, "stroke", "navy");
        lineobj.setAttributeNS(null,
            "transform", "rotate(" + angle + ", 200, 200)");
        rootobj.appendChild(lineobj);
    }

    function drawScale() {
        for (var theta = 6; theta <= 360; theta+=6)
            if (theta % 30 == 0)
                drawLine(50, 4, theta);
            else
                drawLine(40, 3, theta);
    }

    function Hour() {
        this.element = document.getElementById("hour");

        this.rotate = function(h, m, s) {
            var angle = h * 30 + m * 0.5 + s * 0.008;
            attribute = "rotate(" + angle + ", 200, 200)";
```

```
            this.element.setAttributeNS(null,
                "transform", attribute);
        };
    }

    function Minute() {
        this.element = document.getElementById("minute");

        this.rotate = function(m, s) {
            var angle = m * 6 + s * 0.1;
            attribute = "rotate(" + angle + ", 200, 200)";
            this.element.setAttributeNS(null,
                "transform", attribute);
        };
    }

    function Second() {
        this.element = document.getElementById("second");

        this.rotate = function(s) {
            var angle = s * 6;
            attribute = "rotate(" + angle + ", 200, 200)";
            this.element.setAttributeNS(null,
                "transform", attribute);
        };
    }
  ]]></script>
</svg>
```

A.2 スロットマシン

A.2.1 スロットマシンとは何か

この節では、「スロットマシン」と呼ばれるギャンブル機械の SVG 文書を紹介したいと思います。

「スロットマシン」(slot machine) という言葉は、もともとはコインを投入することによって作動するギャンブル機械の総称でしたが、現在では、「リールマシン」と呼ばれるギャンブル機械を指して使われることが多いようです。

「リールマシン」(reel machine) というのは、絵柄が描かれた複数個の円筒を回転させて、その回転が停止したときの絵柄の並び方によって配当を決定する、というギャンブル機械のことです。絵柄が描かれたそれぞれの円筒は、「リール」(reel) と呼ばれます。リールマシンを発明したのは、アメリカの Charles Fey(1862–1944) という人です。

A.2.2 スロットマシンの SVG 文書

次の SVG 文書がスロットマシンなのですが、ただし、かなり単純化してあります。

SVG 文書の例 slot.svg

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="400px" height="300px" viewBox="0 0 400 300"
    xmlns="http://www.w3.org/2000/svg">
  <g font-family="serif" text-anchor="middle">
    <g font-size="128" fill="limegreen">
      <text id="reel1" x="80" y="120">0</text>
      <text id="reel2" x="200" y="120">0</text>
      <text id="reel3" x="320" y="120">0</text>
    </g>
    <g font-size="24">
      <g fill="darkorange">
        <rect x="50" y="140" width="60" height="30"
            onclick="reel1.stop()"/>
        <rect x="170" y="140" width="60" height="30"
```

```
            onclick="reel2.stop()"/>
        <rect x="290" y="140" width="60" height="30"
            onclick="reel3.stop()"/>
        <text x="80" y="190">stop</text>
        <text x="200" y="190">stop</text>
        <text x="320" y="190">stop</text>
      </g>
      <g fill="cornflowerblue">
        <rect x="170" y="220" width="60" height="30"
            onclick="start()"/>
        <text x="200" y="270">start</text>
      </g>
    </g>
    <text id="message" x="200" y="100" font-size="32"
        fill="midnightblue">
    </text>
  </g>
  <script type="text/ecmascript"><![CDATA[
    var messageobj =
        document.getElementById("message").firstChild;
    var reel1 = new Reel("reel1");
    var reel2 = new Reel("reel2");
    var reel3 = new Reel("reel3");
    var status = 0;
    setInterval(function() {
        reel1.increment();
        reel2.increment();
        reel3.increment();
    }, 100);

    function Reel(reelid) {
        this.reelobj =
            document.getElementById(reelid).firstChild;
        this.n = 1;
        this.moving = false;

        this.increment = function() {
            if (this.moving) {
                this.reelobj.data = this.n;
                if (this.n <= 8)
                    this.n++;
                else
                    this.n = 1;
            }
        };

        this.start = function() {
            this.moving = true;
        };

        this.stop = function() {
            this.moving = false;
            status--;
            if (status == 0)
                determine();
        };
    }

    function start() {
        status = 3;
        messageobj.data = "";
        reel1.start();
        reel2.start();
        reel3.start();
    }

    function determine() {
```

```
        if (reel1.n == reel2.n && reel2.n == reel3.n)
            messageobj.data = "Congratulations!";
        else if (reel1.n == reel2.n || reel2.n == reel3.n ||
                 reel3.n ==  reel1.n)
            messageobj.data = "You are looking good.";
        else
            messageobj.data = "It's really too bad.";
    }
  ]]></script>
</svg>
```

遊び方は次のとおりです。

(1) スタートのボタンをクリックすると、リールの回転が開始されます。
(2) それぞれのリールの下にあるストップのボタンをクリックすると、そのリールの回転が停止します。
(3) 三つのリールがすべて停止すると、リールの数字がすべて同じ、二つだけ同じ、まったく異なる、という三通りの結果に応じたメッセージが表示されます。

A.3 15 パズル

A.3.1 15 パズルとは何か

この節では、「15 パズル」と呼ばれるパズルの SVG 文書を紹介したいと思います。

15 パズルは、「スライディングブロックパズル」と呼ばれるパズルの一種です。「スライディングブロックパズル」(sliding block puzzle) というのは、盤面の上で駒をスライドさせていって、特定の駒の配置を作ることを目的とするパズルのことです。

「15 パズル」(15 puzzle) は、4×4 の升目を持つ盤面と、1 から 15 までの数字が書かれた 15 個の駒を使うスライディングブロックパズルです。駒は、初期状態では、

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

という配置になっています。そして、1 箇所だけ空いている升目を利用することによって駒をスライドさせて、目的の配置を作ります。目的となる配置は一つではなく、たとえば、

1	2	3	4
12	13	14	5
11		15	6
10	9	8	7

という渦巻き型などがあります。

A.3.2 15 パズルの SVG 文書

次の SVG 文書は、15 パズルで遊ぶことのできるものです。

SVG 文書の例　15puzzle.svg

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="400px" height="400px" viewBox="0 0 400 400"
    xmlns="http://www.w3.org/2000/svg">
  <script type="text/ecmascript"><![CDATA[
    var SVG = "http://www.w3.org/2000/svg";
    var rsize = 80;
    var fsize = 64;
    var margin = 40;
    var rootobj = document.documentElement;
    var board = new Board();
```

```
function Board() {
    this.board = new Array(16);
    for (var i = 1; i <= 15; i++) {
        this.board[i] = new Piece(i);
        this.board[i].move(Math.floor((i-1)/4), (i-1)%4);
    }
    this.nulli = 3;
    this.nullj = 3;

    this.move = function(n) {
        var piece = this.board[n];
        var i = piece.i;
        var j = piece.j;
        var ni = this.nulli;
        var nj = this.nullj;
        if (i == ni-1 && j == nj) { // down
            piece.move(i+1, j);
            this.movenull(ni-1, nj);
        } else if (i == ni && j == nj+1) { // to right
            piece.move(i, j-1);
            this.movenull(ni, nj+1);
        } else if (i == ni+1 && j == nj) { // up
            piece.move(i-1, j);
            this.movenull(ni+1, nj);
        } else if (i == ni && j == nj-1) { // to left
            piece.move(i, j+1);
            this.movenull(ni, nj-1);
        }
    };

    this.movenull = function(i, j) {
        this.nulli = i;
        this.nullj = j;
    };
}

function Piece(n) {
    this.pieceobj = document.createElementNS(SVG, "rect");
    this.pieceobj.setAttributeNS(null, "width", rsize);
    this.pieceobj.setAttributeNS(null, "height", rsize);
    this.pieceobj.setAttributeNS(null, "rx", rsize*0.2);
    this.pieceobj.setAttributeNS(null, "stroke-width",
        rsize*0.05);
    this.pieceobj.setAttributeNS(null, "stroke", "white");
    this.pieceobj.setAttributeNS(null, "fill", "green");
    rootobj.appendChild(this.pieceobj);
    this.numobj = document.createTextNode(n);
    this.textobj = document.createElementNS(SVG, "text");
    this.textobj.appendChild(this.numobj);
    this.textobj.setAttributeNS(null, "font-family",
        "serif");
    this.textobj.setAttributeNS(null, "font-size", fsize);
    this.textobj.setAttributeNS(null, "text-anchor",
        "middle");
    this.textobj.setAttributeNS(null, "fill", "white");
    rootobj.appendChild(this.textobj);
    this.coverobj = document.createElementNS(SVG, "rect");
    this.coverobj.setAttributeNS(null, "id", "no" + n);
    this.coverobj.setAttributeNS(null, "width", rsize);
    this.coverobj.setAttributeNS(null, "height", rsize);
    this.coverobj.setAttributeNS(null, "opacity", "0");
    this.coverobj.setAttributeNS(null, "onclick",
        "move(evt)");
    rootobj.appendChild(this.coverobj);

    this.move = function(i, j) {
```

```
            this.i = i;
            this.j = j;
            var x = margin + rsize*j;
            var y = margin + rsize*i;
            var tx = x + rsize*0.5;
            var ty = y + rsize*0.75;
            this.pieceobj.setAttributeNS(null, "x", x);
            this.pieceobj.setAttributeNS(null, "y", y);
            this.textobj.setAttributeNS(null, "x", tx);
            this.textobj.setAttributeNS(null, "y", ty);
            this.coverobj.setAttributeNS(null, "x", x);
            this.coverobj.setAttributeNS(null, "y", y);
        };
    }

    function move(evt) {
        var id = evt.target.getAttributeNS(null, "id");
        var num = id.substr(2) - 0;
        board.move(num);
    }
  ]]></script>
</svg>
```

空いている升目の隣にある駒をクリックすると、その駒が、空いている升目にスライドします。

A.4 テニス

A.4.1 当たり判定とは何か

この節では、ラケットを動かすことによってボールを打ち返すことのできる、テニスのようなゲームをプレイすることのできる SVG 文書を紹介したいと思います。

画面上に表示された物体が衝突したときに何らかの処理が実行されるようなゲームを作るためには、「当たり判定」と呼ばれる技術が必要になります。「当たり判定」(collision detection) というのは、ゲームの中に登場する二つの物体が、空間的に接触しているかそれとも離れているかということを判定することです。

A.4.2 線分の当たり判定

まずは、1 次元空間での当たり判定について考えてみましょう。

1 次元空間では、すべての物体は線分という形状を持つことになります。線分は、位置と長さであらわされます。

引数として二つの線分の位置（左端）と長さを受け取って、それらの当たり判定をする関数は、次のように定義することができます。

```
function overlap(a, alen, b, blen) {
    if (a < b)
        return b <= a + alen;
    else
        return a <= b + blen;
}
```

A.4.3 長方形の当たり判定

次に、2 次元空間での当たり判定について考えてみましょう。

2 次元空間の物体はさまざまな形状を持っているわけですが、ここでは説明を簡潔にするために、水平と垂直の辺を持つ長方形だけについて考えることにします。

長方形のオブジェクトを生成するコンストラクタは、次のように定義することができます。

```
function Rectangle(x, y, width, height) {
    this.x = x;
    this.y = y;
    this.width = width;
    this.height = height;
    ...
```

x と y は長方形の左上の頂点の x 座標と y 座標で、width は横の長さ、height は縦の長さです。

二つの長方形の当たり判定は、x 軸方向と y 軸方向のそれぞれについて 1 次元の当たり判定をして、それらの論理積を求めることによって実現することができます。つまり、次のようなメソッドを定義すればいいわけです。

```
this.collision = function(rect) {
    return overlap(this.x, this.width,
                   rect.x, rect.width) &&
           overlap(this.y, this.height,
                   rect.y, rect.height);
};
```

A.4.4 テニスの SVG 文書

次の SVG 文書は、三つの辺を壁で囲まれたコートの中で、ラケットを動かしてボールを打ち返す、というゲームです。

SVG 文書の例 tennis.svg

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="300px" height="400px" viewBox="0 0 300 400"
    xmlns="http://www.w3.org/2000/svg">
  <rect x="0" y="0" width="300" height="400"
     fill="lightgreen"/>
  <g font-size="32" font-family="serif" text-anchor="middle">
    <text id="score" x="150" y="36" fill="coral">0</text>
    <text id="message" x="150" y="200"
        fill="darkgreen">click to start</text>
  </g>
  <circle id="ball" cx="10" cy="10" r="10" fill="brown"/>
  <rect id="racket" x="150" y="380" width="40" height="10"
     fill="steelblue"/>
  <rect x="0" y="0" width="300" height="400" opacity="0"
     onmousemove="racket.move(evt)" onclick="start()"/>
  <script type="text/ecmascript"><![CDATA[
    var scoreobj =
        document.getElementById("score").firstChild;
    var messageobj =
        document.getElementById("message").firstChild;
    var courtrect = new Rectangle(0, 0, 300, 400);
    var racket = new Racket("racket", 150, 380, 40, 10);
    var ball = new Ball("ball", 10, 10, 5);
    var score = 0;
    var interval = null;

    function overlap(a, alen, b, blen) {
        if (a < b)
            return b <= a + alen;
        else
            return a <= b + blen;
    }

    function Rectangle(x, y, width, height) {
        this.x = x;
        this.y = y;
        this.width = width;
        this.height = height;

        this.outside = function(rect) {
            if (this.x < rect.x)
                return "west";
            else if (this.x + this.width >
                     rect.x + rect.width)
                return "east";
            else if (this.y < rect.y)
                return "north";
```

```
            else if (this.y + this.height >
                     rect.y + rect.height)
                return "south";
            else
                return "inside";
        };

        this.collision = function(rect) {
            return overlap(this.x, this.width,
                           rect.x, rect.width) &&
                   overlap(this.y, this.height,
                           rect.y, rect.height);
        };
    }

    function Ball(id, width, height, step) {
        this.element = document.getElementById(id);
        this.rect = new Rectangle(0, 0, width, height);
        this.step = step;
        this.direction = "sw";

        this.reset = function() {
            this.rect.x = 0;
            this.rect.y = 0;
            this.direction = "sw";
        };

        this.move = function(x, y) {
            this.rect.x += this.step * x;
            this.rect.y += this.step * y;
            this.element.setAttributeNS(null, "cx",
                this.rect.x + this.rect.width / 2);
            this.element.setAttributeNS(null, "cy",
                this.rect.y + this.rect.height / 2);
        };

        this.moveDirection = function() {
            switch (this.direction) {
            case "se":
                this.move(1, 1);
                break;
            case "sw":
                this.move(-1, 1);
                break;
            case "nw":
                this.move(-1, -1);
                break;
            case "ne":
                this.move(1, -1);
                break;
            }
        };

        this.turn = function() {
            switch (this.rect.outside(courtrect)) {
            case "west":
                if (this.direction == "sw")
                    this.direction = "se";
                else if (this.direction == "nw")
                    this.direction = "ne";
                break;
            case "east":
                if (this.direction == "se")
                    this.direction = "sw";
                else if (this.direction == "ne")
                    this.direction = "nw";
                break;
```

```
            case "north":
                if (this.direction == "ne")
                    this.direction = "se";
                else if (this.direction == "nw")
                    this.direction = "sw";
                break;
            }
        };

        this.hit = function() {
            if (this.direction == "se")
                this.direction = "ne";
            else if (this.direction == "sw")
                this.direction = "nw";
        };

        this.south = function() {
            return this.direction.substr(0, 1) == "s";
        };

        this.beyondSouth = function() {
            return this.rect.outside(courtrect) == "south";
        };
    }

    function Racket(id, x, y, width, height) {
        this.element = document.getElementById(id);
        this.rect = new Rectangle(x, y, width, height);

        this.move = function(evt) {
            this.rect.x = evt.offsetX - this.rect.width / 2;
            this.element.setAttributeNS(null, "x",
                this.rect.x);
        };
    }

    function hit() {
        ball.hit();
        scoreobj.data = ++score;
    }

    function gameOver() {
        messageobj.data = "click to start";
        clearInterval(interval);
        interval = null;
    }

    function start() {
        if (interval == null) {
            messageobj.data = "";
            scoreobj.data = "0";
            ball.reset();
            score = 0;
            interval = setInterval(function() {
                if (ball.beyondSouth())
                    gameOver();
                ball.turn();
                if (ball.south() &&
                        ball.rect.collision(racket.rect))
                    hit();
                ball.moveDirection();
            }, 10);
        }
    }
  ]]></script>
</svg>
```

コートをマウスでクリックすると、ゲームが開始されます。ラケットは、マウスを使って左右に移動させることができます。ラケットでボールを 1 回打ち返すごとに、得点に 1 点が加算されます。ボールがコートの下の辺まで到達すると、ゲームオーバーになります。

付録 B　JavaScript

B.1　基本的な文法

B.1.1　注釈

`//` から最初の改行までは注釈 (comment) とみなされます。また、`/*` で始まって `*/` で終わる文字列も、注釈とみなされます。後者の注釈は、改行を含んでいてもかまいません。

B.1.2　識別子

識別子 (identifier) は、英字、数字、アンダースコアを並べることによって作ることができます。ただし、識別子の先頭の文字を数字にすることはできません。

英字が大文字か小文字かという点だけが異なる二つの識別子も、異なるものとみなされます。

B.1.3　予約語

予約語 (reserved word) は、識別子として使うことができません。

予約語には、キーワード (keyword)、将来予約語 (future reserved word)、ヌルリテラル (null literal)、真偽値リテラル (Boolean literal)、という 4 種類のものがあります。ヌルリテラルと真偽値リテラルについては、第 B.2.5 項で説明します。

キーワードは次の単語です。

```
break       delete      if          this        while
case        do          in          throw       with
catch       else        instanceof  try
continue    finally     new         typeof
debugger    for         return      var
default     function    switch      void
```

将来予約語は次の単語です。

```
class       enum        extends     super
const       export      import
```

B.1.4　プログラム

プログラム (program) は、文 (statement) または関数宣言 (function declaration) の列です。文については第 B.3 節で説明します。

B.1.5　関数宣言

関数宣言 (function declaration) は、次のような構文を持つ記述です。

```
function 識別子 (仮引数リスト) { プログラム }
```

このような記述を書くことによって、中括弧の中に書かれたプログラムを実行する関数を生成して、その関数の名前として識別子を与えることができます。「仮引数リスト」のところに識別子をコンマで区切って並べると、それらの識別子が仮引数 (formal parameter) の名前になります。

B.2　式

B.2.1　式の基礎

評価する (evaluate) ことのできる文法上の単位は、「式」(expression) と呼ばれます。式を評価すると、何らかのデータが得られます。式を評価することによって得られたデータは、その式の「値」(value) と呼ばれます。

B.2.2 型

データには型 (type) があります（ただし、変数には型はありません）。型には、次の六つのものがあります。

未定義型	undefined type。「未定義値」(undefined) と呼ばれるひとつのデータのみから構成される型。初期化されていない変数は、未定義値を値として持つ。
ヌル型	null type。「ヌル」(null) と呼ばれるひとつのデータのみから構成される型。
真偽値型	Boolean type。「真」(true) と「偽」(false) と呼ばれる二つのデータから構成される型。
文字列型	string type。文字列 (string) から構成される型。
数値型	number type。数値 (number) から構成される型。
オブジェクト型	object type。オブジェクトから構成される型。関数 (function)、配列 (array)、正規表現オブジェクト (regular expression object)、例外オブジェクト (exception object) も、この型を持つ。オブジェクトについての詳細は第 B.4 節参照。

B.2.3 式の分類

式は、`this`、識別子 (identifier)、リテラル (literal)、式を演算子 (operator) で結合したもの、関数式 (function expression)、配列初期化子 (array initializer)、オブジェクト初期化子 (object initializer) などに分類することができます。オブジェクト初期化子については、第 B.4 節で説明します。

B.2.4 `this`

`this` の値は、それを評価した文脈において自分自身とみなされるオブジェクトです。

B.2.5 リテラル

リテラル (literal) には、次の 5 種類のものがあります。

ヌルリテラル	null literal。ヌル (null) を値とするリテラル。`null` と書く。
真偽値リテラル	Boolean literal。真偽値 (Boolean) を値とするリテラル。`true` と `false` の二つがあり、`true` の値は真、`false` の値は偽。
数値リテラル	numeric literal。数値 (number) を値とするリテラル。10 進数リテラルと 16 進整数リテラルの 2 種類がある。 10 進数リテラルは 10 進数で数値を記述したリテラル。たとえば、`8708`、`-663`、`9.02`、`-0.4`、`3e8`、`5e-6`、`0.1e7` など。 16 進整数リテラルは 16 進数で整数を記述したリテラルで、16 進数の左側に `0x` または `0X` を書いたもの。たとえば、`0x4f` など。
文字列リテラル	string literal。文字列 (string) を値とするリテラル。単一引用符 (') または二重引用符 (") で文字列を囲むことによって作られる。文字列リテラルの中では、次のエスケープシーケンス (escape sequence) を使うことができる。

`\b`	バックスペース	`\f`	改ページ
`\t`	水平タブ	`\'`	単一引用符
`\v`	垂直タブ	`\"`	二重引用符
`\n`	改行	`\\`	バックスラッシュ
`\r`	キャリッジリターン		

正規表現リテラル	regular expression literal。正規表現オブジェクト (regular expression object) を値とするリテラル。スラッシュ(`/`) で正規表現を囲むことによって作られる。末尾に正規表現フラグを書くこともできる。

B.2.6 演算子

演算子 (operator) には次のようなものがあります。

a + b	a と b の両方が数値ならば、それらを加算する。a または b のいずれかが文字列ならば、それらを連結する。

a - b	a から b を減算する。
a * b	a と b を乗算する。
a / b	a を b で除算する。
a % b	a を b で除算したあまりを求める。
- a	a の符号を反転させる。
a | b	a と b のビットごとの論理和を求める。
a & b	a と b のビットごとの論理積を求める。
a ^ b	a と b のビットごとの排他的論理和を求める。
~ a	a のビットごとの否定を求める（すべてのビットを反転させる）。
a << b	a を b ビットだけ左シフトする。
a >> b	a を b ビットだけ右シフトする。
a >>> b	a を b ビットだけ論理右シフトする。
a > b	a は b よりも大きい。
a < b	a は b よりも小さい。
a >= b	a は b よりも大きいかまたは等しい。
a <= b	a は b よりも小さいかまたは等しい。
a == b	a と b とは等しい。
a != b	a と b とは等しくない。
a === b	a と b とは同じ型でかつ等しい。
a !== b	a と b とは異なる型であるかまたは等しくない。
a instanceof b	a はコンストラクタ b によって初期化されたオブジェクトである。
a in b	a はオブジェクト b のプロパティーである。
a || b	a と b の論理和を求める。
a && b	a と b の論理積を求める。
! a	a の否定を求める（真偽値を反転させる）。
a ? b : c	a が真ならば b を評価して、そうでなければ c を評価する。
a = b	b を a に設定する。
a op= b	a op b を計算した結果を a に設定する。
a++	a をインクリメントする。値は計算前。
a--	a をデクリメントするする。値は計算前。
++a	a をインクリメントする。値は計算後。
--a	a をデクリメントする。値は計算後。
a, b	a を評価して、そののち b を評価して、b の値を式の値とする。
new a	生成式 a で新しいオブジェクトを生成する。
delete a	a を消滅させる。
void a	a を評価するけれども、その値ではなく、未定義値を式の値とする。
a[b]	配列 a の要素に添字 b でアクセスする。または、オブジェクト a のプロパティー b にアクセスする。
a.b	オブジェクト a のプロパティー b にアクセスする（b は識別子）。
(a)	a をグループ化する。
a(b, c, ···)	関数 a を呼び出して、b、c、··· を引数として渡す。

B.2.7 関数式

関数式 (function expression) は、次のような構文です。

```
function(仮引数リスト) { プログラム }
```

関数式を評価すると、中括弧の中に書かれたプログラムを実行する関数が生成されます。関数式の値は、生成された関数です。

B.2.8 配列初期化子

配列初期化子 (array initializer) は、次のような構文です。

```
[式，式，･･･]
```

配列初期化子を評価すると、その中に書かれたそれぞれの式の値を要素とする配列が生成されます。配列初期化子の値は、生成された配列です。

B.3 文

B.3.1 文の分類

文 (statement) には、次のような種類があります。

ブロック	式文	while 文	for-in 文	return 文	try 文
変数文	if 文	do-while 文	continue 文	with 文	
空文	switch 文	for 文	break 文	throw 文	

B.3.2 ブロック

ブロック (block) は、文の列を中括弧 ({}) で囲んだものです。ブロックの中のそれぞれの文は、先頭から順番に実行されます。

B.3.3 変数文

変数文 (variable statement) は、変数を宣言して初期値を設定します。次のような構文です。

```
var 識別子 = 式，識別子 = 式，･･･;
```

識別子を名前として持つ変数を作って、式の値を初期値としてそれに設定します。イコールと式を省略すると、変数には未定義値が設定されます。

B.3.4 空文

空文 (empty statement) は、何もしない文です。セミコロン (;) を書くだけです。

B.3.5 式文

式文 (expression statement) は、式を評価します。式の右側にセミコロンを書いたものが式文です。

B.3.6 if 文

if 文 (if statement) は、真偽値によって動作を選択します。次のような構文です。

```
if（式）文1 else 文2
```

式の値が真ならば 文$_1$ を実行して、そうでなければ 文$_2$ を実行します。

if 文は、 else 以下を省略して、

```
if（式）文
```

と書いてもかまいません。この構文の if 文は、式の値が偽だった場合は何も実行しません。

B.3.7 switch 文

switch 文 (switch statement) は、式の値によって動作を選択します。次のような構文です。

```
switch（式）case ブロック
```

case ブロックの中には、 case 節と呼ばれる、

```
case 式: 文の列 break;
```

という形のものをいくつか書きます。そうすると、 switch というキーワードの右側に書かれた式の値と、 case の右側に書かれた式の値とが一致した case 節について、その中の文の列が実行されます。

switch 文のブロックの末尾に、 default 節と呼ばれる、

`default:` 文の列

という形のものを書くこともできます。そうすると、値が一致する `case` 節が存在しなかった場合に、`default` 節の文の列が実行されます。

B.3.8 `while` 文

`while` 文 (`while` statement) は、次のような構文です。

`while` (式) 文

式を評価して文を実行する、ということを繰り返します。式の値が偽ならば終了します。

B.3.9 `do-while` 文

`do-while` 文 (`do-while` statement) は、次のような構文です。

`do` 文 `while` (式);

文を実行して式を評価する、ということを繰り返します。式の値が偽ならば終了します。

B.3.10 `for` 文

`for` 文 (`for` statement) は、次のような構文です。

`for` (式$_1$; 式$_2$; 式$_3$) 文

式$_1$ を最初に 1 回だけ評価したのち、式$_2$ の評価、文の実行、式$_3$ の評価を繰り返します。式$_2$ の値が偽ならば終了します。

`for` 文は、次のような構文で書くことも可能です。

`for` (変数文 式$_1$; 式$_2$) 文

変数文を最初に 1 回だけ実行したのち、式$_1$ の評価、文の実行、式$_2$ の評価を繰り返します。式$_1$ の値が偽ならば終了します。

B.3.11 `for-in` 文

`for-in` 文 (`for-in` statement) は、次のような構文です。

`for` (`var` 識別子 `in` 式) 文

最初に式を 1 回だけ評価して、その値として得られたオブジェクトのそれぞれのプロパティーについて、その値を変数（識別子が変数名になります）に設定して文を実行する、ということを繰り返します。

B.3.12 `continue` 文

`continue` 文 (`continue` statement) は、`continue;` と書きます。自分を含んでいるブロックの実行を終了させます。

B.3.13 `break` 文

`break` 文 (`break` statement) は、`break;` と書きます。自分を含んでいる `switch` 文、`while` 文、`for` 文などの実行を終了させます。

B.3.14 `return` 文

`return` 文 (`return` statement) は、次のような構文です。

`return` 式;

式を評価して、自分を含んでいる関数またはメソッドの実行を終了させて、式の値をその戻り値にします。

B.3.15 `with` 文

`with` 文 (`with` statement) は、次のような構文です。

`with` (式) 文

式を評価して、その値として得られたオブジェクトの文脈で文を実行します。

B.3.16 `throw` 文

`throw` 文 (`throw` statement) は、次のような構文です。

```
throw 式;
```

式を評価して、その値として得られた例外を投げます。例外というのは、`Error` というコンストラクタによって初期化されたオブジェクトのことです。s が文字列だとするとき、

```
throw new Error(s);
```

という文を実行することによって、`message` というプロパティーに s が設定された例外を生成して、それを投げることができます。

B.3.17 `try` 文

`try` 文 (`try` statement) は、例外を処理します。`try` 文のもっとも基本的な構文は次のようなものです。

```
try ブロック1 catch（識別子）ブロック2
```

ブロック$_1$ を実行して、その実行中に例外が投げられた場合、変数（識別子が変数名になります）にその例外を設定して ブロック$_2$ を実行します。

`try` 文には、次のような構文もあります。

```
try ブロック1 catch（識別子）ブロック2 finally ブロック3
```

この場合、ブロック$_3$ は、例外が投げられた場合も投げられなかった場合も、かならず実行されます。

次のように、`catch` の部分を省略することも可能です。

```
try ブロック1 finally ブロック2
```

B.4 オブジェクト

B.4.1 オブジェクトの基礎

オブジェクト (object) は、「組」(pair) と呼ばれるものが集まってできている集合です。

組は、「プロパティー」(property) と呼ばれるデータと「値」(value) と呼ばれるデータとを対応させたものです。プロパティーにすることができるのは文字列または数値で、値にすることができるのは任意のデータです。ひとつのオブジェクトの中にある個々の組は、プロパティーによって識別されます。

JavaScript で「オブジェクト」と呼ばれているものは、一般的には「連想配列」(associative array) と呼ばれるものです。同じように、「プロパティー」は一般的には「キー」(key) と呼ばれます。

B.4.2 オブジェクト初期化子

オブジェクトは、「オブジェクト初期化子」」(object initializer) と呼ばれる式によって生成することができます。オブジェクト初期化子は、次のような構文です。

```
{ プロパティー設定, ··· }
```

オブジェクト初期化子を評価すると、その中に書かれたそれぞれのプロパティー設定によって作られた組から構成されるオブジェクトが生成されて、そのオブジェクトが、オブジェクト初期化子の値になります。

プロパティー設定は、次のような構文です。

```
プロパティー名: 式
```

「プロパティー名」のところには、識別子、文字列リテラル、数値リテラルのいずれかを書きます。プロパティー名として識別子を書いた場合、それは、その識別子を値とする文字列リテラルと同じ意味だと解釈されます。たとえば、`tako` というプロパティー名は、`"tako"` と同じ意味だと解釈されます。

プロパティー設定は、プロパティー名から求められた文字列または数値をプロパティー、そして式を評価して得られた値を値とする組を作ります。たとえば、

```
{ namako: null, "<==": true, 8: "umiushi" }
```

というオブジェクト初期化子を評価すると、プロパティーが namako で値が null という組、プロパティーが <== で値が true という組、プロパティーが 8 で値が umiushi という組から構成されるオブジェクトが生成されて、そのオブジェクトがこのオブジェクト初期化子の値になります。

B.4.3 プロパティーからの値の取得

オブジェクト a のプロパティー b から値を取得したいときは、a[b] という形の式を書きます。たとえば、

```
{ x: 33, y: 55, z: 77 }
```

というオブジェクトが hitode という変数に設定されているとするとき、

```
hitode["y"]
```

という式を評価すると、55 という値が得られます。

プロパティー b が識別子の構文を満足している場合、a[b] という式は、a.b と書くこともできます。ですから、先ほどの式は、

```
hitode.y
```

と書くことも可能です。

B.4.4 プロパティーへの値の設定

オブジェクト a のプロパティー b に値 c を設定したいときは、a[b] = c という形の式を書きます。たとえば、

```
{ x: 33, y: 55, z: 77 }
```

というオブジェクトが hitode という変数に設定されているとするとき、

```
hitode["y"] = 99
```

という式を評価すると、hitode の値は、

```
{ x: 33, y: 99, z: 77 }
```

に変わります。

プロパティー b が識別子の構文を満足している場合、a[b] = c という式は、a.b = c と書くこともできます。

B.4.5 プロパティーの追加

オブジェクト a にプロパティー b が存在していないときに a[b] = c を評価すると、新しいプロパティーとして b が a に追加されて、c がその値になります。

B.4.6 プロパティーの削除

プロパティーを削除したいときは、演算子の delete を使います。たとえば、

```
delete hitode["z"]
```

という式を評価すると、hitode というオブジェクトが持っている z というプロパティーが削除されます。この式は、

```
delete hitode.z
```

と書くことも可能です。

B.4.7 すべてのプロパティーへのアクセス

オブジェクトが持っているすべてのプロパティーにアクセスしたいときは、for-in 文を使います。たとえば、

```
{ x: 3, y: 5, z: 7 }
```

というオブジェクトが tako という変数に設定されているとするとき、

```
for (var i in tako) tako[i] *= 2;
```

という for-in 文を実行すると、tako の値は、

```
{ x: 6, y: 10, z: 14 }
```

に変わります。

B.4.8 コンストラクタ

オブジェクトは、new という演算子を使うことによって生成することも可能です。new の右側には、通常、「コンストラクタ」(constructor) と呼ばれる関数を呼び出す式を書きます。

コンストラクタというのは、new によって生成されたオブジェクトを初期化するために使われる関数のことです。コンストラクタの名前は、通常、先頭を大文字にします。new は、自分が生成したオブジェクトを this にしてコンストラクタを呼び出します。たとえば、

```
function MyObject(arg) { this.myproperty = arg; }
```

という関数宣言で MyObject というコンストラクタを定義して、

```
new MyObject(307)
```

という式でオブジェクトを生成したとすると、プロパティーが myproperty で値が 307 という組が、そのオブジェクトの中に作られます。

B.4.9 メソッド

メソッド (method) というのは、関数が設定されたプロパティー、またはプロパティーに設定された関数のことです。たとえば、

```
function MyObject() {
    this.mymethod = function(a) { return a*100; };
}
```

というコンストラクタを定義して、

```
var myobject = new MyObject();
```

という変数文でオブジェクトを生成したとすると、そのオブジェクトは、mymethod という名前のメソッドを持つことになります。このメソッドは、

```
myobject.mymethod(53)
```

というような式で呼び出すことができます。

B.4.10 プロトタイプオブジェクト

関数は、prototype という名前のプロパティーを持っています。そのプロパティーに設定されているオブジェクトは、「プロトタイプオブジェクト」(prototype object) と呼ばれます。

prototype プロパティーには、暗黙のうちにデフォルトのプロトタイプオブジェクトが設定されていますが、デフォルトのものとは別のオブジェクトをプロトタイプオブジェクトとして設定してもかまいません。また、デフォルトのプロトタイプオブジェクトに対してプロパティーを追加する、ということも可能です。

デフォルトのプロトタイプオブジェクトは、constructor というプロパティーを持っていて、そこには関数自身が設定されています。

コンストラクタによって初期化されたオブジェクトは、そのコンストラクタが持っているプロトタイプオブジェクトのプロパティーに対して、あたかもそれが自分自身のプロパティーであるかのようにアクセスすることができます。たとえば、

```
function MyObject() {}
```

という関数宣言で MyObject というコンストラクタを定義して、

```
MyObject.prototype.who = function() { alert("MyObject"); };
```

という式文で、MyObject のプロトタイプオブジェクトに who というメソッドを追加して、

```
var myobject = new MyObject();
```

という変数文で、MyObject で初期化されたオブジェクトを myobject という変数に設定したとします。このとき、who は、

```
myobject.who();
```

というように、あたかも、myobject に設定されているオブジェクト自身が持っているメソッドであるかのように呼び出すことができます。

ひとつのオブジェクトの中に、自分自身のプロパティーと、プロトタイプオブジェクトのプロパティーとで、同じ名前のものが存在する場合は、自分自身のプロパティーがプロトタイプオブジェクトのプロパティーを隠すことになります。

B.4.11 プロトタイプチェーン

JavaScript では、「プロトタイプチェーン」(prototype chain) と呼ばれるメカニズムを使うことによって、継承を記述することができるようになっています。

コンストラクタ E のプロトタイプオブジェクトとして、コンストラクタ B によって初期化されたオブジェクトを設定した場合、E によって初期化されたオブジェクトは、E が持っているプロトタイプオブジェクトのプロパティーだけではなくて、B が持っているプロトタイプオブジェクトのプロパティーに対しても、あたかもそれが自分自身のプロパティーであるかのようにアクセスすることができます。このようなメカニズムが、プロトタイプチェーンと呼ばれるものです。

実際に試してみましょう。まず、

```
function Base() {}
Base.prototype.who = function() { alert("Base"); };
```

というように、Base というコンストラクタを定義して、プロトタイプオブジェクトに who というメソッドを追加します。次に、

```
function Extended() {}
Extended.prototype = new Base();
```

というように、Extended というコンストラクタを定義して、Base で初期化されたオブジェクトをプロトタイプオブジェクトとして設定します。次に、

```
var extended = new Extended();
```

という変数文で、Extended で初期化されたオブジェクトを extended という変数に設定します。そうすると、who は、

```
extended.who()
```

というように、あたかも extended に設定されているオブジェクトが持っているメソッドであるかのように呼び出すことができます。

B.5 グローバルオブジェクト

B.5.1 グローバルオブジェクトの基礎

JavaScript の処理系には、グローバルオブジェクト と呼ばれるオブジェクトが組み込まれています。

グローバルオブジェクトが持っているプロパティーには、汎用的な定数とメソッドが設定されています。それらの定数やメソッドには、そのプロパティー名を指定するだけでアクセスすることができます。

B.5.2 グローバルオブジェクトの定数

グローバルオブジェクトは、定数が設定された次のようなプロパティーを持っています。

NaN	非数 (Not a Number)。演算の結果が正常ではないことを示す数値。
Infinity	プラスの無限大。
undefined	未定義値。未定義型の唯一のデータ。

B.5.3 グローバルオブジェクトのメソッド

グローバルオブジェクトは、次のようなメソッドを持っています。

eval(s)	文字列 s がプログラムの場合は、それを実行する。s が式の場合は、それを評価して、その値を返す。
parseInt(s,r)	文字列 s を基数 r で整数に変換した結果を返す。

parseFloat(s)	文字列 s を数値に変換した結果を返す。
isNaN(n)	数値 n が非数ならば真、そうでなければ偽を返す。
isFinite(n)	数値 n が非数、無限大、無限小のいずれでもなければ真、それらのいずれかならば偽を返す。
decodeURI(s)	encodeURI でエンコードされた文字列 s をデコードした結果を返す。
decodeURIComponent(s)	encodeURIComponent でエンコードされた文字列 s をデコードした結果を返す。
encodeURI(s)	文字列 s を URI にエンコードした結果を返す。
encodeURIComponent(s)	文字列 s を URI にエンコードした結果を返す。 encodeURI よりも多くの文字を URI 文字に変換する。

B.6 関数

B.6.1 関数の基礎

関数 (function) は、関数宣言が実行されたとき、または関数式が評価されたときに生成されるオブジェクトです。

B.6.2 関数のメソッド

関数は、次のようなメソッドを持っています。

apply(o,a)	オブジェクト o を this にして関数を呼び出して、配列 a の要素を引数として関数に渡して、その戻り値を返す。
call(o,$\cdots$)	オブジェクト o を this にして関数を呼び出して、2 個目以降の引数を関数に渡して、その戻り値を返す。
bind(o,$\cdots$)	オブジェクト o を this にして関数を呼び出す関数を生成して、生成した関数を返す。2 個目以降の引数は、関数が呼び出されたときにその関数に渡される。

B.7 配列

B.7.1 配列の生成

配列初期化子を評価すると、その値として配列が得られます。

配列は、 Array というコンストラクタでオブジェクトを初期化することによって作ることもできます。n が 0 以上の整数だとするとき、

```
new Array(n)
```

という式を評価すると、大きさが n の配列が生成されます。

B.7.2 配列の大きさ

配列が持っている length というプロパティーには、その配列の大きさ（要素の個数）が設定されています。

B.7.3 配列のメソッド

配列は、次のようなメソッドを持っています。

toString()	文字列に変換した結果を返す。
toLocaleString()	ロケールを考慮して文字列に変換した結果を返す。
concat($\cdots$)	末尾に任意個の引数を連結した結果を返す。自身は変化しない。
join(s)	文字列 s で区切ってすべての要素を連結してできる文字列を返す。
pop()	末尾の要素を削除して、その要素を返す。
push($\cdots$)	末尾に任意個の引数を要素として追加して、処理後の大きさを返す。
reverse()	要素を逆順に並べ替えて、自身を返す。
shift()	先頭の要素を削除して、その要素を返す。

slice(i,j)	i 番目から $j-1$ 番目までの要素から構成される配列を返す。自身は変化しない。
sort(f)	昇順に並べ替えて、自身を返す。f は大小関係を判定する次のような関数。x と y を引数として受け取って、x が y よりも小さいならば -1、同じ大きさならば 0、x が y よりも大きいならば 1 を返す。
splice(i,n,$\cdots$)	i 番目から始まる n 個の要素を削除して、その位置に 3 個目以降の要素を挿入して、削除された要素から構成される配列を返す。
unshift($\cdots$)	先頭に任意個の引数を要素として追加して、処理後の大きさを返す。
indexOf(o,i)	i 番目から末尾の方向へ o を検索して、見つかった場合はその位置、見つからなかった場合は -1 を返す。i が省略された場合は先頭から検索する。
lastIndexOf(o,i)	i 番目から先頭の方向へ o を検索して、見つかった場合はその位置、見つからなかった場合は -1 を返す。i が省略された場合は末尾から検索する。
every(f)	先頭から順番に要素を引数にして関数 f を呼び出して、f が偽を返す要素が見つかった場合は、その時点で終了して偽を返す。見つからなかった場合は真を返す。
some(f)	先頭から順番に要素を引数にして関数 f を呼び出して、f が真を返す要素が見つかった場合は、その時点で終了して真を返す。見つからなかった場合は偽を返す。
forEach(f)	先頭から順番に要素を引数にして関数 f を呼び出す。
map(f)	先頭から順番に要素を引数にして関数 f を呼び出して、f の戻り値から構成される配列を返す。
filter(f)	先頭から順番に要素を引数にして関数 f を呼び出して、f が真を返した要素のみから構成される配列を返す。
reduce(f,v)	v を初期値として、先頭から順番に、前回の戻り値と要素を引数にして関数 f を呼び出して、最後に呼び出された f の戻り値を返す。v が省略された場合は、先頭の要素を初期値として、その次の要素から処理を開始する。
reduceRight(f,v)	v を初期値として、末尾から順番に、前回の戻り値と要素を引数にして関数 f を呼び出して、最後に呼び出された f の戻り値を返す。v が省略された場合は、末尾の要素を初期値として、その前の要素から処理を開始する。

every、some、forEach、map、filter に引数として渡した関数は、3 個の引数を受け取ります。1 個目は要素、2 個目は添字、3 個目は配列全体です。また、それらの関数に対して、2 個目の引数としてオブジェクトを渡すと、そのオブジェクトが this として使われます。

reduce、reduceRight に引数として渡した関数は、4 個の引数を受け取ります。1 個目は前回の戻り値、2 個目は要素、3 個目は添字、4 個目は配列全体です。

B.8 文字列オブジェクト

B.8.1 文字列オブジェクトの基礎

JavaScript では、文字列はオブジェクトではありませんが、オブジェクトとして文字列を扱うことも可能です。オブジェクトとしての文字列は、「文字列オブジェクト」(string object) と呼ばれます。

文字列オブジェクトは、String というコンストラクタを使うことによって明示的に生成することも可能ですが、必要に応じて暗黙的に生成されることもあります。たとえば、

```
"namako".toUppreCase()
```

という式を評価すると、文字列から暗黙的に文字列オブジェクトが生成されて、その文字列オブジェクトが持っている toUppreCase というメソッドが呼び出されます。

B.8.2 String のメソッド

String というコンストラクタは、次のようなメソッドを持っています。

fromCharCode($\cdots$)	引数のそれぞれを文字コードとする文字から構成される文字列を生成して、生成した文字列を返す。

B.8.3 文字列の長さ

文字列が持っている length というプロパティーには、その文字列の長さ（文字の個数）が設定されています。

B.8.4 文字列オブジェクトのメソッド

文字列オブジェクトは、次のようなメソッドを持っています。

valueOf()	自身が持っている文字列を返す。
charAt(i)	i 番目の文字だけから構成される文字列を返す。
charCodeAt(i)	i 番目の文字の文字コードを返す。
concat($\cdots$)	自身の右側にすべての引数を連結した結果を返す。
indexOf(s,i)	i 番目から末尾の方向へ自身の中で文字列 s を検索して、見つかった場合はその部分文字列の先頭の位置、見つからなかった場合は -1 を返す。i が省略された場合は先頭から検索する。
lastIndexOf(s,i)	i 番目から先頭の方向へ自身の中で文字列 s を検索して、見つかった場合はその部分文字列の先頭の位置、見つからなかった場合は -1 を返す。i が省略された場合は末尾から検索する。
localeCompare(s)	ロケールを考慮して自身と文字列 s とを辞書式順序で比較して、自身が s よりも前ならばマイナス、自身と s とが等しいならば 0、自身が s よりも後ろならばプラスの整数を返す。
match(r)	自身の中で正規表現 r を先頭から末尾の方向へ検索して、一致した部分文字列から構成される配列を返す。一致する部分文字列が存在しない場合は null を返す。正規表現に g フラグがある場合、配列の要素は、正規表現に一致したすべての部分文字列。g フラグがない場合、配列の 0 番目の要素は最初に一致した部分文字列の全体、それ以降の要素は、最初に一致した部分文字列のうちで、正規表現の () で囲まれた部分に一致した部分。
replace(r,s)	自身の中で正規表現 r を検索して、一致した部分文字列を文字列 s に置き換えた文字列を返す。s が関数の場合は、一致した部分文字列などを引数にして t を呼び出して、一致した部分文字列を t の戻り値で置き換えた文字列を返す。
search(r)	自身の中で正規表現 r を検索して、見つかった場合はその部分文字列の先頭の位置、見つからなかった場合は -1 を返す。
slice(i,j)	i 番目から $j-1$ 番目まで（j が省略された場合は最後まで）の文字から構成される文字列を返す。位置がマイナスの場合は文字列の末尾から先頭へ向かって数える。
split(r)	正規表現 r に一致する部分文字列で自身を分割して、それぞれの部分を要素とする配列を返す。r が空文字列の場合は、自身を個々の文字に分割して、1 文字の文字列から構成される配列を返す。
substring(i,j)	i 番目から $j-1$ 番目まで（j が省略された場合は最後まで）の文字から構成される文字列を返す。
toLowerCase()	自身に含まれるすべての英大文字を英小文字に置き換えた結果を返す。
toLocaleLowerCase()	自身に含まれるすべての英大文字を、ロケールを考慮して英小文字に置き換えた結果を返す。
toUpperCase()	自身に含まれるすべての英小文字を英大文字に置き換えた結果を返す。
toLocaleUpperCase()	自身に含まれるすべての英小文字を、ロケールを考慮して英大文字に置き換えた結果を返す。
trim()	自身の先頭と末尾にあるホワイトスペースの列を取り除いた結果を返す。

B.9 Math

B.9.1 Math の基礎

JavaScript の処理系には、Math という名前のオブジェクトが組み込まれています。

Math が持っているプロパティーには、数学的な計算をするときに使われる定数とメソッドが設定されています。

B.9.2 Math の定数

Math は、定数が設定された次のようなプロパティーを持っています。

E	e	自然対数の底（ネイピア数）
LN10	$\log_e 10$	10 の自然対数の値
LN2	$\log_e 2$	2 の自然対数の値
LOG2E	$\log_2 e$	自然対数の底の、2 を底とする対数の値
LOG10E	$\log_{10} e$	自然対数の底の、10 を底とする対数の値
PI	π	円周率
SQRT1_2	$\sqrt{1/2}$	1/2 の平方根
SQRT2	$\sqrt{2}$	2 の平方根

B.9.3 Math のメソッド

Math は、次のようなメソッドを持っています。

abs(x)	x の絶対値を返す。
acos(x)	x のアークコサイン（逆余弦）を返す。
asin(x)	x のアークサイン（逆正弦）を返す。
atan2(x, y)	x と y の符合を考慮して、y/x のアークタンジェント（逆正接）を返す。
ceil(x)	x よりも小さくない最小の整数を返す。
cos(x)	x のコサイン（余弦）を返す。
exp(x)	自然対数の底の x 乗を返す。
floor(x)	x よりも大きくない最大の整数を返す。
log(x)	x の自然対数の値を返す。
max(···)	任意個の引数のうちで最大のものを返す。
min(···)	任意個の引数のうちで最小のものを返す。
pow(x, y)	x の y 乗を返す。
random()	0 以上 1 未満の区間で発生させた乱数を返す。
round(x)	x の小数点以下を四捨五入した結果を返す。
sin(x)	x のサイン（正弦）を返す。
sqrt(x)	x の平方根を返す。
tan(x)	x のタンジェント（正接）を返す。

三角関数のメソッドが扱う角度の単位は、ラジアンです。

B.10 日付オブジェクト

B.10.1 日付オブジェクトの基礎

JavaScript の処理系には、Date という名前のコンストラクタが組み込まれています。

Date を使って生成されたオブジェクトは、「日付オブジェクト」(date object) と呼ばれます。日付オブジェクトは、日付と時刻をあらわすオブジェクトです。

B.10.2 日付オブジェクトの生成

日付オブジェクトを、

new Date(y, m, d, h, m, s, ms)

という式で生成した場合、その日付オブジェクトには、y 年 $m+1$ 月 d 日 h 時 m 分 s 秒 ms ミリ秒が設定されます（最低限必要な引数は 2 個です）。

すべての引数を省略して、

```
new Date()
```

という式で日付オブジェクトを生成することもできます。その場合、その日付オブジェクトには、現在の日付と時刻が設定されます。

B.10.3 Date のメソッド

Date というコンストラクタは、次のようなメソッドを持っています。

parse(s)	文字列 s であらわされる日時の時間値（1970 年 1 月 1 日 0 時 0 分 0 秒 (UTC) からの経過時間（単位はミリ秒）をあらわす整数）を返す。
UTC(y,m,d,h,m,s,ms)	y 年 $m+1$ 月 d 日 h 時 m 分 s 秒 ms ミリ秒の時間値を返す。最低限必要な引数は 2 個。
now()	現在の時間値を返す。

B.10.4 日付オブジェクトのメソッド

日付オブジェクトは、次のようなメソッドを持っています。

toString()	自身をあらわす文字列を返す。
toDateString()	自身の日付をあらわす文字列を返す。
toTimeString()	自身の時刻をあらわす文字列を返す。
toLocaleString()	ロケールで規定された書式で自身をあらわす文字列を返す。
toLocaleDateString()	ロケールで規定された書式で自身の日付をあらわす文字列を返す。
toLocaleTimeString()	ロケールで規定された書式で自身の時刻をあらわす文字列を返す。
valueOf()	自身の時間値を返す。
getTime()	自身の時間値を返す。
getFullYear()	自身のローカル時間の年を返す。
getUTCFullYear()	自身の協定世界時の年を返す。
getMonth()	自身のローカル時間の月（1 月は 0）を返す。
getUTCMonth()	自身の協定世界時の月を返す。
getDate()	自身のローカル時間の日を返す。
getUTCDate()	自身の協定世界時の日を返す。
getDay()	自身のローカル時間の曜日（日曜日は 0）を返す。
getUTCDay()	自身の協定世界時の曜日を返す。
getHours()	自身のローカル時間の時を返す。
getUTCHours()	自身の協定世界時の時を返す。
getMinutes()	自身のローカル時間の分を返す。
getUTCMinutes()	自身の協定世界時の分を返す。
getSeconds()	自身のローカル時間の秒を返す。
getUTCSeconds()	自身の協定世界時の秒を返す。
getMilliseconds()	自身のローカル時間のミリ秒を返す。
getUTCMilliseconds()	自身の協定世界時のミリ秒を返す。
getTimezoneOffset()	自身の協定世界時から自身のローカル時間を減算した結果を分であらわしたものを返す。
setTime(t)	時間値 t を自身に設定して、t を返す。
setMilliseconds(ms)	ms ミリ秒をローカル時間として自身に設定して、設定後の自身の時間値を返す。
setUTCMilliseconds(ms)	ms ミリ秒を協定世界時として自身に設定して、設定後の自身の時間値を返す。

setSeconds(s,ms)	s 秒 ms ミリ秒をローカル時間として自身に設定して、設定後の自身の時間値を返す。最低限必要な引数は 1 個。
setUTCSeconds(s,ms)	s 秒 ms ミリ秒を協定世界時として自身に設定して、設定後の自身の時間値を返す。最低限必要な引数は 1 個。
setMinutes(m,s,ms)	m 分 s 秒 ms ミリ秒をローカル時間として自身に設定して、設定後の自身の時間値を返す。最低限必要な引数は 1 個。
setUTCMinutes(s)	m 分 s 秒 ms ミリ秒を協定世界時として自身に設定して、設定後の自身の時間値を返す。最低限必要な引数は 1 個。
setHours(h,m,s,ms)	h 時 m 分 s 秒 ms ミリ秒をローカル時間として自身に設定して、設定後の自身の時間値を返す。最低限必要な引数は 1 個。
setUTCHours(h,m,s,ms)	h 時 m 分 s 秒 ms ミリ秒を協定世界時として自身に設定して、設定後の自身の時間値を返す。最低限必要な引数は 1 個。
setDate(d)	d 日をローカル時間として自身に設定して、設定後の自身の時間値を返す。
setUTCDate(d)	d 日を協定世界時として自身に設定して、設定後の自身の時間値を返す。
setMonth(m,d)	$m+1$ 月 d 日をローカル時間として自身に設定して、設定後の自身の時間値を返す。最低限必要な引数は 1 個。
setUTCMonth(m,d)	$m+1$ 月 d 日を協定世界時として自身に設定して、設定後の自身の時間値を返す。最低限必要な引数は 1 個。
setFullYear(y,m,d)	y 年 $m+1$ 月 d 日をローカル時間として自身に設定して、設定後の自身の時間値を返す。最低限必要な引数は 1 個。
setUTCFullYear(y,m,d)	y 年 $m+1$ 月 d 日を協定世界時として自身に設定して、設定後の自身の時間値を返す。最低限必要な引数は 1 個。
toUTCString()	自身の協定世界時をあらわす文字列を返す。
toISOString()	ISO 8601 で規定された、YYYY–MM–DDTHH:mm:ss.sssZ という形式で自身をあらわす文字列を返す。
toJSON()	JSON の形式で自身をあらわす文字列を返す。

B.11 正規表現オブジェクト

B.11.1 正規表現オブジェクトの生成

正規表現リテラルを評価すると、その値として正規表現オブジェクトが得られます。

正規表現オブジェクトは、RegExp というコンストラクタでオブジェクトを初期化することによって作ることもできます。r と f が文字列だとするとき、

```
new RegExp(r, f)
```

という式を評価すると、r を正規表現、f を正規表現フラグとする正規表現オブジェクトが生成されます。

B.11.2 正規表現フラグ

正規表現フラグは、次の文字から構成される文字列です。

- g 文字列の全体をグローバル (global) に検索する。
- i 大文字と小文字を区別しない (ignore case)。
- m 複数行 (multiline) を対象として検索する。

B.11.3 正規表現オブジェクトのメソッド

正規表現オブジェクトは、次のようなメソッドを持っています。

exec(s)	文字列 s を対象として、自身に一致する部分文字列を先頭から末尾の方向へ検索する。戻り値は配列で、0 番目の要素は最初に一致した部分文字列の全体、それ以降の要素は、最初に一致した部分文字列のうちで、正規表現の () で囲まれた部分に一致した部分。一致する部分文字列が存在しない場合は null を返す。正規表現

にgフラグがある場合、次にexecが呼び出されたときは、一致した部分文字列の次の位置から検索を開始する。

`test(`*s*`)` 自身に一致する部分文字列が文字列 *s* の中に存在するならば真、そうでなければ偽を返す。

`toString()` 自身を正規表現リテラルであらわす文字列を返す。

参考文献

[Baranovskiy,2010] Dmitry Baranovskiy, “Raphaël Reference”, 2010.

[Campesato,2003] Oswald Campesato, *Fundamentals of SVG Programming: Concepts to Source Code*, Charles River Media, 2003, ISBN 978-1-58450-298-2.

[CSS,2006] Bert Bos, Tantek Çelik, Ian Hickson and Håkon Wium Lie (eds.), “Cascading Style Sheets, level 2 revision 1”, World Wide Web Consortium, 2006.

[DOM2,2000] Arnaud Le Hors, Philippe Le Hegaret, Lauren Wood, Gavin Nicol, Jonathan Robie, Mike Champion and Steve Byrne (eds.), “Document Object Model (DOM) Level 2 Core Specification, Version 1.0”, World Wide Web Consortium, 2000.

[ECMAScript,2009] “ECMAScript Language Specification, Fifth Edition”, Ecma International, 2009.

[Eisenberg,2002] J. David Eisenberg, *SVG Essentials*, O’Reilly & Associates, 2002, ISBN 978-0-596-00223-7.

[Flanagan,1998] David Flanagan, *JavaScript: The Definitive Guide, Third Edition*, O’Reilly & Associates, 1998, ISBN 978-1-56592-392-8. 邦訳（安藤進、垰井正雄）、『JavaScript』第3版、オライリー・ジャパン、2000、ISBN 978-4-87311-027-1。

[Laaker,2002] Micah Laaker, *Sams Teach Yourself SVG in 24 Hours*, Sams Publishing, 2002, ISBN 978-0-672-32290-7.

[Pearlman,2003] Ellen Pearlman and Lorien House, *SVG for Web Developers*, Prentice Hall, 2003, ISBN 978-0-13-100499-3.

[SVG,2003] Jon Ferraiolo, Fujisawa Jun and Dean Jackson (eds.), “Scalable Vector Graphics (SVG) 1.1 Specification”, World Wide Web Consortium, 2003.

[XML,2004] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler and François Yergeau (eds.), “Extensible Markup Language (XML) 1.0 (Third Edition)”, World Wide Web Consortium, 2004.

[大藤,2001] 大藤幹、半場方人、『詳解 HTML&CSS&JavaScript 辞典』、秀和システム、2001、ISBN 978-4-7980-0220-0。

[佐久嶋,2005] 佐久嶋ひろみ、『JavaScript ハッカーズ・プログラミング』、メディア・テック出版、2005、ISBN 978-4-89627-286-4。

[佐藤,2007] 佐藤信正、『JavaScript 完全マスター・再入門編』、メディア・テック出版、2007、ISBN 978-4-89627-344-1。

[羽田野,2007] 羽田野太巳、『標準 DOM スクリプティング——JavaScript+DOM による Web アプリデザインの基礎——』、ソフトバンククリエイティブ、2007、ISBN 4-7973-3638-2。

[羽田野,2010] 羽田野太巳、白石俊平、古籏一浩、太田昌吾、『Google API Expert が解説する HTML5 ガイドブック』、インプレスジャパン、2010、ISBN 978-4-8443-2927-5。

[平野,2010] 平野照比古、「SVG ではじめる Web Graphics・2010 年版」、神奈川工科大学、2010。

[古籏,2008] 古籏一浩、『DOM Scripting——高機能な Web ページ構築のために——』、Web 標準テキストシリーズ、1、技術評論社、2008、ISBN 978-4-7741-3326-3。

[柳井,2010] 柳井政和、『マンガでわかる JavaScript』、秀和システム、2010、ISBN 978-4-7980-2783-8。

[山田,2010] 山田祥寛、『JavaScript 本格入門：モダンスタイルによる基礎から Ajax・jQuery まで』、技術評論社、2010、ISBN 978-4-7741-4466-5。

索引