# javascript :

# éléments du langage

# au programme...

# au programme...

# javascript

présentation partielle, et parfois partiale

# javascript

présentation partielle, et parfois partiale

## Javascript

un langage fonctionnel à objet à base de prototypes

# javascript

présentation partielle, et parfois partiale

## Javascript

un langage fonctionnel à objet à base de prototypes

oui, mais encore...

- un langage de scripts, interprété

- un langage de scripts, interprété
  - les scripts peuvent être placés dans les documents HTML

- un langage de scripts, interprété
  - les scripts peuvent être placés dans les documents HTML
  - le navigateur possède un interprète javascript

- un langage de scripts, interprété
  - les scripts peuvent être placés dans les documents HTML
  - le navigateur possède un interprète javascript
- le code javascript permet

- un langage de scripts, interprété
  - les scripts peuvent être placés dans les documents HTML
  - le navigateur possède un interprète javascript
- le code javascript permet
  - d'agir sur les propriétés des éléments d'un document HTML

- un langage de scripts, interprété
  - les scripts peuvent être placés dans les documents HTML
  - le navigateur possède un interprète javascript
- le code javascript permet
  - d'agir sur les propriétés des éléments d'un document HTML
  - de manipuler l'arbre DOM

- un langage de scripts, interprété
  - les scripts peuvent être placés dans les documents HTML
  - le navigateur possède un interprète javascript
- le code javascript permet
  - d'agir sur les propriétés des éléments d'un document HTML
  - de manipuler l'arbre DOM
  - $\Rightarrow$ dynamicité du document affiché

- un langage de scripts, interprété
  - les scripts peuvent être placés dans les documents HTML
  - le navigateur possède un interprète javascript
- le code javascript permet
  - d'agir sur les propriétés des éléments d'un document HTML
  - de manipuler l'arbre DOM
  - $\Rightarrow$ dynamicité du document affiché

exemples : *diaporama* – *extraction couleurs*

# intégration de javascript dans la page html

on utilise l'élément `script`, dans le corps ou l'entête du document.
2 cas de figure possibles :

# intégration de javascript dans la page html

on utilise l'élément `script`, dans le corps ou l'entête du document.
2 cas de figure possibles :

- code javascript directement placé dans le corps du fichier html :

# intégration de javascript dans la page html

on utilise l'élément `script`, dans le corps ou l'entête du document.
2 cas de figure possibles :

- code javascript directement placé dans le corps du fichier html :

```
<script type="text/javascript">
...  code javascript ici
</script>
```

# intégration de javascript dans la page html

on utilise l'élément `script`, dans le corps ou l'entête du document.
2 cas de figure possibles :

- code javascript directement placé dans le corps du fichier html :

```
<script type="text/javascript">
...  code javascript ici
</script>
```

- code javascript dans un fichier séparé précisé par l'attribut `src` de l'élément `script` :

# intégration de javascript dans la page html

on utilise l'élément `script`, dans le corps ou l'entête du document.
2 cas de figure possibles :

- code javascript directement placé dans le corps du fichier html :

```
<script type="text/javascript">
...  code javascript ici
</script>
```

- code javascript dans un fichier séparé précisé par l'attribut `src` de l'élément `script` :

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

# intégration de javascript dans la page html

on utilise l'élément `script`, dans le corps ou l'entête du document.
2 cas de figure possibles :

- code javascript directement placé dans le corps du fichier html :

```
<script type="text/javascript">
...  code javascript ici
</script>
```

- code javascript dans un fichier séparé précisé par l'attribut `src` de l'élément `script` :

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

exemples : *code dans html* – *code dans fichier*

# intégration de javascript dans la page html

on utilise l'élément `script`, dans le corps ou l'entête du document.
2 cas de figure possibles :

- code javascript directement placé dans le corps du fichier html :

```
<script type="text/javascript">
...  code javascript ici
</script>
```

- code javascript dans un fichier séparé précisé par l'attribut `src` de l'élément `script` :

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

exemples : *code dans html* – *code dans fichier*

NB : flux de chargement du fichier html

# un style de programmation impératif

# un style de programmation impératif

- variables, types de données,

# un style de programmation impératif

- variables, types de données,
- structures de contrôles : séquences, conditionnelles et itératives

# un style de programmation impératif

- variables, types de données,
- structures de contrôles : séquences, conditionnelles et itératives
- modularisation : fonctions

# un style de programmation impératif

- variables, types de données,
- structures de contrôles : séquences, conditionnelles et itératives
- modularisation : fonctions
- objets

# un style de programmation impératif

- variables, types de données,
- structures de contrôles : séquences, conditionnelles et itératives
- modularisation : fonctions
- objets

Si vous connaissez un autre langage impératif

# un style de programmation impératif

- variables, types de données,
- structures de contrôles : séquences, conditionnelles et itératives
- modularisation : fonctions
- objets

Si vous connaissez un autre langage impératif

- une nouvelle syntaxe

# un style de programmation impératif

- variables, types de données,
- structures de contrôles : séquences, conditionnelles et itératives
- modularisation : fonctions
- objets

Si vous connaissez un autre langage impératif

- une nouvelle syntaxe
- quelques différences dans les règles de fonctionnement

# un style de programmation impératif

- variables, types de données,
- structures de contrôles : séquences, conditionnelles et itératives
- modularisation : fonctions
- objets

Si vous connaissez un autre langage impératif

- une nouvelle syntaxe
- quelques différences dans les règles de fonctionnement
- de nouvelles fonctions primitives à apprendre

# au programme...

# au programme...

# types primitifs

- **boolean**

# types primitifs

- **boolean**
  - 2 constantes `true`, `false`

# types primitifs

- **boolean**
  - 2 constantes `true`, `false`
  - opérateurs : négation `!`, et logique `&&`, ou logique `||`

# types primitifs

- **boolean**
  - 2 constantes `true`, `false`
  - opérateurs : négation `!`, et logique `&&`, ou logique `||`
- **number**

# types primitifs

- **boolean**
  - 2 constantes `true`, `false`
  - opérateurs : négation `!`, et logique `&&`, ou logique `||`
- **number**
  - pas de séparation nette entre entiers et flottants

# types primitifs

- **boolean**
  - 2 constantes `true`, `false`
  - opérateurs : négation `!`, et logique `&&`, ou logique `||`
- **number**
  - pas de séparation nette entre entiers et flottants
  - opérateurs : `+`, `-`, `*`, `/` (division flottante), `%` (reste de la division)

# types primitifs

- **boolean**
  - 2 constantes `true`, `false`
  - opérateurs : négation `!`, et logique `&&`, ou logique `||`
- **number**
  - pas de séparation nette entre entiers et flottants
  - opérateurs : `+`, `-`, `*`, `/` (division flottante), `%` (reste de la division)
  - `-Infinity`, `Infinity`

# types primitifs

- **boolean**
  - 2 constantes `true`, `false`
  - opérateurs : négation `!`, et logique `&&`, ou logique `||`
- **number**
  - pas de séparation nette entre entiers et flottants
  - opérateurs : `+`, `-`, `*`, `/` (division flottante), `%` (reste de la division)
  - `-Infinity`, `Infinity`
- **string**

# types primitifs

- **boolean**
  - 2 constantes `true`, `false`
  - opérateurs : négation `!`, et logique `&&`, ou logique `||`
- **number**
  - pas de séparation nette entre entiers et flottants
  - opérateurs : `+`, `-`, `*`, `/` (division flottante), `%` (reste de la division)
  - `-Infinity`, `Infinity`
- **string**
  - pas de type *caractère* séparé de `string`,
    il faut considérer des chaînes de longueur 1

# types primitifs

- **boolean**
  - 2 constantes `true`, `false`
  - opérateurs : négation `!`, et logique `&&`, ou logique ||
- **number**
  - pas de séparation nette entre entiers et flottants
  - opérateurs : `+`, `-`, `*`, `/` (division flottante), `%` (reste de la division)
  - `-Infinity`, `Infinity`
- **string**
  - pas de type *caractère* séparé de `string`,
    il faut considérer des chaînes de longueur 1
  - les chaînes se notent entre " ou ’ : *"exemple"*, *’un autre’*

# types primitifs

- **boolean**
  - 2 constantes `true`, `false`
  - opérateurs : négation `!`, et logique `&&`, ou logique ||
- **number**
  - pas de séparation nette entre entiers et flottants
  - opérateurs : `+`, `-`, `*`, `/` (division flottante), `%` (reste de la division)
  - `-Infinity`, `Infinity`
- **string**
  - pas de type *caractère* séparé de `string`,
    il faut considérer des chaînes de longueur 1
  - les chaînes se notent entre " ou ’ : *"exemple"*, *’un autre’*
  - opérateur de concaténation : `+`

# types primitifs

- **boolean**
  - 2 constantes `true`, `false`
  - opérateurs : négation `!`, et logique `&&`, ou logique ||
- **number**
  - pas de séparation nette entre entiers et flottants
  - opérateurs : `+`, `-`, `*`, `/` (division flottante), `%` (reste de la division)
  - `-Infinity`, `Infinity`
- **string**
  - pas de type *caractère* séparé de `string`,
    il faut considérer des chaînes de longueur 1
  - les chaînes se notent entre " ou ' : *"exemple"*, *'un autre'*
  - opérateur de concaténation : `+`
  - + objet `String` $\Longrightarrow$ nombreuses méthodes

# variables

## déclaration

Il faut déclarer les variables à l'aide du mot-clef `var`.
Une variable doit être déclarée avant d'être utilisée.

# variables

## déclaration

Il faut déclarer les variables à l'aide du mot-clef `var`.
Une variable doit être déclarée avant d'être utilisée.

## affectation

L'opérateur d'affectation se note `=`.
Une variable non initialisée a pour valeur `null` ou `undefined`

*types-variables-et-predicats.js*

# au programme...

# au programme...

# fonctions

## valeur de type fonction

- le mot-clef `function` permet de définir une donnée de type fonction,

```
function
```

# fonctions

## valeur de type fonction

- le mot-clef `function` permet de définir une donnée de type fonction,
- on précise entre parenthèses les **paramètre formels**, séparés par des virgules,

```
function(param1, param2, ...)
```

# fonctions

## valeur de type fonction

- le mot-clef `function` permet de définir une donnée de type fonction,
- on précise entre parenthèses les **paramètre formels**, séparés par des virgules,
- le corps de la fonction est noté entre accolades,

```
function(param1, param2, ...) {
    ...  corps de la fonction

}
```

# fonctions

## valeur de type fonction

- le mot-clef `function` permet de définir une donnée de type fonction,
- on précise entre parenthèses les **paramètre formels**, séparés par des virgules,
- le corps de la fonction est noté entre accolades,
- la valeur de retour d'une fonction est précisée par `return`, `return` n'est pas obligatoire ($\Rightarrow$ valeur retour = `undefined`)

```
function(param1, param2, ...) {
    ... corps de la fonction
    return expression;
}
```

# fonctions

## appel et nommage

- on appelle une fonction en précisant les **paramètres effectifs** entre parenthèses

*exemple-fonction.js*

# fonctions

## appel et nommage

- on appelle une fonction en précisant les **paramètres effectifs** entre parenthèses
- on peut nommer une fonction en définissant une variable dont la valeur est de type fonction

*exemple-fonction.js*

# autre syntaxe

```
                                    // éqv à :
function exemple (x, y) {           // var exemple = function(x,y) {
  var valeur = x+y;
  return 2*valeur + 1 ;
}

exemple(2,3);                       // vaut 11
```

# règle de portée

## locale et globale

- toute définition de variable dans une fonction est **locale** à la fonction
- une variable locale masque une variable globale de même nom

*exemple-portee.js – avec debugger*

# au programme...

# au programme...

# séquence et bloc

## séquence

On construit une séquence d'instructions en les séparant par un ; .

# séquence et bloc

## séquence

On construit une séquence d'instructions en les séparant par un ; .

## bloc d'instructions

Un bloc d'instructions en séquence se note entre accolades.
Un bloc d'instructions est une instruction.

# séquence et bloc

## séquence

On construit une séquence d'instructions en les séparant par un `;` .

## bloc d'instructions

Un bloc d'instructions en séquence se note entre accolades.
Un bloc d'instructions est une instruction.

Contrairement à de nombreux langages, en javascript un bloc ne définit pas de règle de portée.
La seule règle de portée se situe au niveau des fonctions.

# structure conditionnelle

```
if (condition) {
   séquence d'instructions si true
}
else {
   séquence d'instructions si false
}
```

```
var collatz = function(i) {
   if (i % 2 == 0) {
      return i/2;
   }
   else {
      return 3*i+1;
   }
}
```

- la partie **else** n'est pas obligatoire
- **false**, *0*, **""**, **NaN**, **null**, **undefined** valent **false**, tout le reste vaut **true**

# structures itératives : *pour*

```
for (var i =  inf;  i < max ; i=i+1) {     // i=i+1 s'écrit aussi i++
   séquence d'instructions
}
```

```
var sommeEntiers = function(borneMax) {
  var somme = 0;
  for(var i = 0 ; i < borneMax ; i=i+1) {
    somme = somme + i;
  }
  return somme;
}

sommeEntiers(100);                          // somme vaut 4950
```

# structures itératives : *tant que*

```
while ( condition ) {
   séquence d'instructions
}
```

```
var sommeChiffres = function(n) {
  var result = 0;
  while (n > 0) {
    result = result + (n % 10);
    n = Math.floor(n/10);
  }
  return result;
}
sommeChiffres(12345);                    // vaut 15
```

```
do {
   séquence d'instructions
} while (condition)
```

*avec debugger*

# *tant que* et *pour*

Une boucle *pour* peut toujours s'écrire sous la forme d'une boucle *tant que*.

```
pour i variant de borne_inf à borne_sup répéter
  corps_de_boucle
```

≡

```
i ← borne_inf
tant que i ≤ borne_sup répéter
  debutBloc
    corps_de_boucle
    i ← i + 1
  finBloc
```

# *tant que* et *pour*

Une boucle *pour* peut toujours s'écrire sous la forme d'une boucle *tant que*.

```
pour i variant de borne_inf à borne_sup répéter
  corps_de_boucle
```

≡

```
i ← borne_inf
tant que i ≤ borne_sup répéter
  debutBloc
    corps_de_boucle
    i ← i + 1
  finBloc
```

En javascript les boucles **`for`** sont des **`while`** déguisées :

```
for ( init; condition ; increment ) {
  séquence d'instructions
}
```

≡

```
init;
while (condition ) {
  séquence d'instructions;
  increment;
}
```

On peut donc aussi écrire :

```
var sommeChiffres = function(n) {
  for(var result=0; n > 0; n = Math.floor(n/10)) {
    result = result + (n % 10);
  }
  return result;
}
sommeChiffres(12345);                    // vaut 15
```

mais cela ne veut pas dire que l'on doit le faire...

# au programme...

# au programme...

# conversions de types

- javascript est (très) souple sur la notion de typage.
- javascrit applique « automatiquement » certaines conversions de type sur les valeurs lorsque le contexte le nécessite :
  - vers le type `boolean` (cf. remarque précédente)
  - vers le type `string`
  - vers le type `number`
- a une incidence sur la notion d'égalité

# conversion en booléen et en chaîne de caractères

```
var valeurBooleenne = function(val) {
  if (val) {          // dans ce contexte valeur booléenne attendue
    return "'"+val+"' est converti en true";     // chaine attendue
  }
  else {
    return "'"+val+"' est converti en false";
  }
}

valeurBooleenne("abcd");  // -> 'abcd' est converti en true
valeurBooleenne("");      // -> '' est converti en false

var x;
valeurBooleenne(x);       // -> 'undefined' est converti en false
x = 0;
valeurBooleenne(x);       // -> '0' est converti en false
x = 1;
valeurBooleenne(x);       // -> '1' est converti en true
```

# conversion en nombre

- une chaîne dont les caractères représentent un nombre est convertie en ce nombre
  NB : dans un expression avec l'opérateur `+` c'est la conversion vers chaîne qui l'emporte
- `NaN` : *Not a Number*
  valeur de conversion pour toute expression qui ne peut être convertie en un nombre
  peut se tester avec fonction `isNaN`.

```
"12.5"*3;           // -> 37.5
"99"-5;             // -> 94

"99"+5              // -> "995"     /!\

"deux"*3;           // -> NaN
isNaN("deux"*3);    // -> true
```

# parseInt et parseFloat

- convertissent une chaîne en nombre (entier ou flottant)
- seul le premier nombre dans la chaîne est retourné, les autres caractères (y compris correspondant à des nombres) sont ignorés
- si le premier caractère ne peut être converti en un nombre, le résultat sera `NaN`
- les espaces en tête sont ignorés

```
parseFloat("1.24");                    // ->   1.24
parseInt("42");                        // ->  42
parseInt("42 est la reponse");         // ->  42
parseInt("   42 est la reponse");      // ->  42
parseInt("42estlareponse");            // ->  42
parseInt("42 43 44");                  // ->  42
parseInt("reponse = 42");              // -> NaN
```

# égalités étranges

Du fait de la conversion, dans certains cas des valeurs de types différents peuvent être considérées égales.

```
1 == "1"          // -> true
10 != "10"        // -> false
1 == "un"         // -> false
0 == false        // -> true
"0" == false      // -> true /!\ alors que "0" se convertit en true
```

L'opérateur `===` teste à la fois le type et la valeur (négation `!==`).

```
1 === "1"          // -> false
0 === false        // -> false
10 === 9+1;        // -> true

1 !== "1";         // -> true
```

# au programme...

# au programme...

# objets

- les objets possèdent des méthodes (= fonctions)
- une méthode s'invoque sur un objet
- on utilise la « *notation pointée* »

exexmple : avec l'objet **String**

```
var s = new String("timoleon");     // création d'un objet String
var sub = s.substring(2,6);         // sub vaut "mole"
s.charAt(4);                        // vaut "l"
s.length;                           // vaut 8

                // conversion des valeurs string vers objet String
"abracadabra".charAt(2);            // vaut "r"
"abracadabra".substring(4,8);       // vaut "cada"
```

# window et document

Pour un document chargé dans un navigateur, 2 variables objet sont définies par défaut :

- `window` représente la fenêtre du navigateur dans laquelle le document est chargé.
  L'objet `window` est l'objet de base.
  Un objet `window` par onglet.
- `document` représente le document DOM chargé dans la fenêtre.

# premières interactions avec document html

*en attendant mieux...*

- `window.alert` affiche une « popup » d'information

# premières interactions avec document html

*en attendant mieux...*

- `window.alert` affiche une « popup » d'information
- `window.prompt`

# premières interactions avec document html

*en attendant mieux...*

- `window.alert` affiche une « popup » d'information
- `window.prompt`
  - affiche une boîte de dialogue avec une zone de saisie de texte

# premières interactions avec document html

*en attendant mieux...*

- `window.alert` affiche une « popup » d'information
- `window.prompt`
  - affiche une boîte de dialogue avec une zone de saisie de texte
  - a pour résultat le texte saisi

# premières interactions avec document html

*en attendant mieux...*

- `window.alert` affiche une « popup » d'information
- `window.prompt`
  - affiche une boîte de dialogue avec une zone de saisie de texte
  - a pour résultat le texte saisi

  **attention** : le résultat est de type `string`, prévoir des conversions avec `parseInt` ou `parseFloat` si nécessaire.

# premières interactions avec document html

*en attendant mieux...*

- `window.alert` affiche une « popup » d'information
- `window.prompt`
  - affiche une boîte de dialogue avec une zone de saisie de texte
  - a pour résultat le texte saisi

  **attention** : le résultat est de type `string`, prévoir des conversions avec `parseInt` ou `parseFloat` si nécessaire.
- `document.write` et `document.writeln`

# premières interactions avec document html

*en attendant mieux...*

- `window.alert` affiche une « popup » d'information
- `window.prompt`
  - affiche une boîte de dialogue avec une zone de saisie de texte
  - a pour résultat le texte saisi

  **attention** : le résultat est de type `string`, prévoir des conversions avec `parseInt` ou `parseFloat` si nécessaire.
- `document.write` et `document.writeln`
  - écrit du texte dans le flux html

# premières interactions avec document html

*en attendant mieux...*

- `window.alert` affiche une « popup » d'information
- `window.prompt`
  - affiche une boîte de dialogue avec une zone de saisie de texte
  - a pour résultat le texte saisi

  **attention** : le résultat est de type `string`, prévoir des conversions avec `parseInt` ou `parseFloat` si nécessaire.
- `document.write` et `document.writeln`
  - écrit du texte dans le flux html
  - le texte écrit est interprété par le navigateur

# premières interactions avec document html

*en attendant mieux...*

- `window.alert` affiche une « popup » d'information
- `window.prompt`
  - affiche une boîte de dialogue avec une zone de saisie de texte
  - a pour résultat le texte saisi

  **attention** : le résultat est de type `string`, prévoir des conversions avec `parseInt` ou `parseFloat` si nécessaire.
- `document.write` et `document.writeln`
  - écrit du texte dans le flux html
  - le texte écrit est interprété par le navigateur

  **attention** : efface le contenu du document si le flux doit être réouvert

  ⇒ **ne pas utiliser** pour modifier un document, uniquement lors de sa création ! *(en fait ne pas utiliser du tout...)*

# premières interactions avec document html

*en attendant mieux...*

- `window.alert` affiche une « popup » d'information
- `window.prompt`
  - affiche une boîte de dialogue avec une zone de saisie de texte
  - a pour résultat le texte saisi

  **attention** : le résultat est de type `string`, prévoir des conversions avec `parseInt` ou `parseFloat` si nécessaire.
- `document.write` et `document.writeln`
  - écrit du texte dans le flux html
  - le texte écrit est interprété par le navigateur

  **attention** : efface le contenu du document si le flux doit être réouvert

  ⇒ **ne pas utiliser** pour modifier un document, uniquement lors de sa création ! *(en fait ne pas utiliser du tout...)*

exemples *1* – *2* – *3* – *4*

## à suivre...

# javascript : tableaux