OCamlで構築するモダンWeb：
型付きHTML5プログラミングの実際

IT Planning, Inc.

有限会社ITプランニング

今井 敬吾 (@keigoi)

PPLサマースクール2012

関数型言語ベースの先進的Webフレームワーク 午後の部

2012年 8月 21日 (火) 14:30 - 17:30 法政大学 小金井キャンパス

準備

1. VMの ~/ocaml に教材をダウンロード

端末上で

```
cd ~/ocaml
git clone https://github.com/keigoi/material.git
```

→ /home/pplss2012/ocaml に material というディレクトリができます

2. ~/ocaml/material で fixinstall.sh を実行

```
cd ~/ocaml/material
sh fixinstall.sh
```

- 別のダウンロードが始まります
- デスクトップにアイコンが増えます （ドキュメント群とブラウザ上OCaml）
- 一部、処理系が不調だった（警告が出ていた）不具合を修復します

本講義について

- 静的な型が付いて**安全**な、クライアントサイドWebプログラミングの技法についてのお話
- OCamlからJavaScriptへ変換（コンパイル）する処理系 **Js_of_ocaml** を使う
- OCamlは型安全性をもち、ネイティブコンパイルできる高速な言語処理系

私について

- 名古屋から来ました
- （有）ITプランニング のソフトウェアエンジニア
- Haskell / OCaml の両方が好き！ で、日常的にOCamlを書いている
- OCaml toplevel on Android を開発
- 2011年、OCamlJSを使ったHTML5アプリケーションを納品
  - http://www.itpl.co.jp/conga-forex-chart/
- Ph.D. (2012年3月, 名古屋大学)

クライアントサイドWebと JavaScript・ HTML5

- JavaScript：クライアントサイドWebのスタンダード
  - 主要なブラウザ全てが標準装備
- HTML5と関連する技術：
  - グラフィクス(Canvas,WebGL)、通信(WebSockets)、ローカルストレージ、等

潮流：JavaScriptへのコンパイル言語

- CoffeeScript
- Dart
- Haxe
- S2JS (Scala)
- JSX
- **Js_of_ocaml**, Ocamljs (**OCaml**)

OCaml

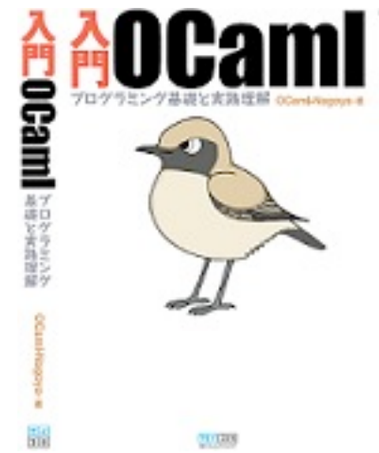
- 関数型プログラミングを基礎におく、マルチパラダイムな言語
- 静的型付けによる信頼性
- 類を見ないユニークな型機能を多くもつ
  - オブジェクト指向（構造的サブタイピング）
  - ラベル付き引数
  - 多相ヴァリアント
  - MLスタイルのモジュールシステム
- なおかつ、ほぼ完全な型推論をもつ
  （プログラムが簡潔になる）

日本語のOCaml書籍

- プログラミング in OCaml:

  PDF版 発売開始！

  https://gihyo.jp/dp/ebook/2012/978-4-7741-5314-8



入門OCaml

(絶版)

- プログラミングの基礎

OCamlを触ってみよう

- OCamlトップレベル：ターミナルから ocaml で起動
  - 式 / 定義を入力し、**;;（セミコロン2つ）で終端**

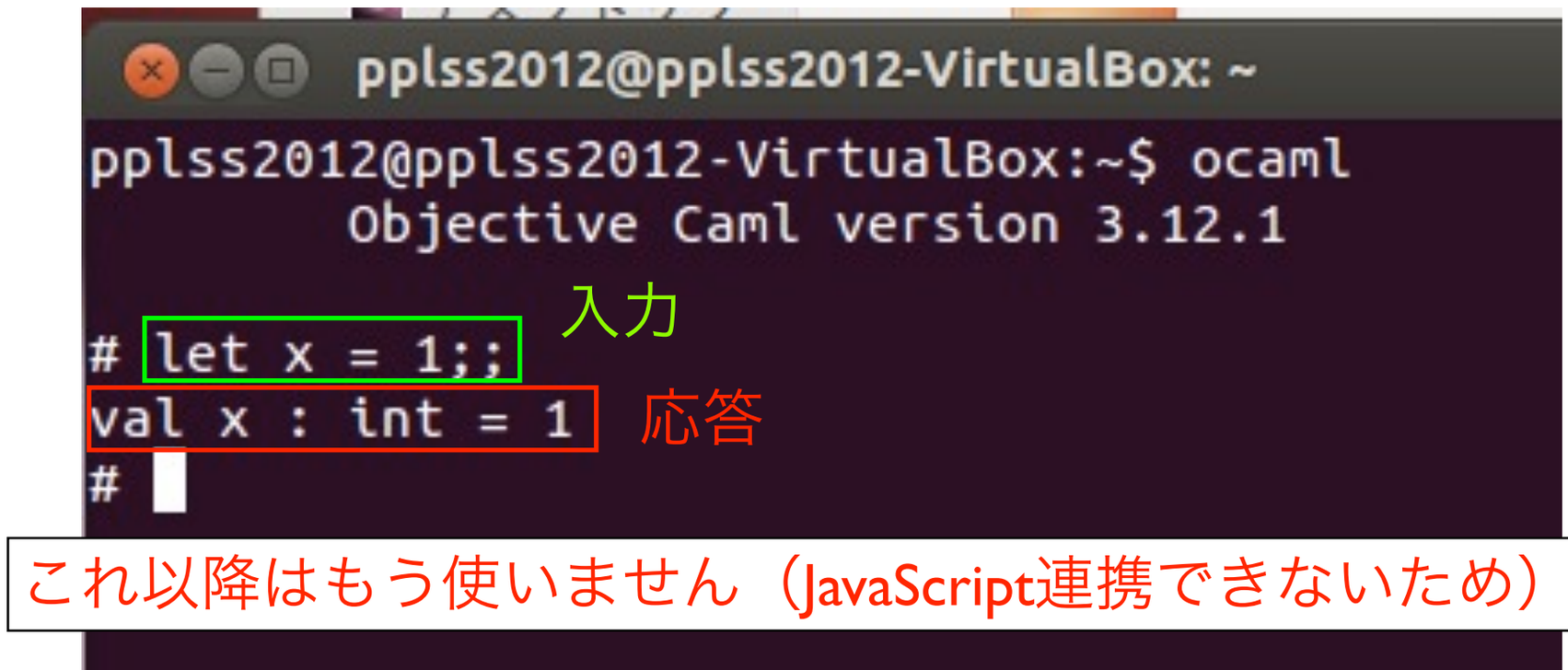
入力してみよう

```
let x = 10 + 10;;
```
int型の値の定義

```
300. *. 1.05;;
```
float型の式 (ドットに注意)

```
print_endline "Hello,World!";;
```
(副作用のある式

（入力支援のためVM環境は alias ocaml='rlwrap ocaml' してあります）



これ以降はもう使いません（JavaScript連携できないため）

Js_of_ocaml

http://ocsigen.org/js_of_ocaml/

- OCamlバイトコードからJavaScriptへのコンパイラ
- 生成されるコードは高速
  - VM実行のOCamlより、Js_of_ocamlが生成したJavaScriptの方が速い事例（！）も
- OCamlのオブジェクト型システムでJavaScriptに型を付ける

到達目標

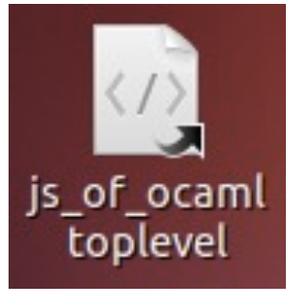
- JavaScriptのメソッド呼び出しやプロパティ参照が、Js_of_ocamlでどのように対応するかを知る
- Js_of_ocamlのドキュメントを読み、所望のオブジェクト/関数/メソッドを得る方法が分かる
- 処理系の基本的な使い方が分かる

PPLSS2012謹製

しばらくこれを使います

Js_of_ocamlトップレベル

- http://proofcafe.org/try_jsocaml/
  をGoogle Chrome  で開いて下さい



（この講義ではGoogle Chromeを使います）

（または**デスクトップ上の アイコン js_of_ocaml toplevel**）

入力してみよう

```
open Dom_html;;
let alert msg = window##alert(msg);;
alert (Js.string”Hello,World!”);;
```

```
alert((jsnew Js.date_now())##toString());;
```

```
open Dom_html;;
let alert msg = window##alert(msg);;
alert (Js.string”Hello,World!”);;
```

`open Dom_html` により、修飾 Dom_html. を省略できるようにする

- JavaScriptのメソッドは、(js_of_ocamlの構文拡張により)
  obj##meth(arg1,arg2,...) のようにして呼び出せます
  つまり

`window##alert(msg)` は `window.alert(msg)` と等価

js_of_ocaml　　JavaScript

- コンストラクタの呼び出し構文 `jsnew constr (arg1,arg2,…)`

ログ出力

- JavaScriptでは、ChromeやFirefoxだと

```
console.log(“ログ”)
```

で、コンソールにログ出力できる

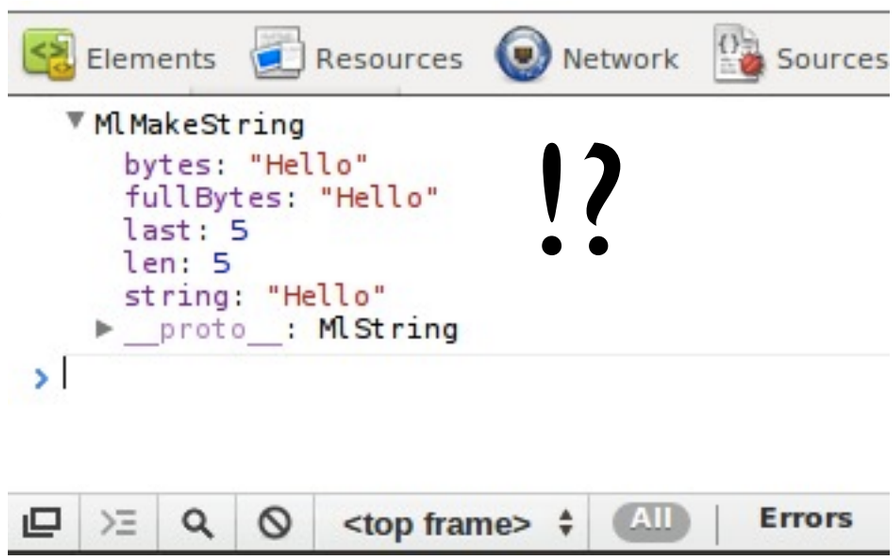
- Js_of_ocaml では

```
Firebug.console##log("Hello!");;
```

入力してみよう

コンソールは

Ctrl-Shift-Iで開きます

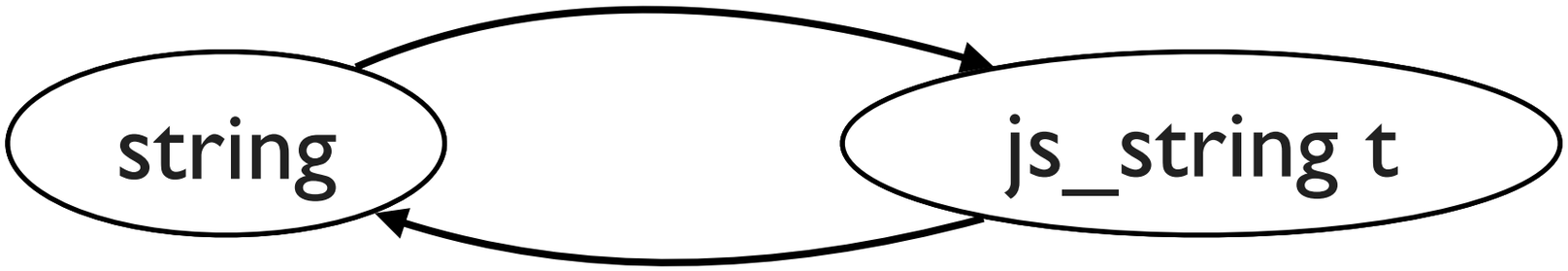


JavaScriptの文字列≠OCamlの文字列

- OCamlの文字列型は string

  JavaScriptの文字列型は js_string t 型

Js.string : string → js_string t

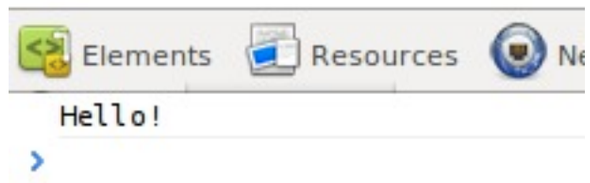


Js.to_string : js_string t → string

- 特に Js.string“文字列” は頻出するので、次を定義しておく

```
let js = Js.string;;
```

```
Firebug.console##log(js"Hello!");;
```

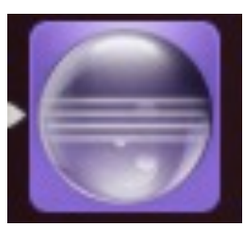


高機能なエディタを使おう

- OCamlは型にうるさい。Js_of_ocaml 然り
- 頻発する型エラーを分かりやすく得たい

VMにインストール済み：

か



+

emacs+caml-mode
+typerex

安定しています

eclipse+OCaIDE

（よくエラーが出ますが使えます）

emacs, eclipseどちらも使いたくない人はゴメンナサイ

emacs+caml-mode +TypeRex

- すぐ使えます

```
emacs ~/ocaml/material/ball/ball.ml
```

- キーバインド：
  - C-C C-C コンパイル (makeが走ります)
  - C-C C-T カーソル位置の型を知る
  - C-O TypeRex機能

eclipse+OCaIDE

- File → Import → Existing Projects into Workspace で

  ~/ocaml/material/ball,
  ~/ocaml/material/canvas をインポートしてください

  - ファイル保存のたびにmakeが走ります

  - マウスカーソル位置の型がポップアップします

追記：

当日使用したのは、今井による改造OCaIDEです

(js_of_ocamlを自動的に呼び出し、JavaScriptを生成する修正を付加)

http://proofcafe.org/~keigoi/OcaIDE_fix_201208/site.xml より入手可能（ソースはhttps://github.com/keigoi/OcaIDE）

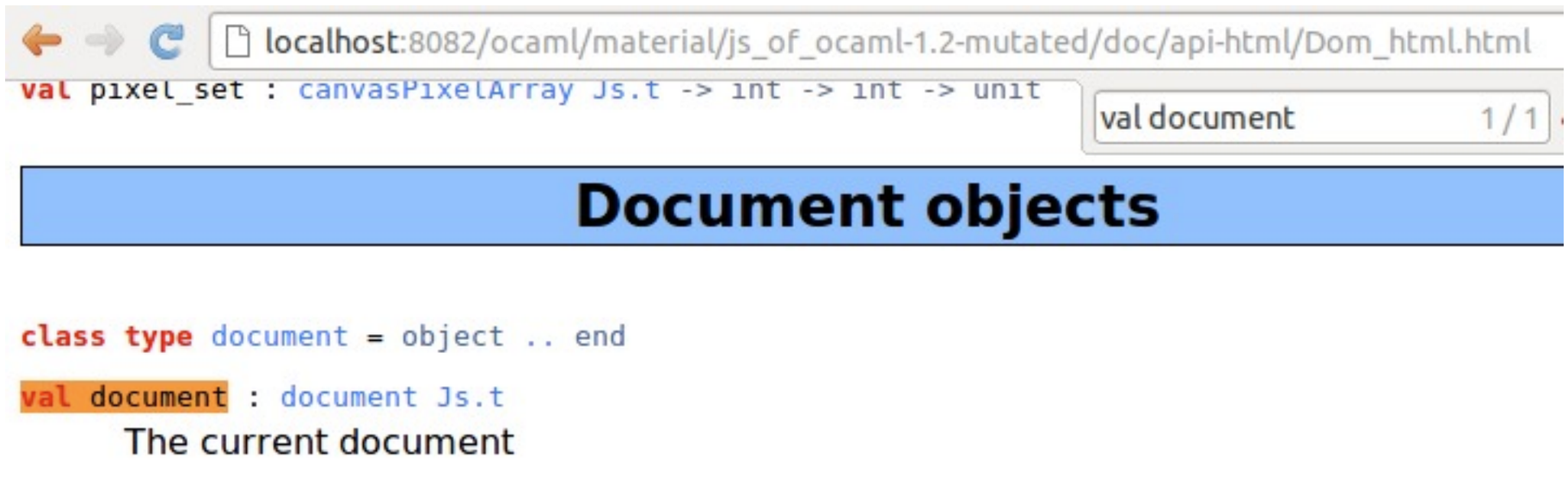
ドキュメントを覗いてみよう

- HTMLの要素など、グローバルなDOMオブジェクトは Dom_htmlモジュールで定義されています

JavaScript	Js_of_ocaml
document	Dom_html.document
window	Dom_html.window

をダブルクリックしてドキュメントを開いてみて下さい

Dom_html をクリックし、Ctrl+F で **val document を検索**



js_of_ocamlで使うモジュール群

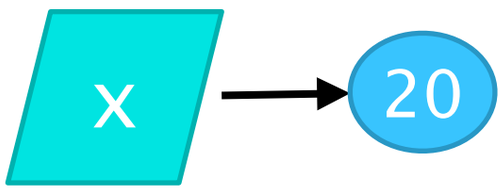
- Js
  基本ライブラリ
- Dom, Dom_html, Dom_events, Form
  DOM / Webページの操作
- Lwt , Lwt_js
  協調的スレッドライブラリ
- XmlHttpRequest
  非同期HTTP通信（Lwtを利用）
- File
  HTML5 local storageライブラリ
- Json, Deriving_Json
  JSON、型安全なJSONの扱い
- Firebug
  Firebugのログ出力、タイマー等
- Regexp
  JavaScript由来の正規表現モジュール
- Url
  Urlのエンコード/デコード/現在表示中のページの情報
- WebGL
  3Dグラフィクスライブラリ
- Typed_array
  WebGLで用いる高速なJavaScript配列

OCamlおさらい

OCamlの基本：letによる定義

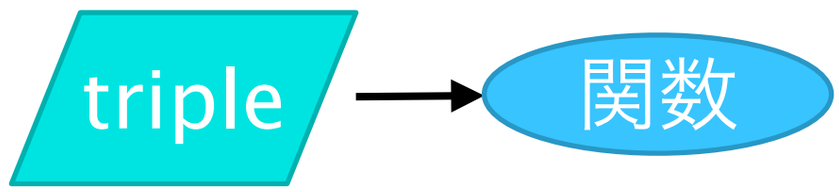
値の定義

```
let x = 10 + 10;;
```



関数定義

```
let triple x = x * 3;;
```




```
let x = 10 + 10;;
let x = "Hello,World!";;
```

上書き（シャドーイング）できる

（OCamlではよくある）

OCamlの式

```
{name="imai";age=100};; レコード生成
person.name;; フィールド参照
fun x -> x + 1;;  ラムダ式
print_endline "Hello";;  関数呼び出し
let pi = 3.14 in pi *. r *. r ;; ローカルlet
if password="ppl" then Ok else Ng;;  if式
match exp with
| Orange -> "I love!"
| Lemon -> "I hate!";;  パターンマッチ
function
| [] -> 0    ラムダ式＋パターンマッチ
| x::xs -> x + sum xs;;
try
    List.assoc key lst
with  例外処理
| Not_found -> "default";;
raise Not_found;;    例外送出
```

```
( exp )

begin exp end
```

括弧の代わりにbegin/endを
使うことがある

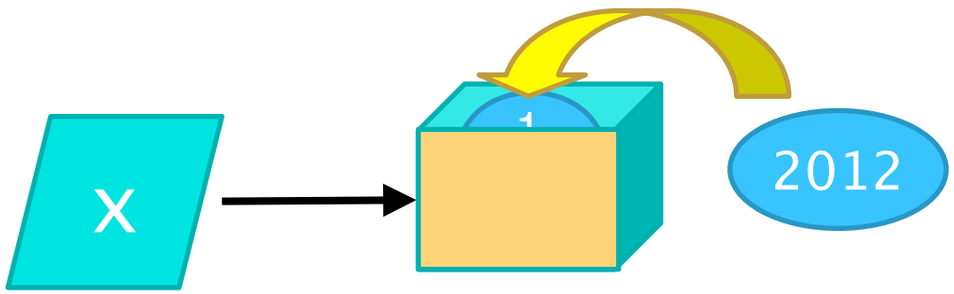
```
if password="ppl" then begin
  …
end else begin
  …
end
```

```
begin match exp with
| Orange -> "I love!"
| Lemon -> "I hate!";;
end;
print_endline "Done"
```

副作用

- 参照セル(ref型)

```
# let x = ref 1;;     初期化
val x : int ref = {contents=1}
# x := 2012;;     破壊的代入
- : unit = ()
# !x;;   参照(dereference)
- : int = 2012
```



参照セル

- 入出力

```
# let x = print_endline "Hello,World!";;
Hello,World!
val x : unit = ()
```

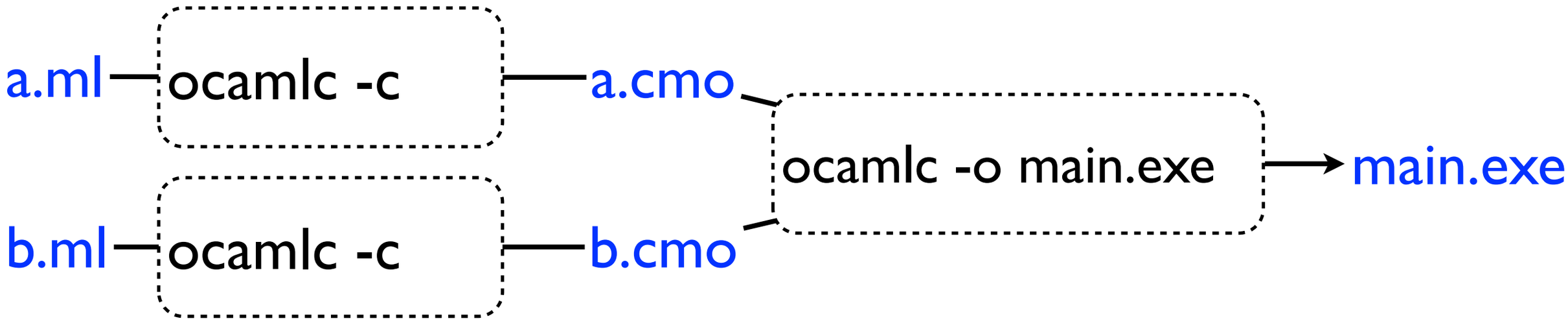
- セミコロンで式を逐次評価

```
# let x =
    print_endline "Hello,World!";
    3.14159;;
Hello,World!
val x : float = 3.14159
```

- 例外処理

モジュールとプログラム

- OCamlプログラムは、**モジュールの集まり**
  - モジュールファイル：
    ソースコード .ml / .mli (インタフェース記述)
    コンパイル済みオブジェクト .cmo / .cmi
  - アーカイブ： .cma （複数のモジュールをまとめたファイル）



ネイティブコンパイルの時 .cmx, .cmxa

モジュール

- モジュールは複数の文（定義の羅列）からなる
- 文は「**上から下に**」順に評価される：
  定義が副作用をもつため

副作用をもつ**定義**の例：

```
let incr : int -> int =
   print_endline "Init!";
   fun x -> x+1;;
```

モジュール A (ファイル名 a.ml)

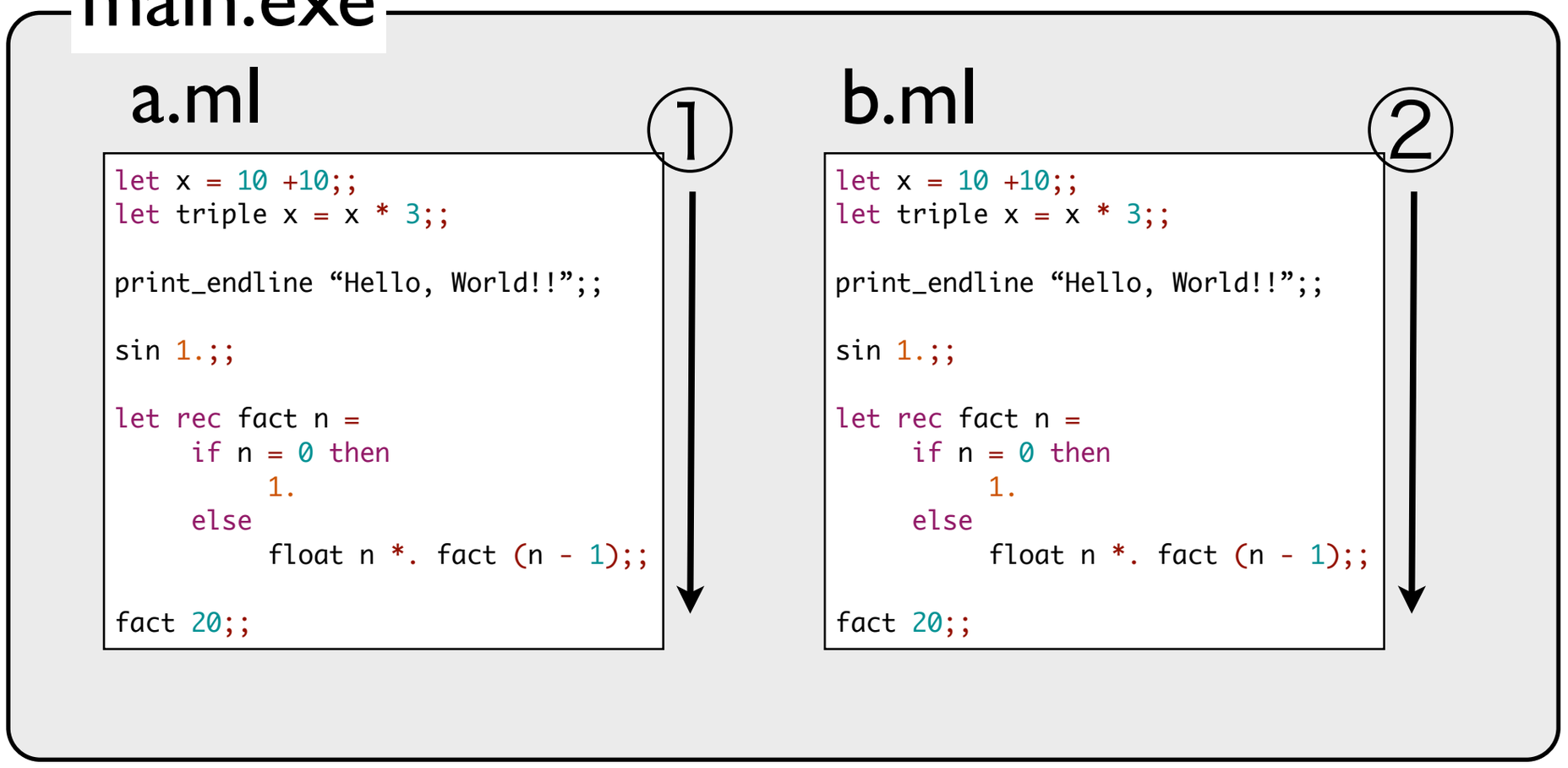
```
let x = 10 +10;;
let triple x = x * 3;;

print_endline “Hello, World!!”;;

sin 1.;;

let rec fact n =
   if n = 0 then
      1.
   else
      float n *. fact (n - 1);;

fact 20;;
```

上から下に

モジュールのロード

- リンク時に指定した順でモジュールが初期化される

① ②
$ ocamlc a.ml b.ml -o main.exe



ロード順序に依存する初期化処理は書くべきでない

コンパイラ ocamlc：標準ライブラリとサーチパス

例

```
ocamlc -o main.exe -I +threads str.cma threads.cma a.ml
```

✴ **デフォルトでは：**

```
.
|- a.ml
```

- stdlib.cma （標準ライブラリ）のみを自動リンク
  (PervasivesやHashtblなどのモジュールを含む)

- サーチパスはカレントと `ocamlc -where` (/usr/lib/ocaml) のみ

✴ 標準ライブラリ以外について：サーチパスは-I で、
モジュール(群) は .cmo, .cma のファイル名指定で明示的に行う

- +記号で相対指定（-I +threads で /usr/lib/ocaml/threads を表す）

```
/usr/lib/ocaml
|-nums.cma
|-str.cma
|-stdlib.cma
|-threads/
  |-threads.cma
|-unix.cma
|...
```

モジュール間の参照

- モジュール名は大文字
- ファイル名は（慣習的に）小文字で始める

プログラム

モジュールA（a.ml）

```
let drawLine (x1,y1) (x2,y2) =
...
```

モジュールB (b.ml)

```
… A.draw_line (0,0)
              (20,15);;
```

ファイル間の循環参照はできない

OCamlコードの読み方

- モジュールは定義の列　　定義1 定義2 定義3 …

定義の開始は

let,type,exception,module,
class, open, include など

```
let x = 10 +10✓ let triple
x = x * 3 ✓ type fruit =
Apple | Banana | Orange ✓
type account = {user:
string; password: string}✓
exception My_exn ✓ module
M = struct end ✓ class c =
object end
```

（区切りを ✓ で明示）

（左と等価だが読みやすいコード）

```
let x = 10 +10                    let定義(値,関数)
let triple x = x * 3

type fruit = Apple | Banana | Orange
type account =                              型定義
   {user: string; password: string}

exception My_exn 例外定義

module M = struct end ネストされたモジュール定義
class c = object end  クラス定義
```

- 定義の区切りは ;; で明示できる

```
let x = 10 +10;;
let triple x = x * 3;;
```

OCamlコードの読み方

- モジュールは定義の列　　定義1 定義2 定義3 …

  定義の開始は

  let,type,exception,module,
  class, open, include など

```
let x = 10 +10  let triple
x = x * 3    type fruit =
Apple | Banana | Orange
type account = {user:
string; password: string}
exception My_exn    module
M = struct end    class c =
object end
```

（区切りを✓で明示）

（左と等価だが読みやすいコード）

```
let x = 10 +10                          let定義(値,関数)
let triple x = x * 3

type fruit = Apple | Banana | Orange
type account =                                  型定義
   {user: string; password: string}

exception My_exn 例外定義

module M = struct end ネストされたモジュール定義
class c = object end  クラス定義
```

- 定義の区切りは ;; で明示できる

```
let x = 10 +10;;
let triple x = x * 3;;
```

ocamlfind（findlib）

- 複雑なモジュールの依存関係をライブラリ単位で管理してくれる便利ツール

```
ocamlfind ocamlc \
  -linkpkg -o main.exe \
  -syntax camlp4o \
  -package js_of_ocaml,js_of_ocaml.syntax \
  a.ml
```

実行形式main.exeを生成

構文拡張をオンに

依存ライブラリの指定

```
ocamlc.opt -verbose -o main.exe -I /opt/local/lib/ocaml3/site-lib/lwt -I /opt/local/
 lib/ocaml3/camlp4 -I /opt/local/lib/ocaml3/site-lib/js_of_ocaml -pp "camlp4 '-I' '/
  opt/local/lib/ocaml3' '-I' '/opt/local/lib/ocaml3/camlp4' '-I' '/opt/local/lib/ocaml3/
      site-lib/js_of_ocaml' 'dynlink.cma' '-parser' 'o' '-parser' 'op' '-printer' 'p'
 'pa_js.cmo' " /opt/local/lib/ocaml3/site-lib/lwt/lwt.cma /opt/local/lib/ocaml3/site-
      lib/js_of_ocaml/js_of_ocaml.cma /opt/local/lib/ocaml3/dynlink.cma a.ml
```

ocamlcに与えるパラメータは多様で複雑

（モジュールのサーチパス-I、依存する全モジュール、構文拡張）

OCamlの
オブジェクトシステム

- Js_of_ocamlの基礎であるOCamlのオブジェクトシステムと、構造的多相性（structural polymorphism）の考え方に関して

OCamlのオブジェクト

js_of_ocamlでは使いません。（OCamlのオブジェクトはJavaScriptに渡せないため。）
オブジェクトの型付けの仕組みだけをJavaScriptに流用します。

- オブジェクト式：

  object method メソッド名 仮引数1 .. = メソッド本体 .. end

```
let hello_obj =
  object
    method hello = print_endline "Hello, World"
    method add x y = x + y
  end;;
```

- メソッド呼び出し: 式#メソッド名 引数1 引数2 ...

```
hello_obj#hello;;
print_int (hello_obj#add 1 2);;
```

オブジェクトの型付けは

構造的

```
object
   method hello = print_endline "Hello, World"
   method add x y = x + y
end
```

の型は

```
< add : int -> int -> int; hello : unit >
```

- どんなメソッドを持っているかが型の**構造**に現れる

（≠Java,C#の名前ベースのサブタイピング）

OCamlのクラス

- クラス宣言

```
class hello_cls =
   object
      method hello = print_endline "Hello, World"
   end;;
```

クラスhello_clsと**クラス型**hello_clsが導入される。クラスはnewできる。

```
let hello_obj : hello_cls = new hello_cls
in hello_obj#hello
```

一方、`hello_cls` 型（**クラス型**）は `<hello : unit>` の別名。

クラス型は Javaでいうinterfaceのようなもの

js_of_ocamlではクラスを使いません。 class type で定義されたクラス型でオブジェクトに型を与えます

クラス型定義

- クラス型のみを定義できる

```
class type hello_cls_typ =
  object
    method hello : unit
  end;;
```

- クラス定義との違い：**newできない**

（~~new hello_cls_typ~~ とは書けない）

ここまでのまとめ

- オブジェクト式

```
object method hello = "Hello" end
```

- オブジェクト型

```
<hello : string>
```

```
(x : <hello : string>)
```

- クラス定義

```
class hello_cls = object method hello = "hello" end
```

```
(x : hello_cls)
```

```
new hello_cls
```

- クラス型定義

```
class type hello_typ = object method hello : string end
```

```
(x : hello_typ)
```

```
new hello_typ
```

構造的サブタイピング

t <: s　（sはtのスーパータイプ / tはsのサブタイプ）とは：

1. tがsのメソッドを含む
2. sの各メソッドの型がtのメソッドのスーパータイプ

```
s = <m_1 : t_1; m_2 : t_2; … ; m_k : t_k >
     ∨          ∨          ∨  …        ∨
t = <m_1 : t_1'; m_2 : t_2'; … ; m_k : t_k'; …; m_n : t_n' >
```

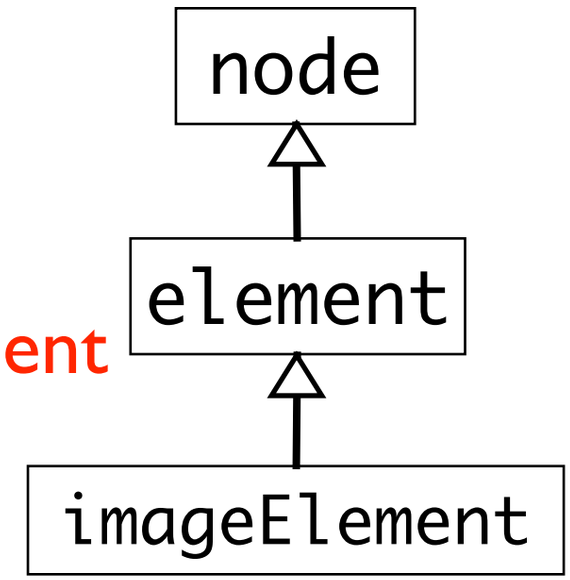
OCamlにおけるサブタイプ多相：

コアーション（型強制）

- 明示的なアップキャスト（**コアーション**）が必要
  - 例：`method appendChild : node -> unit,`
    `img : imageElement` のとき

```
elm##appendChild(img)
```
型エラー，node≠imageElement

```
elm##appendChild(img :> node)
```
Ok



コアーション（アップキャスト）

- コアーションが不要な、**#-型**を使いましょう（次頁）

#-型を使おう

(列多相,row-polymorphism)

- nodeクラスの `method appendChild : node -> unit` の代替の関数

```
Dom.appendChild : #node -> #node -> unit
```

は、コアーションが不要：`Dom.appendChild elm img`

- #-型：残りの部分を表す特殊な型変数（列変数）を含む型

  オブジェクト型の記法では `<hello:string; ..>` と書く(‘..’が列変数)

	クラス型	オブジェクト型
列多相あり	#hello_typ	<hello:string; ..>
列多相なし	hello_typ	<hello:string>

‘..’ は「それ以外の何か」

objでhelloを呼びます！

```
# let say_hello obj = print_endline obj#hello;

val say_hello : < hello : string; .. > -> unit = <fun>
```

helloと、それ以外の何かをもつオブジェクトを下さい！

JavaScriptオブジェクトの扱い

- JavaScriptの値（オブジェクト）は、OCaml側で

```
'a Js.t
```

という抽象型をもつ（'aにはオブジェクトの表現が入る）

- 例：

```
Dom.element Js.t
```

（Dom.element：DOM要素のクラス型）

```
class element = object
  inherit node
  method tagName : js_string t readonly_prop
  method getAttribute : js_string t -> js_string t opt meth
  method setAttribute : js_string t -> js_string t -> unit
meth
  method removeAttribute : js_string t -> unit meth
  method hasAttribute : js_string t -> bool t meth
  method getElementsByTagName : js_string t -> element
nodeList t meth
  method attributes : attr namedNodeMap t readonly_prop
end
```

```
Js.js_string Js.t
```

（Js.js_string：JavaScriptの文字列）

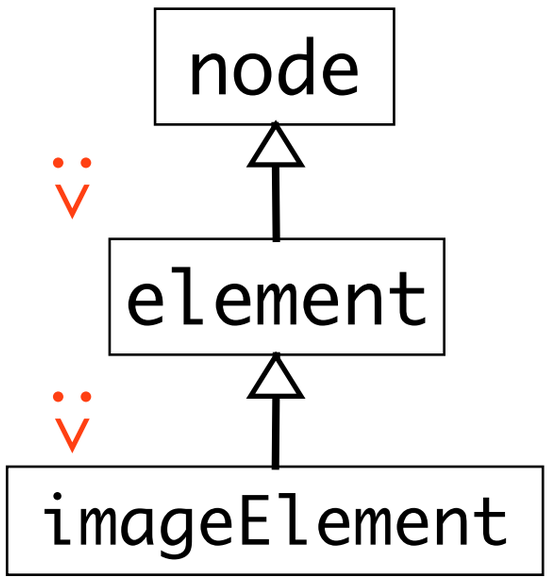
```
class type js_string = object
  method toString : js_string t meth
  method valueOf : js_string t meth
  method charAt : int -> js_string t me
  method charCodeAt : int -> float t me
  method concat : js_string t -> js_str
  method concat_2 : js_string t -> js_s
  method concat_3 :
    js_string t -> js_string t -> js_st
  method concat_4 :
    js_string t -> js_string t -> js_st
    js_string t meth
  method indexOf : js_string t -> int m
  method indexOf_from : js_string t ->
```

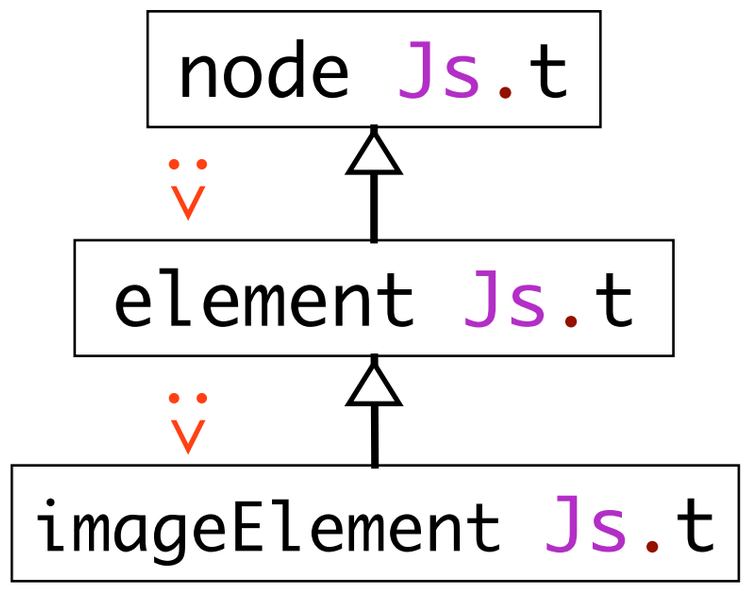
型パラメータと変位（variance）

- Js.t の型パラメータは 共変（＋(プラス)変位）

  （モジュールJsで `type +'a t` と定義されている）

  クラス間のサブタイプ関係が Js.t型でも有効に





JavaScriptオブジェクト型

- JavaScriptのメソッドとプロパティは全てOCamlの**メソッド**で表現される

```
interface Element : Node {
 readonly attribute DOMString tagName;
 DOMString  getAttribute(in DOMString name);
 void  setAttribute(in DOMString name,
                    in DOMString value)
                    raises(DOMException);
 void  removeAttribute(in DOMString name)
...
```

W3C DOMの Elementインタフェースの定義 (IDL)

```
class type element = object
  inherit node
  method tagName : js_string t readonly_prop
  method getAttribute : js_string t
                        -> js_string t opt meth
  method setAttribute : js_string t
                        -> js_string t
                        -> unit meth
  method removeAttribute : js_string t
                        -> unit meth
...
```

(Dom.element：DOM要素のクラス型)

```
interface Element : Node {
 readonly attribute DOMString tagName;
 DOMString  getAttribute(in DOMString name);
 void  setAttribute(in DOMString name,
                    in DOMString value)
                    raises(DOMException);
 void  removeAttribute(in DOMString name)
...
```

メソッドの戻り型で JSのプロパティ/メソッドを区別

```
class type element = object
  inherit node
  method tagName : js_string t readonly_prop
  method getAttribute : js_string t
                     -> js_string t opt meth
  method setAttribute : js_string t
                     -> js_string t
                     -> unit meth
  method removeAttribute : js_string t
                     -> unit meth
...
```

読み取り専用プロパティ

メソッド

メソッド

メソッド

メソッドの戻り値型による区別

戻り値型	種類	構文
`type +'a meth`	メソッド	`obj##meth(args)`
`type 'a readonly_prop`	読取専用 プロパティ	`obj##prop`
`type 'a writeonly_prop`	書込専用 プロパティ	`obj##prop <- exp`
`type 'a prop`	読み書き可能 プロパティ	`obj##prop`, `obj##prop <- exp`

JavaScriptのコンストラクタ

- JavaScriptのコンストラクタは `'a constr` 型の**値**

型	種類	型および構文の例
'a constr	コンストラクタ	`val regExp : (js_string t -> regExp t) constr` `jsnew regExp (js"[A-Za-z0-9_]*")`

名前替えによる疑似オーバーロード

- 複数の型シグネチャをもつメソッド：OCamlの型システムでは扱えない
  _（アンダースコア）とプレフィクスorサフィクスを付ける
  （プレフィクス/サフィクスは

例：Stringクラスのreplace: 第一引数が文字列or正規表現

```
method replace :
  regExp t     -> js_string t -> js_string t meth
method replace_string :
  js_string t -> js_string t -> js_string t meth
```

例：CanvasContext2D ： 補足情報を持つ場合と持たない場合

```
method drawImage :
  imageElement t -> float -> float -> unit meth
method drawImage_withSize :
  imageElement t -> float -> float -> float -> float -> unit meth
```

ヌルに対する安全性: OptとOptdefモジュール

- nullを返し得るメソッドはopt型

```
class type document = object

  method getElementById : js_string t -> element t opt meth
```

- undefinedになり得るプロパティはoptdef型

```
class type window = object

  method localStorage : storage t optdef readonly_prop
```

```
      Js.Opt.get : 'a opt -> (unit -> 'a) -> 'a
Js.Optdef.get : 'a optdef -> (unit -> 'a) -> 'a
```

を使って取り出す

トップレベルに入力してみよう

```
Js.Opt.get (Dom_html.document##getElementById(js"foobar"))
   (fun () -> failwith "element foobar not found");;
```

```
Js.Optdef.get (Dom_html.window##localStorage)
  (fun () -> failwith "localStorage is not supported");;
```

Js.Unsafe

- 文字通り、型安全でないプリミティブの集まり

```
-Js.Unsafe.variable : string -> 'a
```

任意のJavaScript変数にどんな型でも割り当てられる

例： `let jQuery : element -> jQuery = Js.Unsafe.variable "jQuery"`

（js_of_ocamlをjQueryで拡張する）

```
-Js.Unsafe.get : 'a -> 'b -> 'c
-Js.Unsafe.set : 'a -> 'b -> 'c -> unit
```

JavaScriptのプロパティ操作

例： `Js.Unsafe.set (elm##style) "webkitTransform" "translate(10px,10px)"`

（CSS3 のWebKit拡張を呼び出す）

```
-Js.Unsafe.fun_call  : 'a -> any array -> 'b
-Js.Unsafe.meth_call : 'a -> string -> any array -> 'b
```

型安全でない関数／メソッド呼び出し

Lwt

(Lightweight cooperative threads)

Lwtの動機

- OSネイティブなスレッドに対する不満があった
  （もともと、Js_of_ocamlとLwtは無関係だった）
  - 競合条件が厄介 - 「スレッドは人類には早すぎた」
  - スケールしない（メモリを多く消費）

➡ 軽量で、協調的なスレッドをOCamlで実装
（アトミックな処理に分離できる）

➡ Js_of_ocamlでも流用できる

類似のライブラリ：JaneStreet Coreの Asyncなど

Lwt＋Js_of_ocamlの恩恵

- クライアントサイドWeb：非同期処理

  （同期通信はブロックするため、UI記述に不向き）

- 「完了後の処理」をコールバックで記述→見通しが悪い！
  - 本来、逐次的なはずの処理が

    別々のコールバック関数に分断される

Lwtなら非同期通信を見通しよく記述できる

Unison（OCamlで書かれたファイル同期ツール）

でも使われている

Lwtは「約束」モナド

```
p : 'a Lwt.t
```

- 「型'aの値を持っています／（将来）計算します」
- `'a -> 'b Lwt.t` 型の関数と合成できる（モナド）
- HaskellのモナドやOCamlの`Lazy.t`との違い：
  - ✴ `(p >>= f)` は遅延評価**せず**、**すぐに**p,fを実行する
    pかfがサスペンド（Sleep）されるか、完了したら制御が戻る
  - ✴ よって、Lwt は評価済みであることもままある
- すぐ実行されるので、必ずしもpの中身は取り出さなくてよい
  （例： `unit Lwt.t`）し、実際ほとんどしない
  （一応、 `Lwt_main.run : 'a Lwt.t -> 'a`という関数はある）

Lwtを用いたAJAX通信

```
型 http_frame Lwt.t
```

```
XmlHttpRequest.get url >>= fun r ->
let msg = r.XmlHttpRequest.content in
Lwt.return msg
```

```
型 http_frame -> string Lwt.t
```

入力してみよう

```
let (>>=) = Lwt.(>>=);;
```

```
XmlHttpRequest.get "index.html" >>= (fun r ->
let msg = r.XmlHttpRequest.content in
print_endline msg;
Lwt.return ());;
```

Lwt_js.sleep

- トップレベルに入力してみよう

```
let (>>=) = Lwt.(>>=);;
```

```
let rec loop () =
  print_endline "Hi!";
  Lwt_js.sleep 1.0 >>= loop;;
```

```
let rec loop () =
  print_endline "Hi!";
  Lwt_js.sleep 1.0 >>= fun _ ->
  loop ();;
```

```
let t = loop ();;
```

```
Lwt.cancel t;;
```

演習：お絵描きプログラム

- ~/ocaml/material/canvas/canvas.ml の TODOを解消し、マウスで線を描画するようにして下さい

得意なこと、苦手なこと

- DOMとの相性はかなり良い（IDLが定義されているため）
- JQueryとの組み合わせには不向き
- O’Closure:
  - Google Closure widgetバインディング

Js_of_ocaml: まとめ

- 静的型で守られた世界でクライアントサイドWebが書ける
  - OCamlのオブジェクトシステムを流用し、JavaScriptの世界に型を導入できる
  - undefined/nullの可能性を表すopt型
  - 文字列型など、JavaScriptとOCamlで重複する型もあり、
    最初は少しストレスかもしれない
  - しかし、静的型付きの世界でプログラムが書ける信頼感は代え難い
- Lwt
  - 非同期通信のコールバック地獄を、同期的に扱いやすく