PG104 - Programmation Fonctionnelle

D. Renault

ENSEIRB-Matmeca

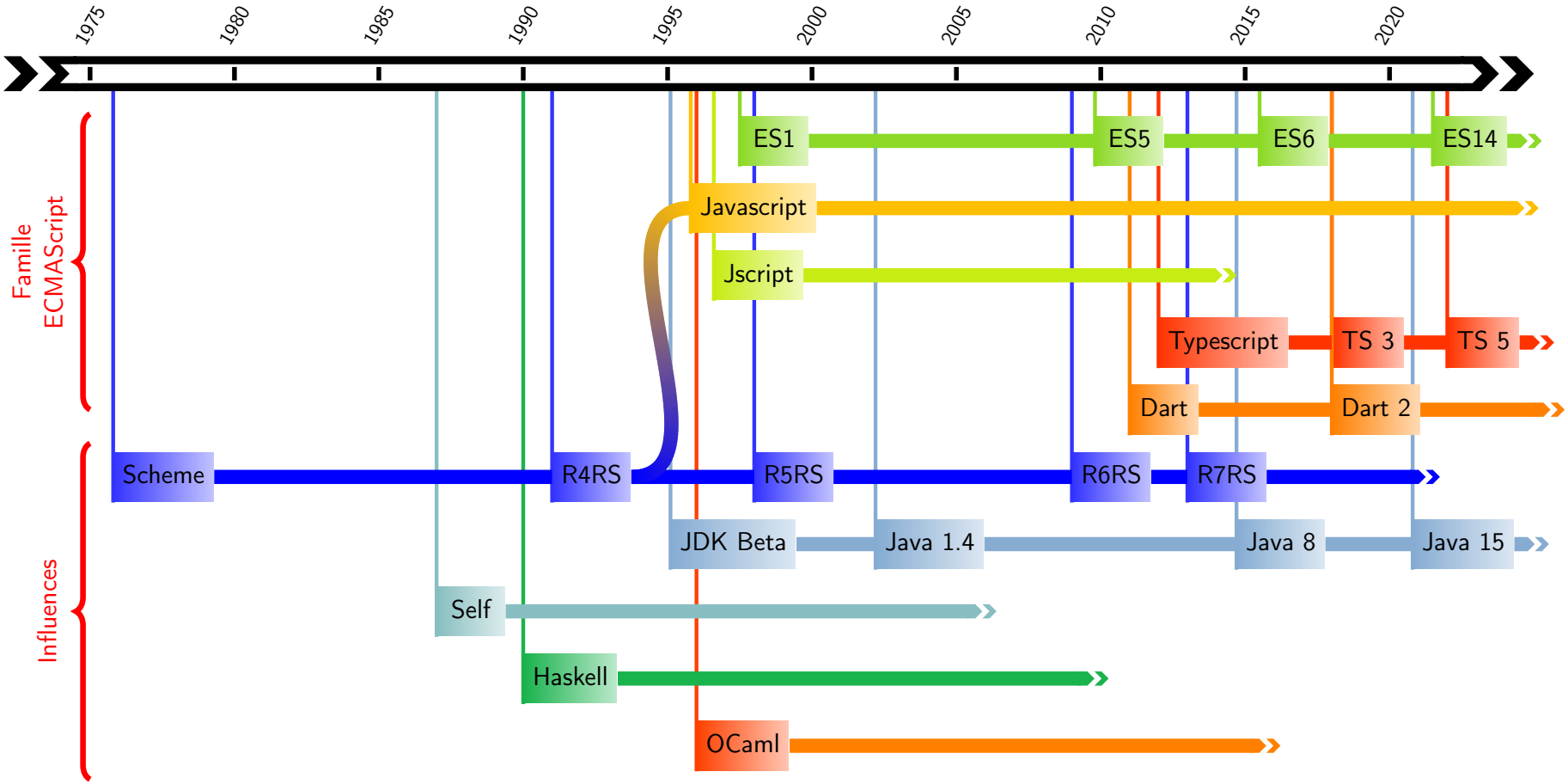
23 janvier 2025, v.1.2.5

https://www.labri.fr/perso/renault/working/teaching/functional/functional.php

Ecmascript

- Javascript est un langage de programmation créé en 1995 par Brendan Eich pour le navigateur Netscape.
- Influencé par Scheme (fonctionnel), Self (objet) et Java (syntaxe)
- Standardisé par l'ECMA International, à travers une spécification nommée **Ecmascript**.
- La version 6 de la norme (2015) est la modification majeure la plus récente, harmonisant les différentes implémentations précédentes.
- Le développement d'extensions au langage est incorporé chaque année dans une nouvelle révision de la norme.

Une brève histoire de Ecmascript



L'environnement Ecmascript

Les programmes Ecmascript s'utilisent de différentes manières :

- dans des navigateurs web, des scripts interagissent :
  - avec les pages HTML (à travers un arbre DOM),
  - avec les utilisateurs (à travers des événements),
  - avec des sites externes (avec par exemple Ajax ou Websocket)
- en tant que code autonome, exécuté par des moteurs comme Node.js,
- dans des bibliothèques de développement (Electron, Cordova), pour développer des clients lourds multi-plateformes.

L'environnement Node

L'environnement Node.js (`https://nodejs.org`) utilisé dans ce cours inclut :

- Le moteur d'exécution `node` qui sert aussi de REPL (acronyme de Read-Eval-Print Loop) ;
- Le gestionnaire de paquetages `npm` (`https://www.npmjs.com`) permettant d'installer des bibliothèques externes.

Exemples de bibliothèques encourageant la programmation fonctionnelle :

- `underscore` (`https://underscorejs.org`)
- `ramda` (`https://ramdajs.com`)
- `lodash` (`https://lodash.com`)

Philosophie de Ecmascript

- Inspiration objet forte et singulière

  (Langage à prototypes, les valeurs ont des méthodes . . .)

- Orientation dynamique très marquée

  (Typage dynamique, modification des méthodes des objets à la volée, introspection . . .)

- Tolérance aux erreurs considérable

  (Pas d'erreur si trop d'arguments, pas assez d'arguments, un champ manquant, un dépassement d'indice dans un tableau, pas d'erreurs arithmétiques)

Pour atténuer le peu de vérification dans le langage, Ecmascript 5 a introduit un **mode strict** qui transforme de nombreux comportements laxistes en erreurs.

Comparaison entre Ecmascript et le C

Le langage Ecmascript et le langage C, bien qu'éloignés, partagent des syntaxes très proches. Quelques différences notables de l'Ecmascript :

- Modèle d'exécution particulier

  (Interprétation par une machine virtuelle (e.g **V8**) générant du bytecode ou du code natif dans des phases de compilation "just-in-time")
- Typage dynamique

  (Pas d'indication de type devant les variables et les arguments, peu de vérification des types, nombreuses conversions implicites ...)
- Gestion de la mémoire automatique

  (Pas de `malloc` ou de `free`, allocation automatique, présence d'un ramasse-miettes)

... sans compter bien sûr tout ce qui a trait à la couche objet.

Ecmascript et la spécification

- La spécification du langage Ecmascript est accessible à :

  https://www.ecma-international.org/ecma-262

  Offre une référence complète, sinon digeste du langage.
- La documentation du langage par Mozilla est aussi une bonne source d'information :

  https://developer.mozilla.org/en-US/docs/Web/JavaScript

  https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference

Ecmascript et la vérification

- Les langages Ecmascript souffrent d'un fort manque de vérification : types, nombre d'arguments, conversions implicites ...
- Plusieurs langages de programmation proposent d'ajouter un compilateur pour plus de vérification statique (Dart, Typescript ...)

Typescript (https://www.typescriptlang.org), un langage développé par Microsoft ajoute des types optionnels et un compilateur vérifiant ces types.

```
function double(n : number) : number {
    return 2*n;
}
console.log(double("six"));
```

```
% tsc file.ts
error: Argument of type 'string'
       is not assignable to
       parameter of type 'number'.
```

Il sera discuté pendant les TDs et utilisé pendant les projets.

La syntaxe du langage

- Cette partie décrit le langage et la syntaxe d'Ecmascript.
- Les éléments du langage sont présentés un par un de manière succinte.
  - les types primitifs,
  - les objets,
  - les tableaux,
  - les fonctions,
  - et les structures de contrôle.
- Pour des vétérans du C, cette syntaxe est très accessible.

Les valeurs

La spécification Ecmascript décrit 8 types de valeurs :

- `undefined` pour les valeurs non initialisées et `null` pour les valeurs nulles, vides ou inexistantes ;
- `boolean` pour les valeurs booléennes ; `true`, `false`
- `string` pour les chaînes de caractères immutables et `symbol` pour les types des clés des dictionnaires ; `"aster"`
- `number` et `bigint` pour les valeurs numériques ; `3.141`, `8642n`
- et les objets.

Parmi les objets, on trouve en particulier :

- les dictionnaires clé-valeurs ; `{ name: "yoda" }`
- les tableaux ; `[ "un", "dos" ]`
- les fonctions. `function (n) return n+1;`

Les types primitifs en Ecmascript (1/2)

Définition (Type primitifs)

Les types primitifs sont tous les types de valeurs non-objets :

`boolean`, `string`, `symbol`, `number`, `bigint`

- Les types primitifs sont tous **immutables**.
- Les valeurs associées ne sont pas des objets (même si la syntaxe leur en donne l'apparence).
- Elles possèdent des **propriétés** et des **méthodes**.

Les types primitifs en Ecmascript (2/2)

Exemple : les chaînes de caractères

Toute valeur de type `string` possède :

- une propriété `length` ;
- 32 méthodes (dont `concat`, `match`, `replace`, `search`, `substring` ...)

```
let s = "archeopteryx";
s[0];                 // → "a" (returns a string)
s[0] = "z";           // no error
s;                    // → "archeopteryx" (s is unchanged since it is immutable)

s.length;             // → 12
s.substring(2, 7);    // → "cheop"
s.match(/c.*p/);      // → [ 'cheop', index: 2 (...) ]
```

En 1ère approximation, une méthode est une fonction prenant un paramètre privilégié et utilisant une syntaxe particulière.

Les objets en Ecmascript (1/2)

- Les objets sont, en 1ère approximation, des **dictionnaires** clé-valeur.

```
let o = {};          // → { }
o.firstName = "Abe";
o.lastName  = "Lincoln";
o;                   // → { firstName: 'Abe', lastName: 'Lincoln' }
o['firstName'];      // → "Abe"
```

- Les objets sont **mutables** et **itérables**.

```
o.firstName = "Abraham";
o;                   // → { firstName: 'Abraham', lastName: 'Lincoln' }
Object.keys(o);      // → [ 'firstName', 'lastName' ]
Object.keys(o).forEach( (p) => { o[p] = o[p] + "␣(updated)"; });
o; // { firstName: 'Abraham (updated)', lastName: 'Lincoln (updated)' }
```

Les objets en Ecmascript (2/2)

- Chaque objet possède une propriété particulière : son **prototype**.
- Le prototype contient l'ensemble des **méthodes** associées à cet objet :

```
let o = {};
o.__proto__;  // → [Object: null prototype] {}
Object.getOwnPropertyNames(o.__proto__); //→[.., 'toString', 'valueOf', ..]
```

- Le prototype est modifiable à volonté :

```
String.prototype.halfLength = function () {
  return Math.floor(this.length / 2); };
new String("abraracourcix").halfLength(); // → 6
```

- Une variable particulière (`this`) fait référence à l'objet courant.

Les tableaux

- Les tableaux sont des objets particuliers.
- Les clés de ces objets sont les indices utilisés du tableau.

```
const fruits = ["Apple", "Banana"]; // only the reference is const
fruits[0];      // → 'Apple'
fruits.length; // → 2
```

- Rien n'oblige les indices à être consécutifs.
  Rien n'oblige les éléments à être du même type.

```
fruits[4] = "Coconut";
fruits;         // [ 'Apple', 'Banana', <2 empty items>, 'Coconut' ]
fruits[2] = 666;
fruits;         // [ 'Apple', 'Banana', 666, <1 empty item>, 'Coconut' ]
```

Les fonctions

- Les fonctions sont des objets particuliers.
- Il existe deux manières différentes de les construire :
  - les fonctions classiques (*regular functions*)

    Version nommée :

    ```
    function s(x) { return x+1; }
    ```

    Version **anonyme** :

    ```
    let s = function (y) { return y-1; }
    ```

  - les fonctions fléchées (*arrow functions*)

    Version standard :

    ```
    (x) ⇒ { return x*2; }
    ```

    Version avec retour implicite :

    ```
    (y) ⇒ 3*y;
    ```

- Les différences entre les 2 constructions sont mineures dans ce cours.

Par exemple, les fonctions classiques peuvent accéder à une variable `arguments` :

```
function dispArgs() { return arguments; }
dispArgs(1, 2, "trois"); // → [Arguments] { '0': 1, '1': 2, '2': 'trois' }
```

De l'importance des blocs

Définition (Bloc)

Un **bloc** est une construction syntaxique particulière délimitant une zone contigüe de code (typiquement entre deux accolades).

```
function aFunction(a, b, c) {
  if (b*b - 4*a*c >= 0) {
    return "real";
  } else {
    return "complex";
} }
```

Bloc "if-true"

```
let anArr = [1,2,3,4,5];
for (let i=0; i<anArr.length; i++) {
  let j = i*i;
  console.log(anArr[i] + j);
}
console.log(anArr);
```

Bloc "for-loop"

Les fonctions acquièrent ainsi un statut spécial : ce sont des blocs **nommés** (donc transportables et réutilisables) et **paramétrés**.

Structures de contrôle

La syntaxe Ecmascript inclut les structures de contrôle suivantes :

- Les expressions conditionnelles :

```
if ⟨bool⟩ {
  ⟨block-true⟩
} else {
  ⟨block-false⟩
}
```

- Les aiguillages (*switch*) :

```
switch ⟨expr⟩ {
case ⟨val⟩:
  ⟨code-ending-with-break⟩
default:
  ⟨code-dflt⟩
}
```

- Les boucles bornées :

```
for ( ⟨init⟩; ⟨test⟩; ⟨next⟩ ) {
  ⟨block-loop⟩
}
```

- Les boucles non bornées :

```
while ( ⟨test⟩ ) {
  ⟨block-loop⟩
}
```

Noter la décomposition en blocs variant selon la structure de contrôle.

Portée des données

Considérons un programme et réfléchissons à la question suivante :

> Qui a accès à quoi ?

- Notion de visibilité, de portée, ou de droits d'accès ;
- Mécanismes multiples selon les langages de programmation ;

Une gestion soignée des visibilités permet de nombreuses techniques logicielles, généralement basées sur la notion d'**abstraction**.

Définition (Abstraction)

Propriété logicielle selon laquelle les composants séparent leur part publique (interface) de leur part privée (implémentation).

Variables et liaisons

Définition (Variable)

Une **variable** est une association entre un nom (son identifiant) et un emplacement stockant une valeur lors de l'exécution (sa valeur).

Pour créer une variable, il suffit de la déclarer :

```
let aVar = "X"; // example of variable named 'aVar' and storing the value "X"
```

- Une variable est une **liaison** (*binding*) entre un nom et une valeur.
- Le mot "variable" est parfois mal venu, la liaison pouvant être constante.
- A fur et à mesure de l'exécution, l'ensemble des variables et les valeurs qu'elles contiennent constituent un **environnement**.

Portée (lexicale)

Définition (Portée)

La **portée** (*scope*) d'un identifiant dans un code donné est la région du code dans laquelle la variable associée est accessible dans l'environnement.

La **portée lexicale** (*lexical scope*) d'un identifiant débute au moment de sa déclaration et se termine à la fin du bloc dans lequel il est défini.

```
function foo(t) {
  return 2*t;
}
function baz(z) {
  console.log(z);
  let t = 42;
  return foo(z+t);
}
baz(10);
```

Portée (lexicale)

Définition (Portée)

La **portée** (*scope*) d'un identifiant dans un code donné est la région du code dans laquelle la variable associée est accessible dans l'environnement.

La **portée lexicale** (*lexical scope*) d'un identifiant débute au moment de sa déclaration et se termine à la fin du bloc dans lequel il est défini.

```
function foo(t) {
  return 2*t;
}
function baz(z) {
  console.log(z);
  let t = 42;
  return foo(z+t);
}
baz(10);
```

Portée lexicale de *t*

Portée (lexicale)

Définition (Portée)

La **portée** (*scope*) d'un identifiant dans un code donné est la région du code dans laquelle la variable associée est accessible dans l'environnement.

La **portée lexicale** (*lexical scope*) d'un identifiant débute au moment de sa déclaration et se termine à la fin du bloc dans lequel il est défini.

```
function foo(t) {
  return 2*t;
}
function baz(z) {
  console.log(z);
  let t = 42;
  return foo(z+t);
}
baz(10);
```

Une autre portée

Portée lexicale de *t*

Remarque

Si deux identifiants sont les mêmes, celui dans le bloc courant **masque** l'autre (*variable shadowing*).

La norme Ecmascript 6 propose deux manières de déclarer des variables avec une portée **lexicale** :

- `let` pour une variable (la liaison dans l'environnement est mutable)
- `const` pour une constante (la liaison dans l'environnement est constante)

Le langage autorise aussi de déclarer des variables avec la syntaxe suivante :

- `var` pour une variable à la portée non lexicale

Mais ceci permet entre autres d'utiliser une variable **avant** sa déclaration.

Principe de localité des variables

Dans ce cours, et afin de mieux contrôler les visibilités, on se limitera à n'utiliser des variables qu'avec une portée **lexicale**.

Exemples de portées lexicales

Deux fonctions indépendantes

```
let g = 1;
function foo() {
  let h = 2;
}
function baz() {
  let i = 3;
  foo();
}
baz();
```

Une boucle dans une fonction

```
let g = 1;
function loop() {
  let h = 2;
  for (let i=3; i<4; i++) {
    console.log(g+h+i);
  }
}
loop();
```

Une liaison masquant une autre liaison

```
let g = 1;
function foo() {
  let g = 2;
  function baz() {
    let g = 3;
  }
  baz();
}
foo();
```

Une fonction renvoyant une fonction

```
function foo() {
  let h = [];
  function baz() {
    h.push(1);
    return h;
  }
  return baz;       // not a typo
}
let f = foo();
f();
```

Exemples de portées lexicales

Deux fonctions indépendantes

```
let g = 1;              // g
function foo() {        // g
  let h = 2;            // g, h
}
function baz() {        // g
  let i = 3;            // g, i
  foo();
}
baz();                  // g
```

Une boucle dans une fonction

```
let g = 1;
function loop() {
  let h = 2;
  for (let i=3; i<4; i++) {
    console.log(g+h+i);
  }
}
loop();
```

Une liaison masquant une autre liaison

```
let g = 1;
function foo() {
  let g = 2;
  function baz() {
    let g = 3;
  }
  baz();
}
foo();
```

Une fonction renvoyant une fonction

```
function foo() {
  let h = [];
  function baz() {
    h.push(1);
    return h;
  }
  return baz;       // not a typo
}
let f = foo();
f();
```

Exemples de portées lexicales

Deux fonctions indépendantes

```
let g = 1;              // g
function foo() {        // g
  let h = 2;            // g, h
}
function baz() {        // g
  let i = 3;            // g, i
  foo();
}
baz();                  // g
```

Une boucle dans une fonction

```
let g = 1;                          // g
function loop() {                   // g
  let h = 2;                        // g, h
  for (let i=3; i<4; i++) {         // g, h, i
    console.log(g+h+i);             // g, h, i
  }
}
loop();                             // g
```

Une liaison masquant une autre liaison

```
let g = 1;
function foo() {
  let g = 2;
  function baz() {
    let g = 3;
  }
  baz();
}
foo();
```

Une fonction renvoyant une fonction

```
function foo() {
  let h = [];
  function baz() {
    h.push(1);
    return h;
  }
  return baz;       // not a typo
}
let f = foo();
f();
```

Exemples de portées lexicales

Deux fonctions indépendantes

```
let g = 1;              // g
function foo() {        // g
  let h = 2;            // g, h
}
function baz() {        // g
  let i = 3;            // g, i
  foo();
}
baz();                  // g
```

Une boucle dans une fonction

```
let g = 1;                          // g
function loop() {                   // g
  let h = 2;                        // g, h
  for (let i=3; i<4; i++) {         // g, h, i
    console.log(g+h+i);             // g, h, i
  }
}
loop();                             // g
```

Une liaison masquant une autre liaison

```
let g = 1;              // g_1
function foo() {
  let g = 2;            // g_2
  function baz() {
    let g = 3;          // g_3
  }
  baz();                // g_2
}
foo();                  // g_1
```

Une fonction renvoyant une fonction

```
function foo() {
  let h = [];
  function baz() {
    h.push(1);
    return h;
  }
  return baz;       // not a typo
}
let f = foo();
f();
```

Exemples de portées lexicales

Deux fonctions indépendantes

```
let g = 1;              // g
function foo() {        // g
  let h = 2;            // g, h
}
function baz() {        // g
  let i = 3;            // g, i
  foo();
}
baz();                  // g
```

Une boucle dans une fonction

```
let g = 1;                          // g
function loop() {                   // g
  let h = 2;                        // g, h
  for (let i=3; i<4; i++) {         // g, h, i
    console.log(g+h+i);             // g, h, i
  }
}
loop();                             // g
```

Une liaison masquant une autre liaison

```
let g = 1;              // g_1
function foo() {
  let g = 2;            // g_2
  function baz() {
    let g = 3;          // g_3
  }
  baz();                // g_2
}
foo();                  // g_1
```

Une fonction renvoyant une fonction

```
function foo() {    //
  let h = [];       // h (an array)
  function baz() { // h
    h.push(1);      // h
    return h;       // h
  }
  return baz;       // h
}
let f = foo();      // f
f();                // f (returns h = [1])
```

Portée

A titre culturel, noter qu'il existe en programmation (et en Ecmascript) d'autres formes de portée des variables :

- la portée **globale** — L'identifiant est accessible à partir de son point de définition et dans toute la suite du code.
- la portée **dynamique** — L'identifiant possède une pile de liaisons différentes qui évolue au cours de l'exécution.
- la portée **fonction** — L'identifiant déclaré au milieu d'une fonction est accessible dans toute la fonction, même au début.

Même si ces notions ont chacun leur histoire et leurs avantages (et possiblement inconvénients), elles sortent du cadre de ce cours.

Durée de vie

Autre question naturelle concernant les données dans un programme :

Qui vit combien de temps ?

Définition (Durée de vie)

La **durée de vie** (*lifetime*) d'une valeur est l'intervalle de temps pendant l'exécution du programme où cette valeur est accessible.

Gestion de la durée de vie

La durée de vie des valeurs peut-être gérée de différentes manières :

- En les plaçant dans la **pile** (*stack*), la gestion est **automatique** : elles terminent leur vie au dépilement ou survivent par copie.

  Ex : allocation par défaut en C, types primitifs en Ecmascript (V8)

- En les plaçant dans le **tas** (*heap*), la gestion peut alors se faire :
  - en allouant et désallouant la mémoire de manière **manuelle**,

    Ex : allocation dynamique en C avec `malloc` et `free`

  - ou en délaissant cette gestion à un système **automatique** comme un **ramasse-miettes** (*garbage collector*).

    Ex : allocation des objets en Ecmascript (V8), gérés comme des références

Exemple de durée de vie

La fonction `countUnique` compte le nombre d'éléments distincts dans un tableau de nombres :

```
function countUnique(anArray) {
  let aSet = setEmpty();
  anArray.forEach((anElem) ⇒ {
      setAdd(aSet, anElem);
  });
  return setSize(aSet);
}
countUnique([1,2,3,1,2,3]); //→ 3
```

Exemple de durée de vie

La fonction `countUnique` compte le nombre d'éléments distincts dans un tableau de nombres :

```
function countUnique(anArray) {
  let aSet = setEmpty();
  anArray.forEach((anElem) ⇒ {
      setAdd(aSet, anElem);
  });
  return setSize(aSet);
}
countUnique([1,2,3,1,2,3]); //→ 3
```

```
function setEmpty() {
  return {};
}
function setSize(aSet) {
  return Object.keys(aSet).length;
}
function setAdd(aSet, anElem) {
  aSet[anElem] = 1; // dummy val
}
```

Exemple de durée de vie

La fonction `countUnique` compte le nombre d'éléments distincts dans un tableau de nombres :

```
function countUnique(anArray) {
  let aSet = setEmpty();
  anArray.forEach((anElem) ⇒ {
      setAdd(aSet, anElem);
  });
  return setSize(aSet);
}
countUnique([1,2,3,1,2,3]); //→ 3
```

```
function setEmpty() {
  return {};
}
function setSize(aSet) {
  return Object.keys(aSet).length;
}
function setAdd(aSet, anElem) {
  aSet[anElem] = 1; // dummy val
}
```

La variable stockée dans `aSet` est :

- allouée dans `setEmpty`,
- est transmise et mise à jour lors des appels à `setAdd` et `setSize`,
- et finalement détruite au retour de `countUnique`.

Exemple de manipulation des portées (1/2)

Considérons l'exemple d'un générateur pseudo-aléatoire (Lehmer) :

```
let seed = 12345;
const m = 65537; const a = 75;
function rand() {
  seed = (a * seed) % m;
  return seed;
};
```

Si on répète la fonction `rand()` 10 fois, on obtient :

$$8357, 36942, 18096, 46460, 11039, 41481, 30836, 18905, 41598, 39611$$

Problèmes :
- la variable `seed` est globale, tout le code peut la modifier ;
- les constantes du générateur sont publiques.

Solution :
- limiter les portées en encapsulant le code dans un bloc.

Exemple de manipulation des portées (2/2)

Encapsulons les paramètres, tout en laissant la fonction `rand` accessible :

```
let seed = 12345;
const m = 65537; const a = 75;
function rand() {
  seed = (a * seed) % m;
  return seed;
};
```

$\Rightarrow$

Exemple de manipulation des portées (2/2)

Encapsulons les paramètres, tout en laissant la fonction `rand` accessible :

```
let seed = 12345;
const m = 65537; const a = 75;
function rand() {
  seed = (a * seed) % m;
  return seed;
};
```

$\Rightarrow$

```
{
  let seed = 12345;
  const m = 65537; const a = 75;
  function rand() {
    seed = (a * seed) % m;
    return seed;
  };

}
```

Exemple de manipulation des portées (2/2)

Encapsulons les paramètres, tout en laissant la fonction `rand` accessible :

```
let seed = 12345;
const m = 65537; const a = 75;
function rand() {
  seed = (a * seed) % m;
  return seed;
};
```

$\Rightarrow$

```
function makeRng() {
  let seed = 12345;
  const m = 65537; const a = 75;
  function rand() {
    seed = (a * seed) % m;
    return seed;
  };

}
```

Exemple de manipulation des portées (2/2)

Encapsulons les paramètres, tout en laissant la fonction `rand` accessible :

```
let seed = 12345;
const m = 65537; const a = 75;
function rand() {
  seed = (a * seed) % m;
  return seed;
};
```

⇒

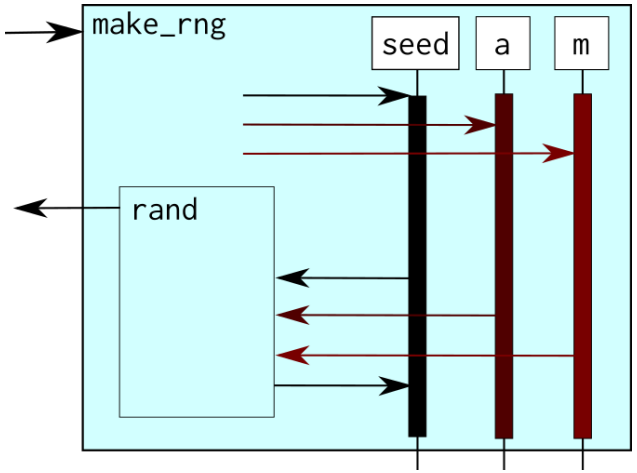
```
function makeRng() {
  let seed = 12345;
  const m = 65537; const a = 75;
  function rand() {
    seed = (a * seed) % m;
    return seed;
  };
  return rand;
}
let rand = makeRng();
```

Les variables `seed`, `a` et `m` sont locales à la fonction `makeRng`.
Néanmoins, elles restent **accessibles** à la fonction `rand`, hors de `makeRng`.

Fermetures

Définition (Fermeture)

Une **fermeture** (*closure*) consiste en la donnée d'une fonction, ainsi que de l'environnement nécessaire à son fonctionnement.



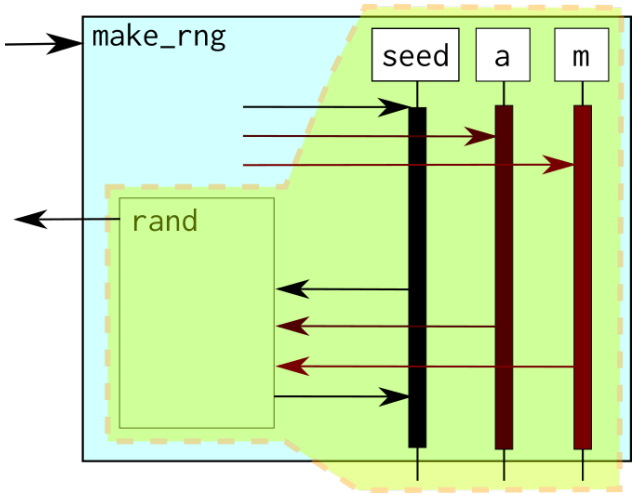
- Grâce aux fermetures, les fonctions deviennent des entités autonomes.
- Grâce aux portées, les fonctions ont une séparation public/privé.

Ces propriétés font des fonctions des petites unités de code portables.

Fermetures

Définition (Fermeture)

Une **fermeture** (*closure*) consiste en la donnée d'une fonction, ainsi que de l'environnement nécessaire à son fonctionnement.



- Grâce aux fermetures, les fonctions deviennent des entités autonomes.
- Grâce aux portées, les fonctions ont une séparation public/privé.

Ces propriétés font des fonctions des petites unités de code portables.

Conclusion sur la vie des données

- Un des aspects importants de la programmation consiste à bien comprendre les **dépendances** entre les entités d'un code.
- La notion de **portée** (et d'accessibilité) est une notion centrale pour comprendre les dépendances.
- Les fonctions, à travers les mécanismes de fermeture et de portée, permettent de contrôler ces dépendances.
- En cela, elles permettent un style de programmation basé sur l'indépendance des calculs.

Prochain chapitre, la **pureté**, une qualité pour limiter les dépendances.

※ Exercice : quel style utiliser ? (1/2)

Comparons les deux écritures d'une fonction très simple :

```
function fact(n) {
  if (n <= 1)
    return 1;
  else
    return n * fact(n-1);
}
```

```
function fact(n) {
  if (n <= 1)
    return 1; // Early return
  // ←---------- Nothing here !
  return n * fact(n-1);
}
```

Quel style adopter ?

※ Exercice : quel style utiliser ? (1/2)

Comparons les deux écritures d'une fonction très simple :

```
function fact(n) {
  if (n <= 1)
    return 1;
  else
    return n * fact(n-1);
}
```

```
function fact(n) {
  if (n <= 1)
    return 1; // Early return
  // ←---------- Nothing here !
  return n * fact(n-1);
}
```

Quel style adopter ?

- Le `if-then-else` est une expression* ;
- Il s'agit d'un style **fonctionnel**, proche de la définition mathématique ;
- Le code est explicite sur le fait que les deux cas forment une disjonction ;
- Un compilateur pourrait utiliser cette information pour **appliquer des optimisations**.

※ Exercice : quel style utiliser ? (1/2)

Comparons les deux écritures d'une fonction très simple :

```
function fact(n) {
  if (n <= 1)
    return 1;
  else
    return n * fact(n-1);
}
```

```
function fact(n) {
  if (n <= 1)
    return 1; // Early return
  // ←---------- Nothing here !
  return n * fact(n-1);
}
```

Quel style adopter ?

- Le `if-then-else` est une expression* ;
- Il s'agit d'un style **fonctionnel**, proche de la définition mathématique ;
- Le code est explicite sur le fait que les deux cas forment une disjonction ;
- Un compilateur pourrait utiliser cette information pour **appliquer des optimisations**.

- Le `if-then` est une instruction ;
- Il s'agit d'un style **impératif**, parfois aussi appelé un **early return** ;
- Le code est implicite sur le fait que les deux cas forment une disjonction ;
- Un compilateur sera gêné pour optimiser ;
- La fonction est **plus lisible**.

Il n'y a pas de réponse évidente.

※ Exercice : quel style utiliser ? (2/2)

Quelles types d'optimisations un compilateur pourrait-il appliquer ?
⇒ principalement des vérifications par **analyse statique**.

Exemple : les discriminated unions en Typescript

```
interface Circle { kind: "circle"; radius: number; }
interface Square { kind: "square"; sideLength: number; }

type Shape = Circle | Square;

function getArea(shape: Shape) {
  switch (shape.kind) {
    case "circle":
      return Math.PI * shape.radius ** 2;
    case "square":
      return shape.sideLength ** 2;
  }
}
```

Un `Circle` a un attribut `radius` contrairement à un `Square`.

Un `Shape` est soit un `Circle`, soit un `Square`.

Dans ce bloc, le compilateur sait que `shape` possède un champ `radius` contrairement au bloc suivant.

Il peut le faire car la distinction entre les deux cas est explicite.

Cf. le **pattern matching** des langages comme OCaml et Haskell.

Quelques lectures . . .

Fogus, M.: Functional Javascript.
O'Reilly, 2013.

Narbel, P.: Programmation fonctionnelle, générique et objet : Une introduction avec le langage OCaml.
Vuibert, 2005.

Reade, C.: Elements of Functional Programming.
Addison-Wesley, 1989.

Bird, R. et P. Wadler: Introduction to Functional Programming.
Prentice Hall, 1988.

Okasaki, C.: Purely functional data structures.
Cambridge University Press, 1999.

Narbel, P.: Techniques Avancées de Programmation.
Cours de Master 2 à l'Université de Bordeaux.