Chapitre 2-2
Conditionnement des traitements

1. Présentation de la syntaxe

La structure conditionnelle en JavaScript est très proche syntaxiquement de celle vue précédemment en langage descriptif algorithmique.

Tout à fait classiquement le bloc d'instructions à exécuter dans le cas où la condition testée est vraie est délimité par un jeu d'accolades ({}). Il est aussi possible de prévoir une séquence d'instructions alternative avec le mot-clé `else`. Cette séquence sera aussi encadrée par des accolades.

Le langage JavaScript est assez permissif quant au positionnement de ces accolades. Ainsi vous rencontrerez dans les scripts les constructions suivantes :

```
if (condition)
{
        Actions_1;
}
else
{
        Actions_2;
}
```

ou

```
if (condition) {
        Actions_1;
}
else {
        Actions_2;
}
```

ou

```
if (condition) { Actions_1; } else { Actions_2; }
```

avec :

– condition représentant un test de comparaison générant un résultat booléen `true` ou `false`,
– `Actions_1` et `Actions_2` représentant des séquences d'instructions (en général réparties sur plusieurs lignes).

Remarque

Attention pour effectuer un test de comparaison en égalité l'opérateur est le double égal (==) à ne pas confondre avec le simple égal (=) qui sert à effectuer les affectations.

Dans les tests de comparaison avec des constantes on préférera par exemple :

```
if (5 == compteur)
{
        Actions_1;
}
else
{
        Actions_2;
}
```

à

```
if (compteur == 5)
{
        Actions_1;
}
else
{
        Actions_2;
}
```

car l'étourderie consistant à confondre le == avec le = est plus facile à détecter dans le premier cas, JavaScript renvoyant une erreur. Dans le second cas (si vous mettez un = au lieu d'un ==) la séquence `Actions_1` sera systématiquement exécutée car l'évaluation de compteur = 5 donnerait toujours `true` (JavaScript estimant que l'affectation est réussie).

2. Exemples

Remarque

Pour faciliter le repérage des exercices JavaScript, la numérotation vue dans le chapitre Développement à partir d'algorithmes sera conservée.

2.1 Exercice n°6 : Polynôme du second degré

Sujet

Calculer les racines d'un polynôme du second degré Ax^2+Bx+C (avec A<>0 dans l'absolu mais ce test ne sera pas effectué ici). Les valeurs A, B et C seront saisies au clavier.

Corrigé (partiel) en JavaScript

```
/* Saisie des paramètres */
a = prompt("a :");
b = prompt("b :");
c = prompt("c :");

/* Calcul du discriminant */
delta = (parseInt(b) * parseInt(b)) - (4 * parseInt(a) * parseInt(c));

/* Affichage des paramètres */
document.write("a : " + a + "<br />");
document.write("b : " + b + "<br />");
document.write("c : " + c + "<br />");
document.write("Discriminant delta : " + delta + "<br />");

/* Détermination des racines */
if (delta < 0)
{
   document.write("Pas de solutions");
}
else
{
   if (delta == 0)
   {
         document.write("Solution unique : " + (-b / (2 * a)));
   }
   else
   {
         document.write("Solution n°1 : " + (-b +  Math.sqrt(delta)) /
         (2 * a) + "<br />");
         document.write("Solution n°2 : " + (-b -  Math.sqrt(delta)) /
```

```
            (2 * a));
    }
}
```

Commentaires du code JavaScript

Vous remarquerez qu'il a fallu imbriquer deux structures conditionnelles pour traiter le problème posé. La non-imbrication des structures était possible (équivalent de trois **Si ... Finsi** successifs au niveau algorithmique) mais cette solution n'aurait pas été optimale. Vous noterez aussi le soin particulier apporté au niveau de l'alignement des accolades et également au niveau du décalage (indentation) des blocs d'instructions.

Pour déterminer la racine carrée du discriminant (`delta`), il a fallu avoir recours à la méthode `sqrt` de l'objet JavaScript `Math`. Nous aurons l'occasion de revoir cet objet plus loin dans ce livre. Le calcul de la racine carrée aurait aussi pu être effectué par une élévation de `delta` à la puissance 0.5. L'opération de l'élévation à la puissance (exponentiation) est notée `^`.

2.2 Exercice n°8 : Impression du libellé d'un mois

Sujet

Imprimer en lettres le mois correspondant à un numéro donné au clavier (compris entre 1 et 12). Le contrôle de la saisie n'est pas à prévoir.

Corrigé (partiel) en JavaScript

```
/* Déclaration de variables locales */
var mois;

/* Saisie du numéro de mois */
mois = parseInt(prompt("Numéro du mois (1 à 12) :"));

/* Affichage du résultat */
switch (mois)
{
   case 1:
         document.write("Mois n° " + mois +" : Janvier");
         break;
   case 2:
         document.write("Mois n° " + mois +" : Février");
         break;
   case 3:
         document.write("Mois n° " + mois +" : Mars");
         break;
   case 4:
         document.write("Mois n° " + mois +" : Avril");
```

```
            break;
      case 5:
            document.write("Mois n° " + mois +" : Mai");
            break;
      case 6:
            document.write("Mois n° " + mois +" : Juin");
            break;
      case 7:
            document.write("Mois n° " + mois +" : Juillet");
            break;
      case 8:
            document.write("Mois n° " + mois +" : Août");
            break;
      case 9:
            document.write("Mois n° " + mois +" : Septembre");
            break;
      case 10:
            document.write("Mois n° " + mois +" : Octobre");
            break;
      case 11:
            document.write("Mois n° " + mois +" : Novembre");
            break;
      case 12:
            document.write("Mois n° " + mois +" : Décembre");
            break;
      default:
            document.write("Erreur de saisie sur le n° de mois");
}
```

Commentaires du code JavaScript

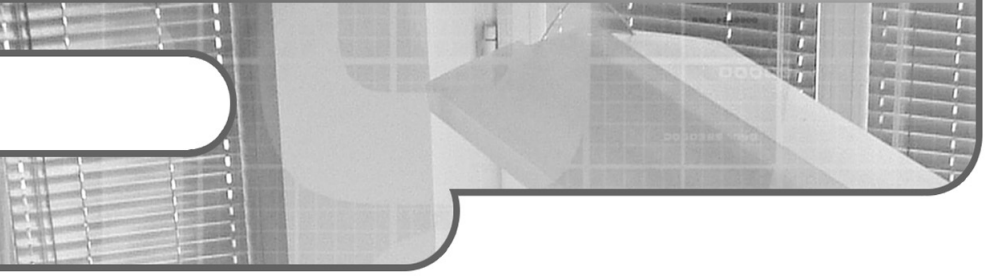
Plutôt que, comme dans l'exercice précédent, d'imbriquer des structures conditionnelles (fastidieux à écrire dans notre cas), la structure `switch` est ici utilisée.

Notez au niveau de la saisie du numéro du mois dans la variable `mois` (non typée au niveau de sa déclaration) qu'une conversion est demandée par l'intermédiaire de la méthode `parseInt`. La variable `mois`, une fois la saisie et la conversion effectuées, sera de type entier pour la suite du traitement.

La variable `mois` est testée par la structure conditionnelle `switch`. Dans le cas où la variable `mois` vaut 1, l'affichage de l'intitulé "`Janvier`" est effectué, et ainsi de suite.

Dans le cas où une saisie erronée du numéro de mois est faite alors l'instruction intégrée dans le cas `default` est exécutée.

Revenons aussi sur le rôle primordial de l'instruction `break`. En son absence en fin de chacun des cas, dès que la condition est vérifiée le traitement associé est déclenché mais ce serait également le cas pour tous les traitements suivants (même si la condition n'est pas respectée pour ces traitements). Par exemple, la saisie de la valeur 5 en tant que numéro de mois générerait l'affichage de `Mai`, `Juin`, `Juillet`, ..., `Décembre`.



Chapitre 3
Adopter les bonnes pratiques

1. Espace de noms

1.1 Principe

Lorsque nous développons, nous éviterons toujours d'exposer un trop grand nombre de fonctions ou de variables/constantes dans l'espace de noms global afin d'éviter les conflits de nom. Cela signifie de ne donner l'accès qu'à ce qui est utile et masquer tout le reste. C'est un peu le principe d'une boîte noire avec des entrées et sorties bien définies mais la mécanique interne reste cachée.

C'est d'autant plus important si le code risque d'être utilisé dans d'autres contextes. Le problème vient que nous avons l'habitude d'ajouter des fonctions dans nos fichiers sans prendre en compte la réutilisation. Notre projet devenant plus conséquent, il devient de plus en plus difficile à maintenir. Associer deux codes peut alors provoquer des conflits qui ne sont pas forcément visibles au premier coup d'œil et le résultat d'exécution devient incertain.

Pour limiter les conflits de nom, nous ajouterons un contexte supplémentaire (une sorte de super contexte) qui garantira que nos fonctions et variables/constantes ne pourront être altérées/utilisées accidentellement. Cet espace de noms est une sorte de conteneur de noms. Un nom n'a alors de sens que par rapport à son espace de noms. Invoquer une fonction qui n'est pas dans l'espace de noms attendu devient donc impossible.

Chaque code ayant son espace de noms, il n'y a plus de risque de collisions associé à un code appartenant à un autre espace de noms.

Un espace de noms est une possibilité associée à de nombreux langages. Dans le langage Java, par exemple, le mot-clé `package` sert à effectuer la déclaration, de même en C# avec `namespace`. En JavaScript, nous n'avons malheureusement pas de mot-clé pour cet usage mais nous avons d'autres astuces tout aussi puissantes pour l'obtenir.

Ce principe est important pour la qualité de vos programmes. Une fois que l'aurez compris, il deviendra naturel et votre développement s'améliorera d'autant plus.

Plus votre programme est important et plus vous avez besoin d'espaces de noms. C'est la dimension de votre projet qui en impose l'usage. Prendre en compte l'espace de noms sera donc le gage d'un code de meilleure qualité, qui pourra évoluer plus simplement et avec plus de sécurité et donc avec moins de régressions et/ou d'effets de bord.

1.2 Fonction

1.2.1 Fonction interne

Toutes les déclarations faites dans une fonction deviennent locales à cette fonction. En effet, pour rappel, lorsqu'une variable est déclarée dans une fonction via le mot-clé `let`, son contexte d'exécution est lié à celui de la fonction et d'un bloc d'instructions et en dehors de cette dernière, elle n'existe plus. On peut donc percevoir que la fonction est un puissant moyen pour limiter la portée des déclarations et donc semble un candidat idéal pour l'espace de noms.

Exemple

```
function aireRectangle( longueur, largeur ) {
  const aire = longueur * largeur;
  alert( aire + " cm2" );
}

aireRectangle( 10, 5 );
alert( aire );   // aire is not defined
```

Dans cet exemple, la constante `aire` est déclarée à l'intérieur de la fonction `aireRectangle`, elle n'existe donc pas en dehors de ce périmètre, c'est pourquoi la dernière instruction `alert(aire)` échoue. Notre fonction `aireRectangle` agit comme un espace de noms.

Cette particularité n'est pas limitée qu'aux déclarations de variables/constantes mais est également vraie pour les déclarations de fonctions.

```
function aire( type, longueur, largeur ) {

  let aire = 0;

  function aireRectangle( longueur, largeur ) {
    const aire = ( longueur * largeur );
    return aire;
  }
  function aireCarre( cote ) {
    return aireRectangle( cote, cote );
  }
  if ( type == "rectangle" ) {
    aire = aireRectangle( longueur, largeur );
  } else
  if ( type == "carre" ) {
    aire = aireCarre( longueur );
  }
  return aire;
}

alert( aire( "rectangle", 10, 20 ) );  // 200
alert( aire( "carre", 10 ) );          // 100
alert( aireCarre( 20 ) );              // Erreur !
```

Les fonctions `aireRectangle` et `aireCarre` n'existent pas en dehors de la fonction `aire`, preuve en est que notre dernière instruction consistant à utiliser la fonction `aireCarre` a provoqué une erreur d'exécution.

En combinant ces fonctions internes avec des variables locales, nous obtenons en réalité un espace de noms pour `aireRectangle` et `aireCarre` qui n'appartiennent qu'à la fonction `aire`. Personne ne peut donc altérer ou manipuler ces fonctions en dehors de votre fonction. Il s'agit donc d'une technique simple pour éliminer les accès indésirables.

1.2.2 Fonction anonyme

Si nous voulons protéger notre exécution, il est également possible de passer par une fonction anonyme (ou bien une fonction fléchée) comme nous l'avons vu succinctement dans le premier chapitre. Cette fonction sans nom garantira un contexte indépendant.

Exemple

```
(function () {
  function aireRectangle( longueur, largeur ) {
    const aire = ( longueur * largeur );
    return aire;
  }
  function aireCarre( cote ) {
    return aireRectangle( cote, cote );
  }
  alert( aireRectangle( 10, 20 ) );
  alert( aireCarre( 20 ) );
} )();

alert( aireRectangle( 10, 20 ) ); // Erreur
```

La fonction anonyme est exécutée dès qu'elle est déclarée puisque, n'ayant pas de nom, elle ne peut être invoquée ultérieurement. Son rôle n'est pas de fournir un service réutilisable mais de protéger le contenu de tout accès ultérieur.

Donc, une fois l'exécution réalisée, tout ce qui était à l'intérieur de cette fonction anonyme devient inaccessible.

La fonction anonyme sert donc d'espace de noms à notre code. Cela explique pourquoi l'invocation ultérieure de la méthode `aireRectangle` échoue.

L'exécution est saine car nous ne pouvons pas créer de collision sur nos noms de fonctions. Si nous associons notre code à un autre code qui lui-même a défini la fonction `aireRectangle`, notre code continuera de fonctionner normalement.

À noter que la forme fléchée de la fonction est plus compacte. Dans l'exemple ci-dessous, pour vous montrer que notre fonction fléchée n'altère pas ce qui est déjà dans l'espace global, nous avons écrit une nouvelle fonction `aireRectangle` avant d'invoquer notre fonction fléchée.

```
function aireRectangle( longueur, largeur ) {
    alert( `L'air du rectangle est ${longueur} * ${largeur}` );
}

(() => {
    function aireRectangle( longueur, largeur ) {
        const aire = ( longueur * largeur );
        return aire;
    }
    function aireCarre( cote ) {
        return aireRectangle( cote, cote );
    }
    alert( aireRectangle( 10, 20 ) );
    alert( aireCarre( 20 ) );
} )();


aireRectangle(100,200); // L'aire du rectangle est 100x200
```

Bien que notre fonction fléchée ait aussi à l'intérieur une fonction `aireRectangle`, elle n'empiète pas sur celle qui avait été définie avant et qui continue à marcher normalement lors du dernier appel.

1.2.3 Fonction anonyme avec paramètres

Le cas précédent est simple mais insuffisant dans la pratique car nous aimerions malgré tout disposer d'une fonction `aire` accessible partout par exemple mais sans rendre visibles les fonctions annexes `aireCarre` et `aireRectangle`.

Pour cela, nous pouvons créer un objet vide qui va nous servir de conteneur pour les fonctions accessibles. Il jouera alors le rôle d'espace de noms pour ces dernières.

Exemple

```
monAire = {};

(function ( ns ) {
  function aireRectangle( longueur, largeur ) {
    const aire = ( longueur * largeur );
    return aire;
  }
  function aireCarre( cote ) {
    return aireRectangle( cote, cote );
  }
  function aire() {
    if ( arguments.length == 2 ) {
      return aireRectangle( arguments[ 0 ], arguments[ 1 ] );
    } else
    if ( arguments.length == 1 ) {
      return aireCarre( arguments[ 0 ] );
    } else
    return "calcul d'aire impossible";
  }

  ns.aire = aire; // Contenu accessible
} )( monAire );

alert( monAire.aire( 10,20 ) );
```

Dans cet exemple, nous avons créé une fonction `aire` interne à une fonction anonyme. Celle-ci utilise les fonctions `aireCarre` et `aireRectangle`. Pour que la fonction `aire` puisse être utilisable partout, nous avons fait en sorte que la fonction anonyme exécutée puisse avoir un paramètre objet stockant la référence à la méthode `aire`.

C'est cet objet qui va faire office d'espace de noms pour la méthode `aire`. Les autres méthodes `aireCarre` et `aireRectangle` restent définitivement cachées de l'utilisateur.

Nous aurions pu également faire un test de la propriété `aire` de l'objet en fin de fonction anonyme pour ne pas écraser une précédente déclaration, par exemple avec :

```
ns.aire = ns.aire ?? aire
```

Ainsi, nous retrouvons par ce système quelque chose de similaire aux méthodes privées et publiques en programmation objet. La fonction anonyme joue alors le rôle de classe.

Le fait que la méthode `aire` puisse continuer à fonctionner en dehors de la fonction anonyme est lié au principe de fermeture (*closure*). La fermeture est la faculté pour une fonction qui est bien accessible d'utiliser un contexte qui ne peut plus l'être (il est donc privé).

La variable `monAire` est dans l'espace de noms global. Il est cependant possible de réduire légèrement les conflits de nom en utilisant l'objet prédéfini `window`. Cet objet a la particularité d'avoir ses propriétés visibles dans l'espace de noms global. En remplaçant `monAire` par `window.monAire`, on réduit la visibilité de notre variable `monAire` qui devient alors une propriété de `window`. À l'usage, il n'y a pas vraiment de différence.

D'une manière générale, l'association de nouvelles facultés à `window` est plutôt dépendante du contexte web, on cherchera à apporter des facultés supplémentaires à notre fenêtre.

Pour remarque, les collisions de noms sont limitées par les espaces de noms mais à condition que ceux-ci soient aussi distingués. Sinon, on reporte le problème à un niveau supérieur. Généralement, on emploiera une convention pour le choix d'un nom d'espace de noms de type URI (*Uniform Resource Identifier*) pour être sûr d'en être les seuls détenteurs. Par exemple, si on travaille dans une société ABC, on pourrait faire en sorte que l'objet `monAir` s'appelle `abc`, voire `com_abc`...

1.3 Fermeture

Nous avons déjà utilisé une fermeture pour notre fonction `aire`, mais pour être plus explicite, nous allons ici l'utiliser d'une autre façon, en ne gardant au final qu'une fonction d'accès.

```
cont monAire = ( function () {
  function aireRectangle( longueur, largeur ) {
    const aire = ( longueur * largeur );
    return aire;
  }
  function aireCarre( cote ) {
    return aireRectangle( cote, cote );
  }
  function aire() {
    if ( arguments.length == 2 ) {
      return aireRectangle( arguments[ 0 ], arguments[ 1 ] );
    } else
    if ( arguments.length == 1 ) {
      return aireCarre( arguments[ 0 ] );
    } else
    return "calcul d'aire impossible";
  }
  return function() {
    return aire.apply(this,arguments);
  };
} )();

alert( monAire( 20 ) );
alert( monAire( 10,20 ) );
```

Dans cet exemple, notre fonction anonyme retourne une nouvelle fonction. C'est elle qui conserve la logique de fonctionnement que nous souhaitons exposer. Ici, nous faisons en sorte que la fonction `aire` invisible puisse être utilisée avec n'importe quel nombre d'arguments, c'est pour cette raison que nous employons `apply`.

La fermeture est bien respectée car la fonction retournée continue à accéder aux fonctions `aire`, `aireCarre` et `aireRectangle` alors que ces dernières sont inaccessibles dans le contexte global.

Cette technique n'est pas adaptée si vous avez plusieurs fonctions à rendre accessibles.

1.4 Classe

Comme nous venons de le voir, l'espace de noms qu'induit la fonction anonyme sert à déclarer des fonctions ou des variables dont l'usage reste interne et donc invisible en dehors de la fonction. Ce système est similaire à ce que nous pouvons trouver en programmation objet avec des méthodes de classe d'accès publiques ou privées. Grâce à cela, nous allons pouvoir créer une nouvelle forme de classe combinant à la fois une partie cachée et une partie accessible.

Reprenons notre exemple de calcul d'aire, nous avons :

```
const figure = ( function () {
  // Partie privée

  function aireRectangle( longueur, largeur ) {
    const aire = ( longueur * largeur );
    return aire;
  }
  function aireCarre( cote ) {
    return aireRectangle( cote, cote );
  }
  function aire() {
    if ( arguments.length == 2 ) {
      return aireRectangle( arguments[ 0 ], arguments[ 1 ] );
    } else
    if ( arguments.length == 1 ) {
      return aireCarre( arguments[ 0 ] );
    } else
      return "calcul d'aire impossible";
  }

  // Partie publique de notre objet

  return {
    longueur : 0,
    largeur : 0,
    setLongueur : function( longueur ) {
      this.longueur = longueur;
    },
    setLargeur : function( largeur ) {
      this.largeur = largeur;
```