OCamlで構築するモダンWeb：
型付きHTML5プログラミングの実際

有限会社ITプランニング　今井 敬吾

IT Planning, Inc.

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

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

本チュートリアルでは

クライアントサイドWebと JavaScript・ HTML5

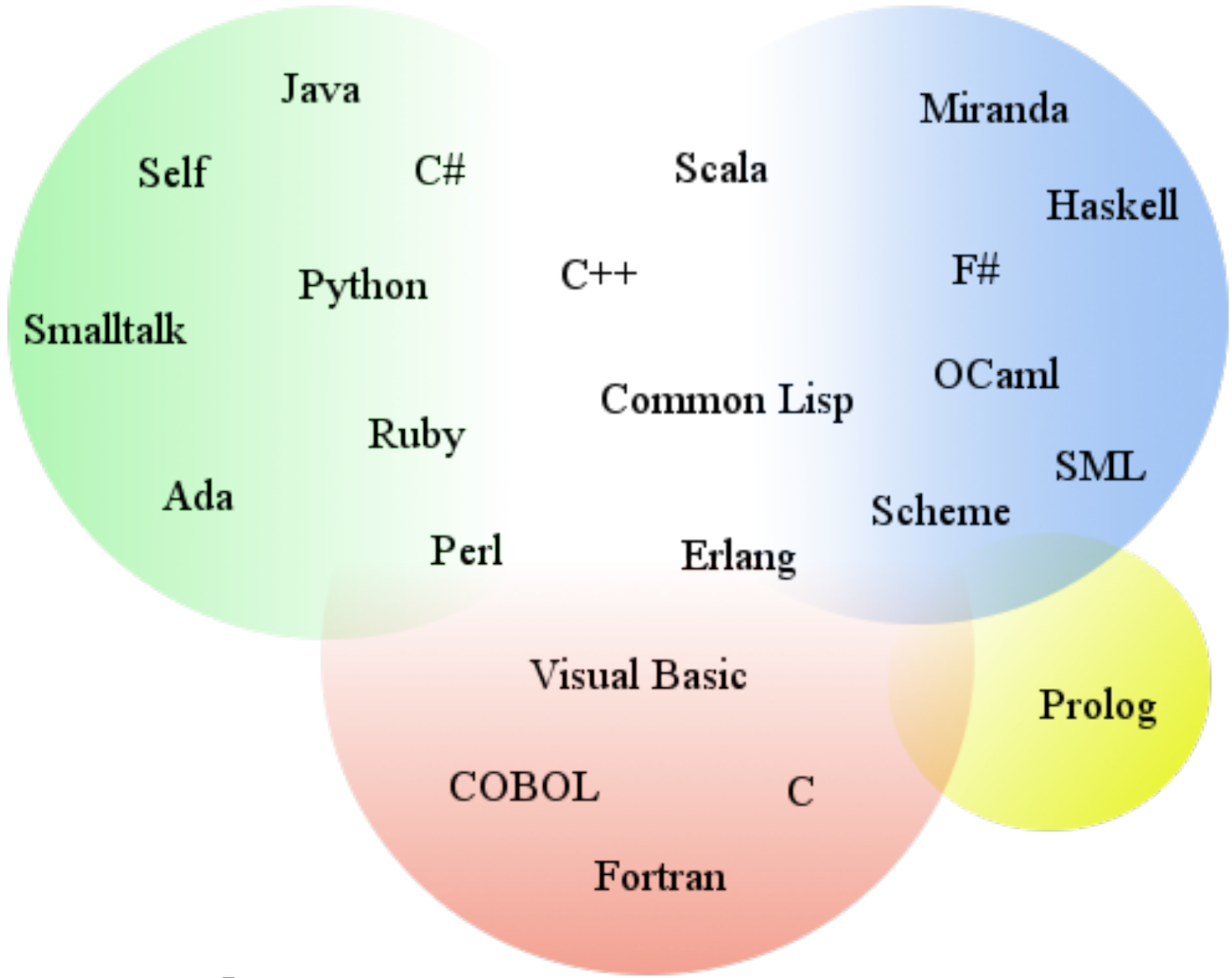
クライアントサイドWeb

新潮流：JavaScriptへのコンパイラ

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

OCaml

- マルチパラダイム言語



日本語の書籍

js_of_ocaml

OCaml製
クライアントサイドWeb

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

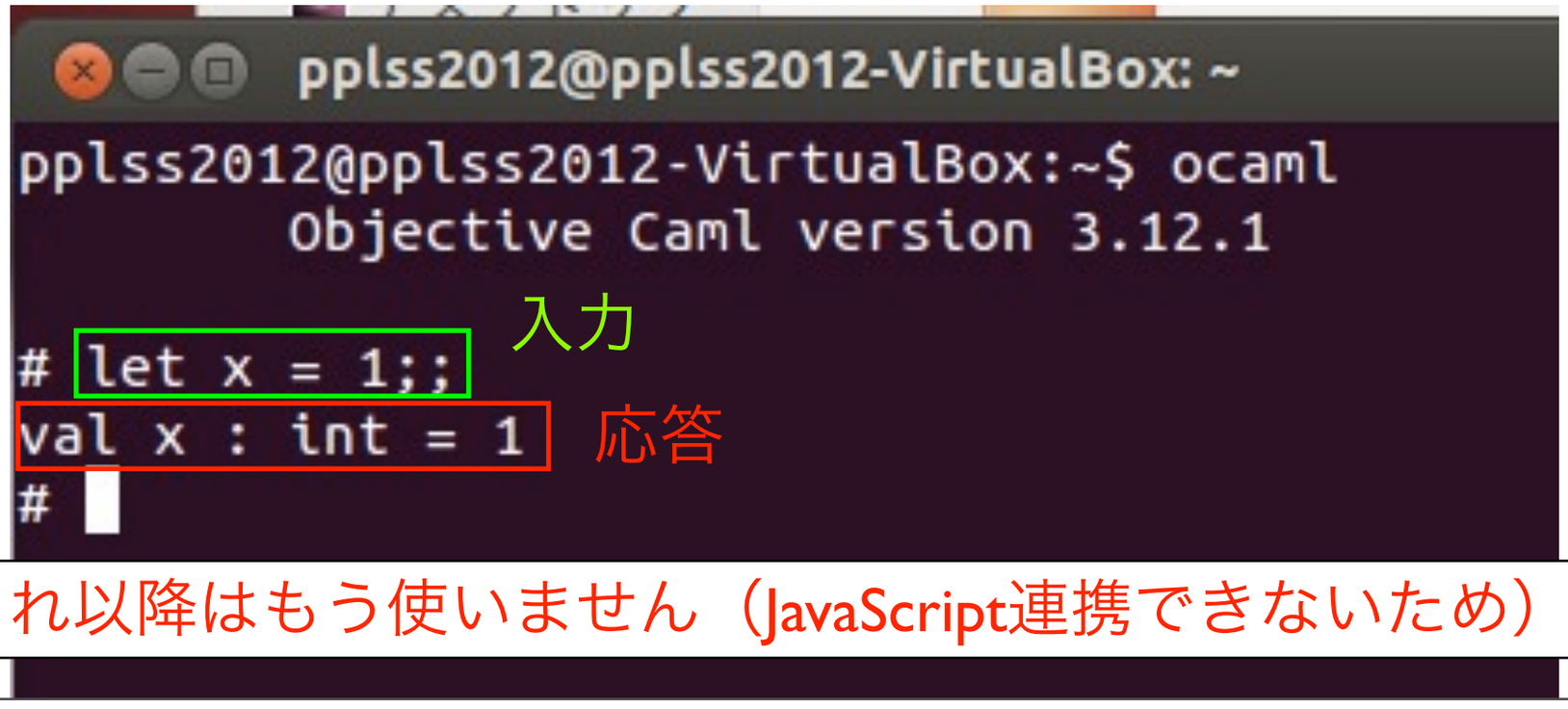
```
300. *. 1.05;;
```

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版トップレベル

- http://localhost:8082/ocaml/material/toplevel/index.html
  を開いて下さい

入力してみよう

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

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

OCamlからJavaScriptを呼ぶ

- 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

document	Dom_html.document
window	Dom_html.window

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

- 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

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

js_of_ocaml　　JavaScript

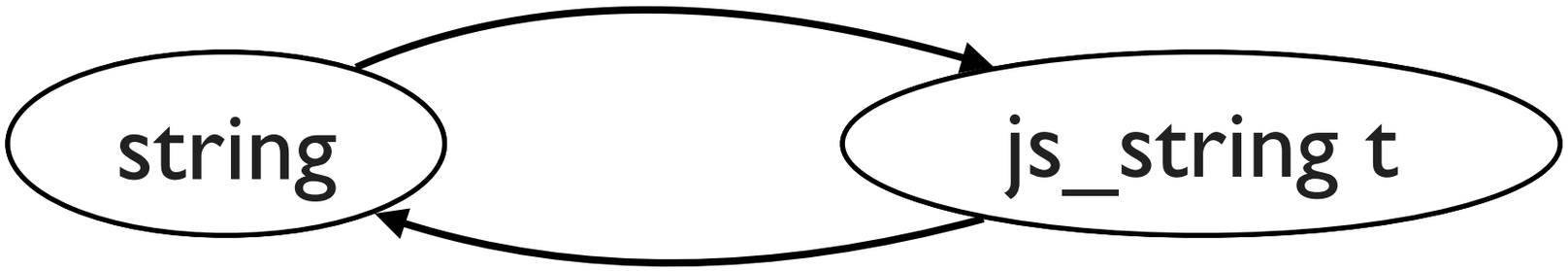
- コンストラクタの呼び出し構文 `jsnew constr (arg1,arg2,…)`

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

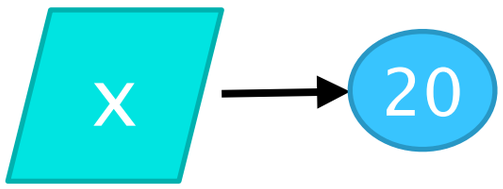
```
let js = Js.string;;
```

（js”ABC” のように少し短く書ける）

OCamlの基本：letによる定義

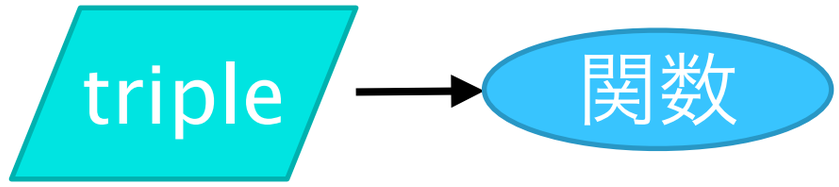
値の定義

```
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

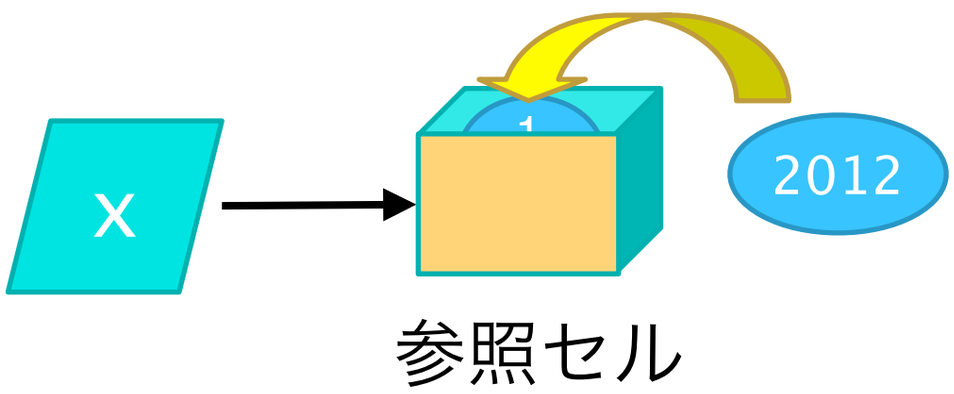
```
if password="ppl" then begin
  …
end else begin
  …
end
```

```
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 = ()
```

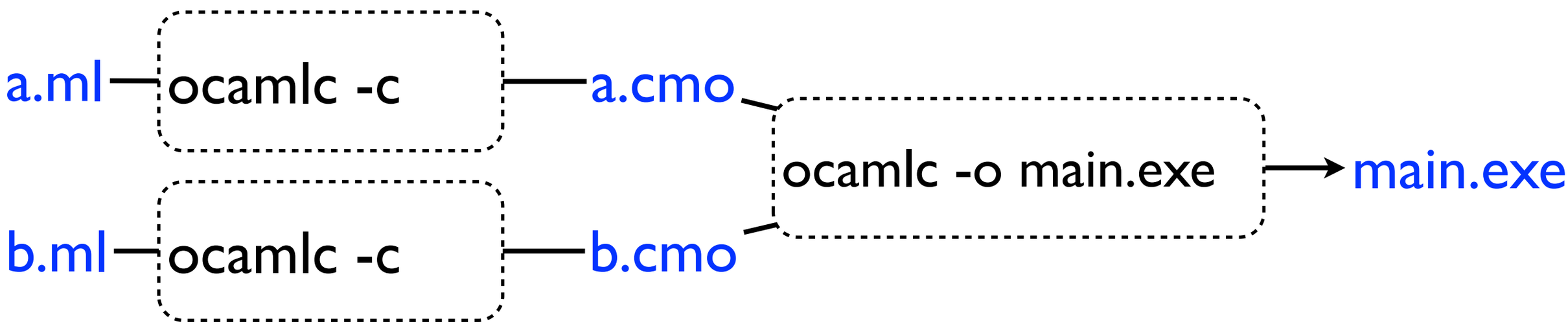
- セミコロンで式を逐次評価

```
# 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

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
- Typed_array
  WebGLで用いる高速なJavaScript配列
- 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

OCamlの
オブジェクトシステム

- Js_of_ocamlのオブジェクトに触れる前に、OCamlのオブジェクトシステムと、構造的多相性（structural polymorphism）の考え方に慣れ親しんで頂きます.

OCamlのオブジェクト

- 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);;
```

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
```

FAQ

- Q. オブジェクト型とクラス型の違いは？

  A. ほぼ同じ．例えば

```
type hello_typ = <hello : string>
```

と

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

は、ほぼ等価（後者は **#hello_typ** という型を使える点で異なる（後述））

（また~~type hello_typ = object … end~~とは書けない。）

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' >
```

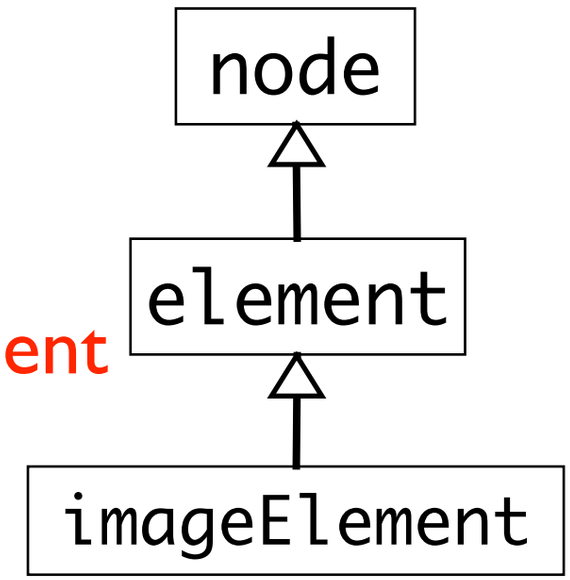
OCamlにおけるサブタイプ多相：コアーション（型強制）と #-型

- 明示的なアップキャストが必要
  - 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！

- OCamlの素晴らしい型推論は、JavaScriptらしい列多相をもつ型をうまく推論してくれる

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）

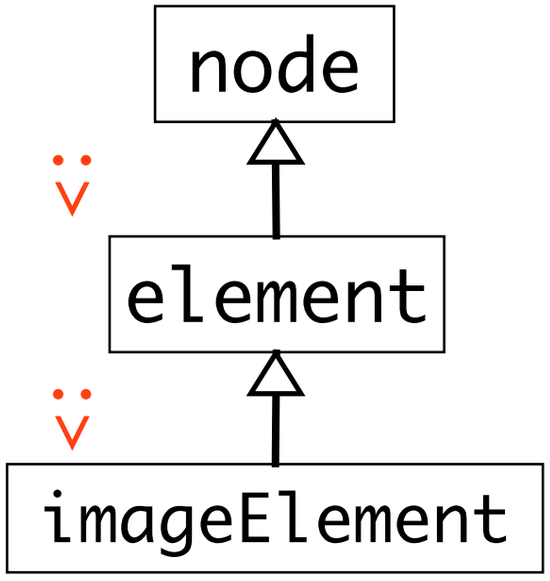
```
class type js_string = object
  method toString : js_string t meth
  method valueOf : js_string t meth
  method charAt : int -> js_string t me
  method charCodeAt : int -> float t me
  method concat : js_string t -> js_str
  method concat_2 : js_string t -> js_s
  method concat_3 :
    js_string 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 ->
  method lastIndexOf : js_string t -> i
```

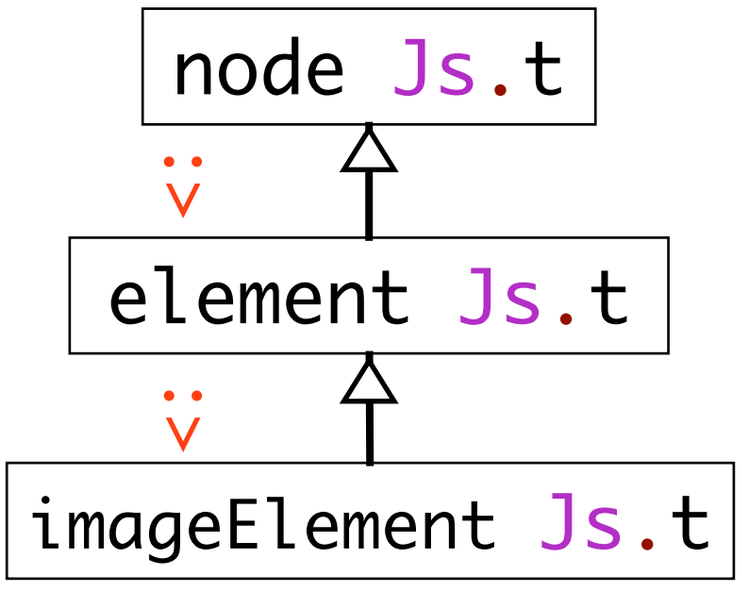
型パラメータと変位（variance）

- 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

メソッドの戻り型による区別

戻り型	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`	読書可能 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_]*")`

安全性への配慮：nullとundefinedの扱い

- 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
```

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

を使って取り出す

入力してみよう

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

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

Unsafe