LA PROGRAMMATION COBOL

Sorow, Rod

29 octobre 2015

Reste de savoir

Table des matières

1 Introduction

Vous en rêviez (ou pas) ? Nous avons décidé de sortir le COBOL de la cave sombre où il se trouvait pour vous le faire découvrir et vous montrer qu'il n'est pas aussi méchant qu'il en a l'air. :D

-> 

<-

Il faut savoir que ce langage, bien que peu connu est omniprésent dans la vie de tous les jours. Une grande majorité des transactions bancaires et des communications transitent par des programmes... en COBOL ! Pour résumer, voici une présentation de ce que serait un monde sans COBOL *(en anglais)* :

-> !(https ://www.youtube.com/watch ?v=gx2YCbIxYPI) <-

Maintenant que vous savez que vous ne pouvez plus vivre sans faire du COBOL, ce cours est accessible à partir du moment où vous avez des notions de programmation (variables, conditions, etc.). Sinon, je vous invite à lire la première partie de ce cours afin d'assimiler les principaux concepts. Mais il faut avouer que de manière générale, c'est rarement le premier langage que vous apprenez... :D

2 Les bases du COBOL

Puisqu'il faut bien commencer quelque part, autant attaquer par la base ! Dans cette partie, nous allons voir les notions fondamentales du Cobol, ça sera donc votre point de départ si vous n'en avez jamais fait car même si vous maitrisez ces notions dans d'autres langages, la syntaxe de Cobol possède de nombreuses particularités.

2.1 Introduction

Si vous avez déjà travaillé en COBOL et que vous avez déjà le nécessaire, vous pouvez passer au prochain chapitre.

Toujours là ? Comme vous l'aurez sûrement compris, ce chapitre sera dédié à faire un petit tour du propriétaire, au téléchargement et à l'installation d'un IDE COBOL.

2.1.1 Petit historique

Pour commencer, l'acronyme COBOL veut dire : **CO**mmon **B**usiness **O**riented **L**anguage, autrement dit, un langage orienté vers la gestion.

Pour la question historique, je ne vais pas réinventer la roue et je vais vous citer un petit extrait de notre ami à tous, Wikipédia :

> Le COBOL a initialement été créé en 1959 par le Short Range Committee, un des trois comités proposés à une rencontre au Pentagone en mai 1959 organisée par Charles Phillips du département de la défense des États-Unis.
>
> Le comité a été formé pour recommander une approche à court terme pour un langage commun, indépendant des constructeurs, pour les applications de gestion de l'administration américaine.
>
> Il était constitué de membres représentant six constructeurs d'ordinateurs et trois agences gouvernementales. Les six constructeurs informatiques étaient Burroughs Corporation, IBM, Minneapolis-Honeywell, RCA, Sperry Rand, et Sylvania Electric Products.
>
> Les trois agences du gouvernement étaient le US Air Force, le David Taylor Model Basin, et l'Institut national des standards. Ce comité était présidé par un membre du NBS. Des comités à moyen et long terme ont également été proposés au Pentagone.
> Source : Wikipédia

C'est vieux et lourd, mais il faut se remettre dans le contexte de l'époque, les moyens techniques n'étaient pas les mêmes. C'est encore très utilisé dans les domaines fiscaux et de l'entreprise, car le coût d'entretien du code est moins important qu'une refonte totale des systèmes informatiques actuellement en place.

Ce n'est pas pour vous faire peur, mais certains ont eu le loisir de traduire l'acronyme COBOL d'une manière différente : "Compiles Only Because Of Luck" ce qui se traduit par "Compile seulement par chance". :p

2.1.1.0.1 La forme

Je vais vous montrer à quoi ressemble un programme Cobol afin que vous découvriez sa forme particulière :

```
       IDENTIFICATION DIVISION.
       PROGRAM-ID. Exemple.

       SCREEN SECTION.
       1 pla.
           2 BLANK SCREEN.
           2 LINE 6 COL 10 VALUE 'Hello world !'.


       PROCEDURE DIVISION.

       DISPLAY pla.

       STOP RUN.
```

----------end-of-text----------

->

<-

Pour l'instant, je ne vous demande pas de comprendre le code, simplement de regarder la capture d'écran. Vous voyez la zone blanche où on va écrire nos lignes ? Eh bien elle est aussi grande que sur la capture, et on ne peut pas l'agrandir.

[[question]] | Quoi ?! Mais elle est vraiment petite...

En fait c'est surtout historique, elle sert à représenter la taille des cartes perforées d'antan ! Voyez à quoi ressemble une carte :

->

Cliquez pour agrandir <-

Hé oui, avant même les disquettes, nos programmes étaient stockés sur des cartes possédant un certain nombre de colonnes comme celle-ci ! Pour tout vous dire, la zone blanche de la capture représente des colonnes, numérotées de 8 à 72 (c'est là qu'on va passer 99% de notre temps).

En réalité le compilateur va interpréter uniquement les colonnes 7 à 72, les codes venant après étant tout simplement ignorés par la machine chargée d'interpréter ces cartes. Ce qui veut dire que si vous écrivez à droite ou à gauche de la zone blanche, le compilateur ignorera ce que vous y avez écrit ! On verra que la colonne 7 joue un rôle un peu particulier dans la suite de ce cours.

2.1.2 Ayez les bons outils

Tout le long de ce tutoriel nous utiliserons le logiciel **Net Express** sous Windows. C'est un logiciel payant, mais vous pouvez l'utiliser pendant 30 jours avec la période d'essai.

[[information]] | Malheureusement, je n'ai pas trouvé d'autre alternative simple et gratuite proposant un compilateur Cobol. Ceux que nous avons testés ne fonctionnent pas toujours correctement. Cependant, après avoir contacté MicroFocus, il s'avère qu'ils prévoient de donner à leur outil un accès plus simple pour les étudiants, mais aucune date n'a été communiquée. Si jamais vous connaissez un compilateur complet, libre et facile à installer, nous sommes preneurs !

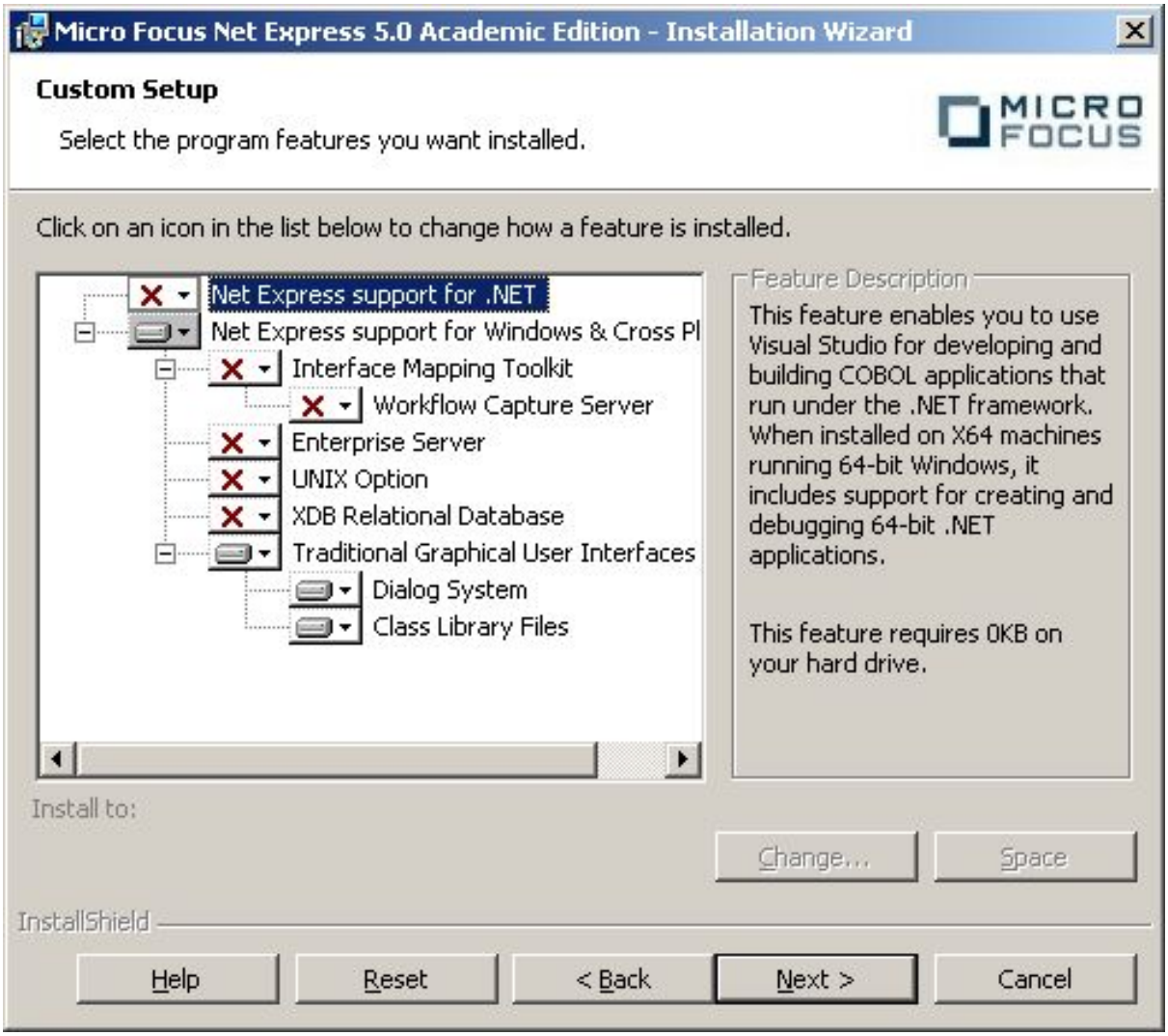
2.1.2.1 L'IDE NetExpress

2.1.2.1.1 Téléchargement Je vais vous guider pas à pas pour l'installation de Net Express pour Windows 7, car celle-ci comporte quelques particularités. Tout d'abord, je vous invite à télécharger le logiciel :

->  Télécharger Net Express 5.1 <-

2.1.2.1.2 Installation [[information]] | Pour Windows XP et Vista, l'installation devrait être sensiblement la même.

Lançons donc l'installation. Si vous avez acheté une licence, entrez votre clef, sinon choisissez la version d'évaluation. C'est ensuite qu'il faut faire attention, veuillez à ne sélectionner que les options suivantes :



->
<-

Pour la suite il suffit de laisser l'installation se poursuivre, il n'y a rien d'autre à mentionner.

2.1.3 Présentation de l'interface

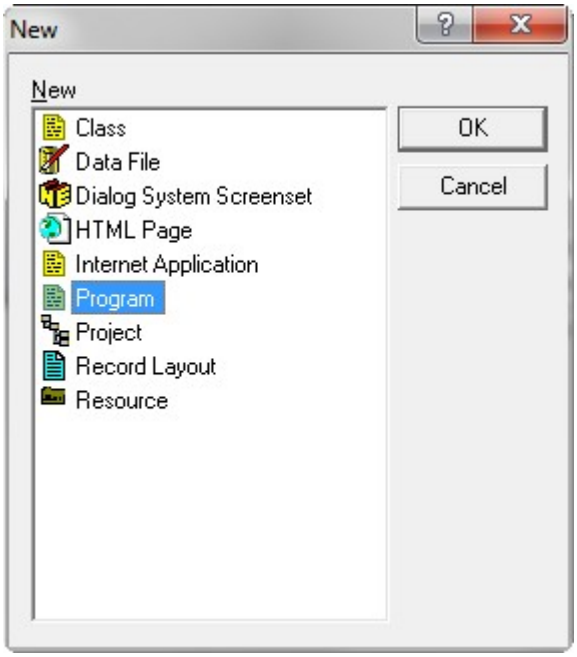
2.1.3.1 NetExpress

Une fois l'installation faite, en ouvrant l'IDE vous obtenez cette fenêtre :


->

Cliquez pour agrandir. <-

Rien de bien exceptionnel jusque là, on va ouvrir notre premier fichier, pour cela : **File** > **New** (||Ctrl|| + ||N||) :

-> 

<-

Sélectionnez ensuite **Program**, une fenêtre s'ouvre et c'est ici que tout va se jouer :

->  <-

Cliquez pour agrandir. <-

Personnellement, j'aime bien travailler avec la fenêtre en plein écran, il suffit de cliquer sur cette icône dans le coin supérieur droit :

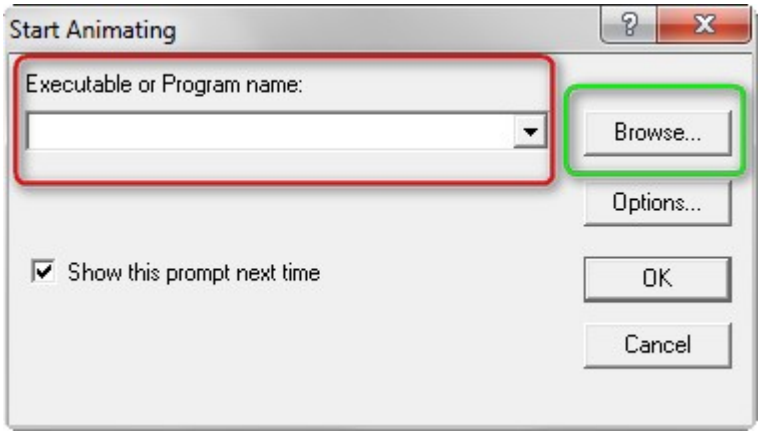
->  <-

Maintenant, je vais vous montrer comment compiler et exécuter votre programme.

Pour compiler rien de plus simple, il suffit de cliquer sur cette icône  située juste sous la barre d'outils.

2.1.3.1.1 Tout va bien

Si vous n'avez pas d'erreur à la compilation, vous pouvez lancer votre application en cliquant ici  , et une petite fenêtre s'ouvre alors :

-> 

<-

J'ai mis en valeur deux champs, ce sont ceux-ci que nous allons utiliser le plus régulièrement. Le cadre rouge sert de raccourci quand vous avez déjà exécuté au moins une fois votre projet, et quand vous cliquez sur la petite flèche, une liste des derniers programmes exécutés vous sera proposée. Sinon, vous devez cliquer sur **Browse** dans le cadre vert pour aller chercher votre fichier à l'endroit où vous l'avez enregistré.

Ensuite il ne vous reste plus qu'à valider et la console s'ouvre.

[[attention]] | Ce n'est pas parce que votre programme compile qu'il fonctionne, ça peut paraitre évident mais je préfère quand même le rappeler.

Si jamais votre programme plante par la suite, NetExpress vous propose un mode de debug. Pour le lancer il suffit de cliquer sur cette icône  . Cela vous permettra de voir le déroulement du programme étape par étape, en mettant en valeur les morceaux de code qui sont actifs.

2.1.3.1.2 Oup's ! Ça ne compile pas... ->  <-

Quand vous aurez commencé à coder, vous remarquerez des petites croix rouges qui apparaissent sur la gauche de vos lignes de codes : elles vous donnent un indice sur la localisation de votre erreur.

[[attention]] | Elles ne sont pas toujours exactes, car parfois l'erreur peut venir de plus haut dans votre code.

En survolant ces croix, une infobulle s'ouvre pour vous donner un indice sur le type de l'erreur rencontrée. Il ne vous reste plus qu'à trouver la source du problème ! :D

Lorsque vous corrigez une erreur, il peut arriver que toutes les autres croix disparaissent ! Cela fait généralement plaisir, mais l'inverse peut aussi se produire...

[[information]] | Ne vous attendez pas à une auto-complétion ou quoi que ce soit dans le genre avec NetExpress, il est loin d'être au niveau des fameux Eclipse ou NetBeans... Tout se fait à l'huile de coude ! :diable :

Maintenant que vous avez le nécessaire, plus d'excuse, nous allons commencer à étudier ! :pirate :

2.2 Vos premiers pas

Nous allons découvrir toutes les notions essentielles de ce langage. À la fin du chapitre vous serez capable de créer un programme qui gère de manière simple les entrées clavier d'un utilisateur.

2.2.1 Structure générale

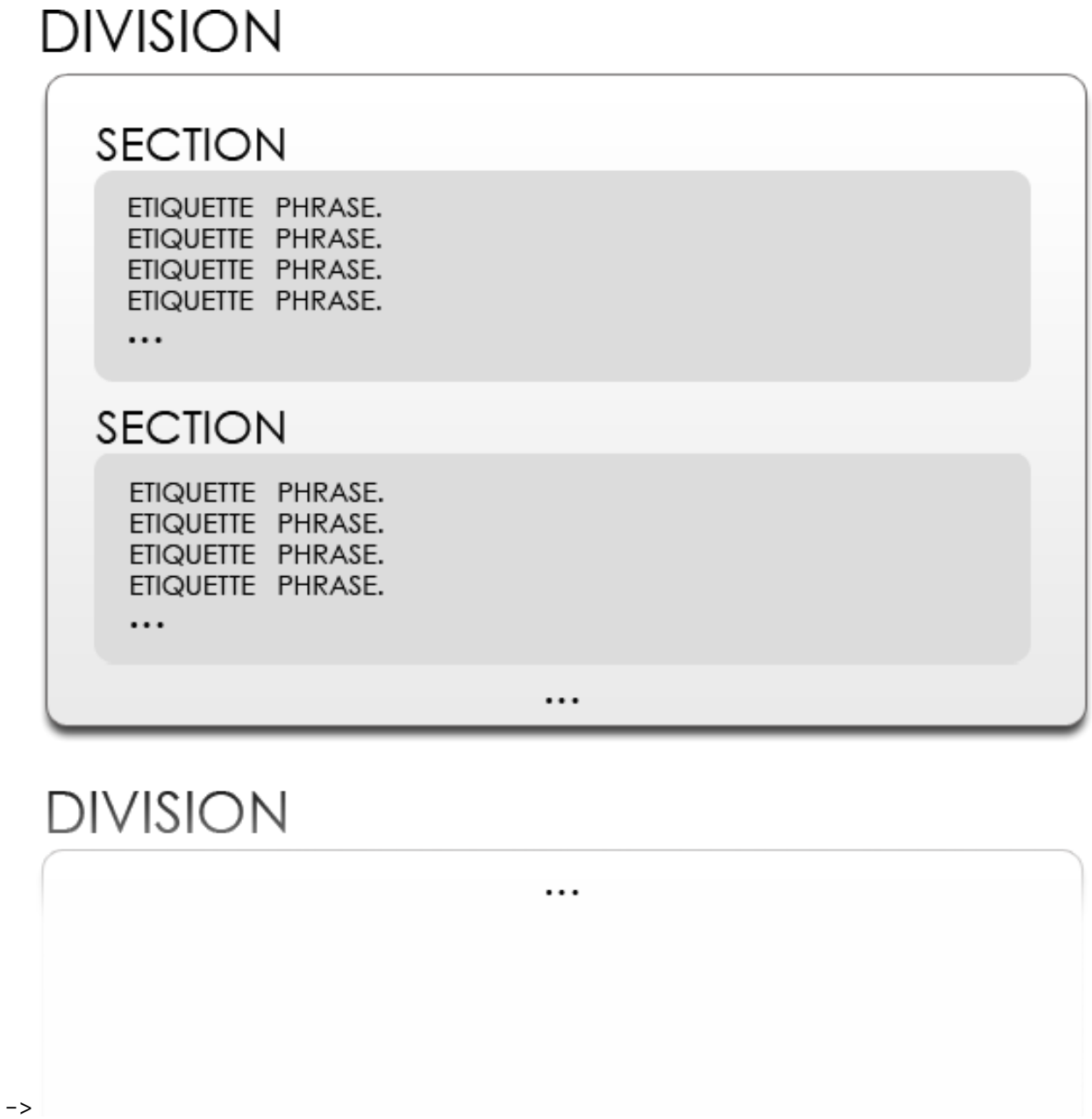
Avant d'attaquer la structure classique d'un programme, j'aimerais vous faire faire un petit tour d'horizon de la syntaxe générale.

2.2.1.1 Principe global

Cobol est un langage de programmation très structuré, vous ne pouvez pas noter ce que vous voulez où ça vous chante. Il y a une section consacrée à la déclaration de toutes vos variables, une autre pour noter ce que vous allez afficher et/ou pour récupérer les saisies de l'utilisateur, etc.

En gros, cela se décompose en **divisions** qui contiennent plusieurs **sections** et qui elles-mêmes sont formées de **paragraphes**. Et ~~comme dans la langue française~~ un paragraphe est composé d'une **étiquette** et de **phrases** ! Et cette fois-ci, comme en français, chaque phrase finit par un **point**.

Voici un schéma qui vous récapitule cela de manière très simpliste :




```
->
```

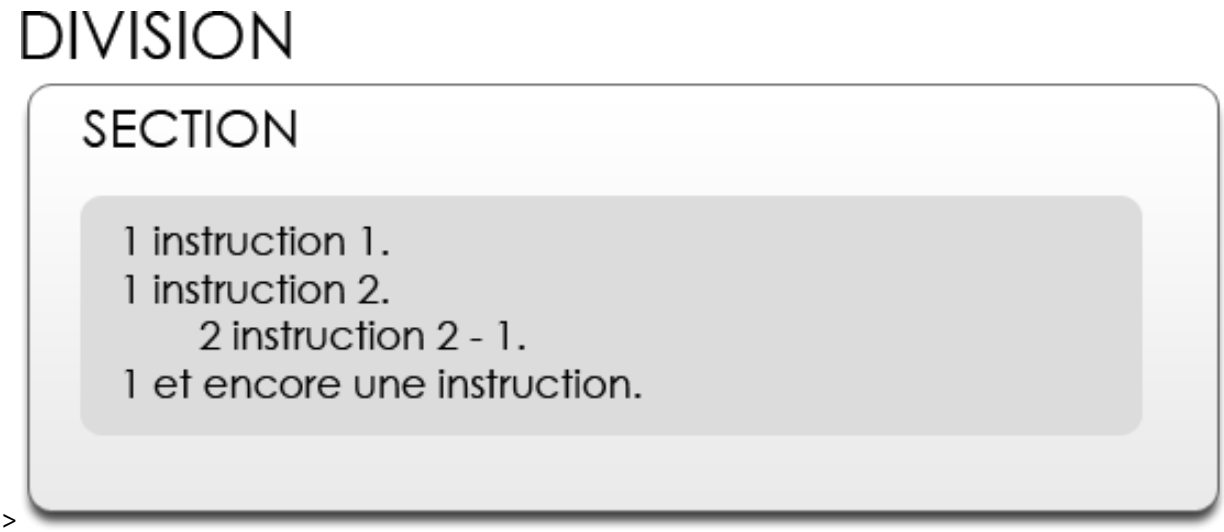
Notez bien les points à la fin de chaque phrase. <-

[[question]] | Ok, des étiquettes, des phrases… Mais c'est quoi ?!

C'est tout bête, l'étiquette va permettre de hiérarchiser vos phrases. Celle-ci se distingue par un

nombre entier, la première valeur étant 1, jusqu'à XX.
Et les phrases sont simplement des instructions pour ceux qui ont fait de la programmation. Et pour ceux qui n'en ont jamais fait, les instructions sont des directives que l'on donne à notre programme pour lui dire ce qu'il doit faire.

Re-voyons le petit schéma de tout à l'heure de manière un peu plus concrète :



->
<-

Vous avez sûrement remarqué un petit quelque chose, on a imbriqué une étiquette et phrase dans une autre : sachez que c'est possible et régulièrement utilisé. Nous nous pencherons plus en détail là-dessus en temps voulu.

[[attention]] | Encore une fois je vous avertis, ça n'a l'air de rien comme ça mais croyez-moi, l'oubli du point à la fin des phrases est assez fréquent !

2.2.1.2 Les divisions

Un programme en Cobol est toujours composé de plusieurs divisions, je vais vous présenter les plus courantes mais nous en verrons d'autres d'ici la fin du cours.

2.2.1.2.1 En tête du programme L'en-tête n'est pas obligatoire pour passer la compilation, mais elle est quand même utile pour avoir un minimum d'informations sur le programme qui se trouve sous vos yeux, car quand c'est frais dans votre tête ça passe, mais dans plusieurs mois ça ne sera plus forcement le cas...

Il est temps de voir vos premières (ou pas) lignes de code :

```
IDENTIFICATION DIVISION.
PROGRAM-ID. Votre-programme.
```

Je pense qu'il n'y a pas grand chose à expliquer ici, car il n'y a rien à comprendre. Ici `IDENTIFICATION DIVISION` est le nom de la division qui permet de créer l'en-tête, et `PROGRAM-ID.` est suivi du nom de votre programme et d'un point.

2.2.1.2.2 Partie déclarations C'est là que nous allons stocker toutes les variables que l'on va utiliser dans notre programme, **ici et jamais ailleurs** !

```
DATA DIVISION.
WORKING-STORAGE SECTION.
   77 nomVariable PIC type.

SCREEN SECTION.
   1 instruction.
```

Observons ce code d'un peu plus près. Pour les deux premières lignes, il n'y a pas vraiment de questions à se poser, c'est comme ça que l'on déclare la zone où nos données seront stockées (comme le terme `DATA DIVISION` nous le laisse d'ailleurs penser).

Jetons un œil du côté de ce bout de code maintenant :

```
WORKING-STORAGE SECTION.
   77 nomVariable PIC type.
```

`WORKING-STORAGE SECTION` est la section dans laquelle on va déclarer toutes les variables du programme. Les détails sur la syntaxe et les mots clés seront dévoilés un peu plus bas dans cette page, le but étant ici de vous expliquer comment est construit le code en Cobol.

Passons à `SCREEN SECTION` :

```
SCREEN SECTION.
   1 instruction
```

C'est ici que va se trouver le code qui sera chargé de gérer les entrées et sorties. Ou autrement dit, c'est ici que l'on mettra tout ce qui est affichage et entrées au clavier.

2.2.1.2.3 Partie instructions Et là, on trouvera... les instructions ! (Non ce n'est pas une blague ! :-°)

```
PROCEDURE DIVISION.
instructions une.
instructions deux.
```

C'est dans `PROCEDURE DIVISION` que votre programme va trouver ce qu'il est censé faire, on y placera nos boucles, conditions, etc.

2.2.1.2.4 Synthèse et fin de programme Voilà pour le principe général, si on rassemble tout ce qu'on a vu jusqu'à présent nous obtenons ceci :

“‘cobol hl_lines=“16” IDENTIFICATION DIVISION. PROGRAM-ID. Votre-programme.

```
DATA DIVISION.
WORKING-STORAGE SECTION.
   77 nomVariable PIC type.

SCREEN SECTION.
   1 instruction.

PROCEDURE DIVISION.
instruction1.
instruction2.
etc.

STOP RUN.
```

“‘

Remarquez qu'à la ligne 16 j'ai noté une nouvelle instruction, `STOP RUN`. Cette ligne permet simplement de signaler la fin de votre programme.

2.2.1.3 Les commentaires

Les commentaires sont un point essentiel en programmation et j'aimerai aborder ce dernier point avec vous.

Pour un petit programme de quelques lignes ce n'est pas très utile, mais quand il y en a des centaines, c'est indispensable pour vous y retrouver ou pour l'éventuelle personne qui relira votre code ! Évidemment les commentaires ne sont pas interprétés par le compilateur, c'est uniquement destiné aux yeux des développeurs ! ;)

2.2.1.3.1 NetExpress Les commentaires se signalent dans la marge avec une étoile (*), comme ceci :

```
* Et hop ! Un commentaire !
* Oh ! Une deuxième ligne !
```

Figure 2.1 –

2.2.2 Les variables

Comme dit précédemment, les déclarations de nos variables se font dans `DATA DIVISION`, et plus précisément dans `WORKING-STORAGE SECTION`.

Revoyons le petit bout de code de tout à l'heure :

```
WORKING-STORAGE SECTION.
   77 variable PIC 9.
```

Décomposons un peu cette bizarrerie…

— On retrouve l'étiquette, ici : 77

[[information]] | Je vous entends déjà hurler en me demandant "Mais pourquoi 77 ?!", et bien sachez que c'est tout simplement une convention de nommage pour les variables en COBOL. Autant vous habituez tout de suite puisque dans tous ce cours nos variables seront déclarés avec l'étiquette 77. ;)

— Ensuite on retrouve le nom de notre variable, sachant que son nom doit être composé seulement de caractère alphanumérique.
— Passons à : `PIC 9`
`PIC` est un mot clé qui permet de décrire la **forme** qu'aura votre variable. Et le chiffre 9 sert à indiquer le type de donnée que la variable va contenir. Ici 9 correspond à un chiffre entier (de 0 à 9).
Pour en revenir à la **forme**, elle correspond à la manière de représenter la variable. Par exemple si la variable vaut 457.24 (nombre au pif) la forme sera 999v99. Ici v marque la présence d'une virgule.

Je vous propose une tableau pour synthétiser la déclaration des types de manière simple :

En COBOL	Correspondance
9	Nombre à 1 chiffre
9(5)	Nombre à 5 chiffres
a(10)	Chaîne de 10 caractères alphabétiques
x(25)	Chaîne de 25 caractères alphanumériques
9v9	Nombre à 1 chiffre et 1 décimale
9(3)v9(2)	Nombre à 3 chiffres à 2 décimales
s9(3)	Nombre à 3 chiffres signé (+) ou (-)

2.2.2.0.1 Un mot sur les structures Vous pouvez également créer des structures de données avec vos variables, comme ceci :

```
1 personne.
   2 nom PIC x(30).
   2 prenom PIC x(30).
   2 adresse.
      3 numero PIC 9(3).
      3 rue PIC x (40).
      3 code postal PIC x(5).
      3 ville PIC x(20).
```

Je ne vous en dis pas plus pour l'instant nous verrons cela plus tard en détail mais sachez juste que ça existe.

2.2.3 Les plages

Les plages sont une notion indispensable dans un programme (pas de panique, c'est très simple), sans elles votre application n'aurait aucun dialogue avec l'utilisateur ; or de nos jours, c'est primordial ne serait-ce que pour informer du bon ou mauvais déroulement du programme.

Les plages permettent de gérer les entrées et sorties de l'ordinateur, l'entrée étant le clavier (ce que l'utilisateur va taper) et la sortie étant l'affichage à l'écran. On va pouvoir observer 2 types de plage pour l'instant, les plages dîtes d'**affichage**, et celles de **saisie**.

2.2.3.0.1 Plage d'affichage

Regardez attentivement ce code :

```
SCREEN SECTION.

1 a-plg-titre.
    2 BLANK SCREEN.
    2 LINE 3 COL 15 VALUE 'Hello world !'.
```

La gestion des entrées/sorties se fait dans `SCREEN SECTION`, donc on déclare ligne 1 ce que l'on va faire.

Ensuite, comme je vous l'ai dit tout à l'heure on doit mettre des étiquettes avant chaque instruction. Et ici nous sommes à la racine de la hiérarchie, donc l'étiquette logique sera 1.

[[information]] | J'emploie le terme "logique" car évidemment vous pouvez mettre n'importe quel chiffre. Mais mieux vaut se plier aux bonnes pratiques directement.

Ce qui suit l'étiquette n'est rien d'autre que le nom de notre paragraphe, ici **a-plg-titre**, autrement dit affichage-plage-titre. C'est ce nom qui sera utilisé pour appeler l'affichage.

Ligne 4 nous descendons d'un niveau dans la hiérarchie, donc l'étiquette est 2. Si on avait besoin de descendre plus bas, ça serait 3, puis 4, puis 5, etc. `BLANK SCREEN` permet d'effacer ce qui est affiché à l'écran.

Ligne 5, nous sommes toujours au même niveau que la ligne 4, donc l'étiquette sera encore une fois égale à 2. C'est là que ça devient intéressent ! `LINE` en anglais veut dire ligne, cela signifie donc qu'on se positionne à la ligne 3, et `COL` pour colonne. Donc on commencera à afficher notre texte à la ligne 3, colonne 15.

[[information]] | Selon votre compilateur, le mot COL est interprété comme étant une erreur. Dans ce cas vous devez remplacer COL par COLUM.

Le positionnement de votre texte est obligatoire, ce n'est pas très commun dans un langage de programmation de le faire de cette manière, mais... c'est du Cobol ! :D

Passons à `VALUE`, qui signifie valeur. Ce qui suit ce mot-clé sera donc ce qui sera affiché dans la console. Il suffit donc d'écrire notre texte entre des guillemets simples comme ceci : `'Hello world !'` sans oublier notre **point** !

2.2.3.0.2 Plage de saisie

Toujours dans `SCREEN SECTION`, on place nos plages de saisie :

```
1 s-plg-nom.
   2 LINE 5 COL 8 VALUE 'Quel est votre nom : '.
   2 PIC x(25) TO nom REQUIRED.
```

Rien de plus compliqué qu'avant pour l'instant aux lignes 1 et 2, s-plg-nom, pour saisie-plage-nom, nous placerons le texte indicatif à la 5e ligne, 8e colonne.

C'est ici qu'il y a des nouveautés, pour rappel voici ce que je vous ai dit à propos du mot-clé `PIC` :

> PIC est un mot-clé qui permet de décrire la forme qu'aura votre variable.

[[question]] | Mais indiquer la forme de la variable au début, ça ne suffit pas ? :-°

Eh bien… Non ! Ici, donner la forme de la variable n'influera pas sur la variable en elle-même mais sur la manière dont la saisie va être affichée. Et c'est aussi valable pour les plages d'affichage lorsque l'on voudra afficher la valeur d'une variable !

Voici une capture d'écran qui devrait vous aider à comprendre le fonctionnement des formes :

Figure 2.2 –

[[information]] | Quand vous avez plusieurs champs à remplir avec des entrées clavier, vous pouvez utiliser la touche ||TAB|| pour passer au champ suivant, sinon utilisez les flèches.

Ensuite, x(25) va permette d'entrer une chaine de 25 caractères qui sera récupérée dans nom via le mot-clé **TO**. Il y a également autre chose de nouveau, le mot **REQUIRED** permet de rendre obligatoire la saisie du nom pour continuer le programme.

2.2.3.0.3 Résumons, vous voulez bien ?

Voici ce à quoi devrait ressembler le code dans sa globalité :

"'cobol hl_lines="20" IDENTIFICATION DIVISION. PROGRAM-ID. HelloWorld. DATA DIVISION. WORKING-STORAGE SECTION. 77 nom PIC x(25).

```
SCREEN SECTION.
1 a-plg-titre.

2 BLANK SCREEN.     2 LINE 6 COL 10 VALUE 'Hey !'.

1 s-plg-nom.
    2 LINE 8 COL 8 VALUE 'Quel est ton nom ? '.

2 PIC x(25) TO nom REQUIRED.

1 a-plg-nom.

2 LINE 10 COL 8 VALUE 'Salut'.     2 COL 15 PIC x(25) FROM nom. "'
```

Je vais profiter du résumé pour introduire une nouvelle fois un mot clé… :-°

Vous avez très certainement vu que j'ai ajouté une plage, regardez la ligne surlignée, j'ai créé une nouvelle plage qui va nous permettre d'afficher le nom que l'utilisateur a entré. Le mot-clé `FROM` permet simplement de récupérer la valeur que contient la variable `nom`.

Si on synthétise, on a donc : - `TO` pour attribuer une valeur à notre variable ; - `FROM` pour récupérer la valeur d'une variable.

2.2.4 Les procédures

C'est dans les procédures que va se situer le cœur de notre programme ! C'est ici que l'on va écrire toutes les instructions du programme.

On va continuer sur l'exemple de tout à l'heure, on va demander à notre utilisateur d'entrer son nom et ensuite nous lui dirons bonjour. Vous verrez, c'est facile !

[[information]] | Je change les conventions de nommage de mes plages afin de vous habituer aux différentes possibilités, ici pla : plage d'affichage et pls : plage de saisie.

```
PROCEDURE DIVISION.

* On affiche le nom de notre programme
DISPLAY pla-titre.

* Ensuite on affiche le formulaire de saisie
DISPLAY pls-nom.
ACCEPT pls-nom.
DISPLAY pla-nom.

STOP RUN.
```

2.2.4.0.1 Examinons cela Quelques nouveautés sont à expliquer ici. Je pense que pour la première ligne ça devrait aller, on a juste déclaré qu'on allait écrire nos instructions.

```
DISPLAY pla-titre.
DISPLAY pls-nom.
ACCEPT pls-nom.
DISPLAY pla-nom.
```

On retrouve plusieurs fois le mot clé `DISPLAY`, ce qui signifie "Afficher". En gros, on lui dit "affiche moi cette plage" ! Donc sur les lignes 1, 2 et 4 on affiche enfin les plages que nous avons créées tout à l'heure.

Regardez ensuite la ligne 3, elle contient une instruction très importante ! Celle-ci permet en effet d'accepter les valeurs entrées et de les mettre en mémoire.

Et à la fin on a l'instruction `STOP RUN` qui arrête le programme tout bêtement.

2.2.4.0.2 Bonus : Les étiquettes Voyons ensemble un dernier exemple !

Je vais vous donner un code complet, essayez de comprendre ce que j'ai modifié par rapport à la manière que je vous ai décrite tout à l'heure :

"'cobol hl_lines="13 17 22" IDENTIFICATION DIVISION. PROGRAM-ID. ExempleEtiquette.

```
DATA DIVISION.
WORKING-STORAGE SECTION.
77 a PIC 9(15).
```

```
  SCREEN SECTION.
  1 pls-exemple.
```

2 BLANK SCREEN. 2 LINE 6 COL 10 VALUE ‘Saisir une valeur : ’. 2 saisie PIC z(15) TO a REQUIRED.

```
  1 pla-exemple.
```

2 LINE 8 COL 10 VALUE ‘Resultat : ’. 2 COL 30 PIC z(15) FROM saisie.

```
  PROCEDURE DIVISION.

  DISPLAY pls-exemple.
  ACCEPT saisie.
  DISPLAY pla-exemple.

  STOP RUN.
```

“‘

Vous avez vu ? J’ai ajouté quelque chose ligne 13, cette chose s’appelle **une étiquette**, ici j’ai décidé de l’appeler **saisie** mais c’est bien-entendu au choix. Elle vous permet d’accéder à la valeur de **a** sous un autre nom tout simplement. Le résultat sera donc :

```
Saisir une valeur : 42
Résultat : 42
```

2.2.5 Manipulations

Nous allons voir comment faire certaines manipulations sur nos variables. Cobol propose des outils très simples pour cela et on va en voir quelques uns.

Voilà le code que l’on va poser afin de poursuivre notre apprentissage :

```
      IDENTIFICATION DIVISION.
      PROGRAM-ID. Manipulation.

      DATA DIVISION.
      WORKING-STORAGE SECTION.
      77 a PIC 99.
      77 b PIC 99.
      77 c PIC 99.

      SCREEN SECTION.
      * On créera nos plages ici
```

```
      PROCEDURE DIVISION.
      * Idem pour les procédures

      STOP RUN.
```

2.2.5.0.1 MOVE

Celui-ci permet d'attribuer une valeur à une variable, comme ceci :

"'cobol hl_lines="20" IDENTIFICATION DIVISION. PROGRAM-ID. Manipulation.

```
  DATA DIVISION.
  WORKING-STORAGE SECTION.
  77 a PIC 99.
  77 b PIC 99.
  77 c PIC 99.


  SCREEN SECTION.

  1 pla-res.

  2 BLANK SCREEN.    2 LINE 5 COL 10 'C vaut : '.     2 PIC 99 FROM c.

  PROCEDURE DIVISION.

  MOVE 37 TO c.

  DISPLAY pla-res.

  STOP RUN.
```

"'

Résultat :

```
C vaut : 37
```

Vous pouvez également importer une valeur dans plusieurs variables : `MOVE 5 TO a b c.`

2.2.5.0.2 COMPUTE

Permet simplement d'effectuer des calculs sur une ou plusieurs variables.

"'cobol hl_lines="25" IDENTIFICATION DIVISION. PROGRAM-ID. Manipulation.

```
  DATA DIVISION.
  WORKING-STORAGE SECTION.
  77 a PIC 99.
  77 b PIC 99.
```

```
  77 c PIC 99.

  SCREEN SECTION.

  1 pla-res.
     2 BLANK SCREEN.
```

2 LINE 5 COL 10. 2 PIC 99 FROM a. 2 COL 12 ‘ + ’. 2 PIC 99 FROM b. 2 COL 18 ‘ = ’. 2 PIC 99 FROM c.

```
  PROCEDURE DIVISION.

  MOVE 5 TO a.
  MOVE 15 TO b.
  COMPUTE c = a + b.

  DISPLAY pla-res.

  STOP RUN.
```

“‘

Résultat :

```
05 + 15 = 20
```

Bien sur ici j’ai pris l’exemple de l’addition, mais toutes les opérations de base fonctionnent, j’aurais très bien pu faire `COMPUTE c = 3 - a / (2 * b) + 1.`;)

2.2.5.0.3 ADD `ADD` va nous permettre par exemple d’incrémenter une variable **qui a déjà été initialisée**. Voyons ça :

“‘cobol hl_lines=“18” IDENTIFICATION DIVISION. PROGRAM-ID. Manipulation.

```
  DATA DIVISION.
  WORKING-STORAGE SECTION.
  77 a PIC 99.

  SCREEN SECTION.

  1 pla-res.

  2 LINE a COL 10 ‘Valeur de a : ’.  2 PIC 99 FROM a.

  PROCEDURE DIVISION.

  MOVE 5 TO a.
```

```
  DISPLAY pla-res.
  ADD 2 TO a.
  DISPLAY pla-res.

  STOP RUN.
```

‘‘‘

Résultat :

```
Valeur de a : 5

Valeur de a : 7
```

Voilà c’est tout simple ! J’attire également votre attention sur la ligne 11, il y a là un p’tit quelque chose dont je ne vous ai pas encore parlé, mais l’occasion s’est présentée avec `ADD`. :)

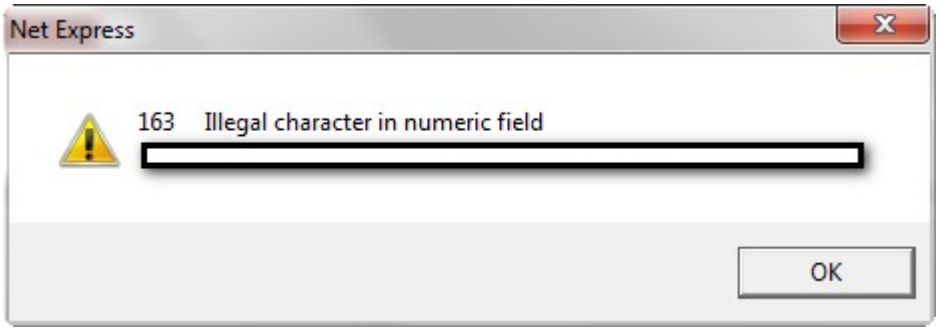
Vous avez dû le comprendre si vous avez bien fait attention au code que je vous ai donné, mais quand vous indiquez les lignes et les colonnes de votre affichage, vous pouvez le faire via des nombres entiers comme nous l’avons fait jusqu’à présent, et vous pouvez également passer par des variables qui contiennent un entier !

2.2.5.0.4 INITIALIZE Cette fonction permet d’initialiser une variable à la valeur dite “*null*” qui correspond au type de la variable sur laquelle on utilise cette fonction.

Voilà un exemple sans `INITIALIZE` :

```
IDENTIFICATION DIVISION.
PROGRAM-ID. Manipulation.

DATA DIVISION.
WORKING-STORAGE SECTION.
77 a PIC 99.

SCREEN SECTION.

1 pla-ini.
    2 BLANK SCREEN.
    2 LINE 5 COL 10 'Pour a : '.
    2 PIC 99 FROM a.

PROCEDURE DIVISION.

DISPLAY pla-ini.

STOP RUN.
```

Essayez de compiler et d’exécuter… Et là ! Un beau message d’erreur apparaît :

-> 

<-

Ré-essayons avec INITIALIZE :

“‘cobol hl_lines=“4” * … PROCEDURE DIVISION.

```
  INITIALIZE a.
  DISPLAY pla-ini.

  STOP RUN.
```

“‘

Et là ça marche ! Voilà le résultat :

```
Pour a : 00
```

2.2.5.0.5 Attention… Interro surprise ! Pour vous entrainer je vous invite à essayer de faire un programme qui demande 2 chiffres à l’utilisateur et ensuite qui calcule le produit des 2 chiffres. Je ne vous donnerai pas la solution, c’est très simple et vous avez tout le nécessaire dans ce chapitre, si jamais vous avez du mal ! ;)

Pour vos premiers pas avec COBOL, vous avez réalisé un programme simple qui récupère une valeur saisie et qui vous l’affiche à l’écran. C’est déjà pas mal, et ça doit vous changer si vous connaissez déjà d’autres langages ! :-°

Ce chapitre est un peu dense, alors n’hésitez pas à y revenir en cas d’oubli.

Pour vous aider, voici un rappel sur les erreurs à éviter :

— Faites bien attention aux points, qui sont un oubli très fréquent !
— Si vous voulez afficher ou effectuer certaines opérations sur une variable, initialisez-la au préalable.

Dans le prochain chapitre, nous allons aborder les conditions.

2.3 Les conditions

Les conditions sont un concept simple et repris dans pratiquement (tous ?) les langages de programmation, et nous allons voir comment cela fonctionne en COBOL.

2.3.1 Paragraphe et booléen

2.3.1.1 Notion de paragraphe

L'utilisation de paragraphe peut être utile pour répéter certaines portions de code sans avoir à les copier/coller (comme une sorte d'include en PHP). Un paragraphe peut contenir des plages, des conditions, etc. En bref, un paragraphe est constitué d'une suite d'instructions diverses.

Tout ce passe encore une fois dans `PROCEDURE DIVISION`, on va **déclarer** notre paragraphe :

```
mon-paragraphe.
    * instruction 1.
    * instruction 2.
    * instruction 3.
    * etc.
```

Comme vous le voyez, afin de définir le nom de notre paragraphe on va lui attribuer une étiquette qui va nous être utile au moment où on va vouloir réutiliser la portion de code.

[[information]] | Un paragraphe se termine **toujours** par un point !

Ensuite pour **appeler** notre paragraphe il suffit de faire ceci :

```
PERFORM mon-paragraphe.
```

2.3.1.2 Les booléens

Je ne vous en ai pas parlé plus tôt mais en Cobol, il est tout à fait possible d'utiliser des booléens, voici la manière de procéder :

2.3.1.2.1 Déclaration

```
DATA DIVISION.

77 note PIC 99v99.
88 parfait VALUE 20.
```

Décodons un peu ces quelques lignes, comme vous pouvez le constater j'ai créé une variable note qui est de type numérique avec 4 chiffres dont 2 décimales ; jusque là, rien de nouveau.

Si vous regardez la ligne du dessous, on a 88, c'est une convention pour déclarer nos booléens. S'ensuit le nom de notre booléen et la valeur qui fait que notre booléen soit à `true`. Pour que parfait soit à `true` il faut que la valeur de note soit de 20.

Étoffons un peu notre exemple comme ceci :

```
DATA DIVISION.

77 note PIC 99v99.
88 passable VALUE 10 THRU 11.99.
```

```
88 assez_bien VALUE 12 THRU 13.99.
88 bien VALUE 14 THRU 16.
88 parfait VALUE 20.
```

Ici, il y a une petite nouveauté avec le mot clef **THRU** : si on traduit la ligne 4 en français, cela signifie “Si la valeur de note va de 10 à 11,99 passable vaut true, sinon faux”.

Vous l’aurez peut-être compris, mais toutes les déclarations de booléen après une variable sont rattachées à cette même variable. Dans notre exemple : passable, assez_bien, bien, parfait sont rattachées à la variable note.

Gardez cela dans un coin de votre tête, on va en avoir besoin très bientôt ! ;)

2.3.2 La base “if ... else”

Voici le code sur lequel on va se baser pour cette partie :

```
IDENTIFICATION DIVISION.
PROGRAM-ID. conditions.
DATA DIVISION.

77 val PIC 999.

SCREEN SECTION.

01 pla-titre.
   02 BLANK SCREEN.
   02 LINE 2 COL 25 VALUE "Superieur ou inferieur".

01 pla-sup.
   02 LINE 6 COL 10 VALUE "La valeur de 'val' est superieur ou".
   02 VALUE " egale a 100".

01 pla-inf.
    02 LINE 6 COL 10 VALUE "La valeur de 'val' est inferieur a 100".

PROCEDURE DIVISION.

MOVE 50 TO val.
DISPLAY pla-titre.

* C'est dans ce bloc que l'on va placer nos conditions.

STOP RUN.
```

Les conditions n’ont rien de plus compliqué qu’ailleurs, la syntaxe se rapproche un peu des scripts Batch.

2.3.2.0.1 Exemple classique

```
IF val < 100 THEN
   DISPLAY pla-inf
ELSE
   DISPLAY pla-sup
END-IF.
```

[[erreur]] | Attention, vous voyez qu'il n'y a pas de point à l'intérieur de la condition ! C'est une erreur d'en mettre !

En effet, dans cet exemple c'est pla-inf qui sera affiché sur votre écran mais si vous essayez de changer la valeur insérée dans val et que vous mettez 125, ce sera pla-sup qui s'affichera.

2.3.2.0.2 Exemple avec un paragraphe Comme je vous l'ai expliqué plus haut un paragraphe permet d'exécuter un bloc d'instructions, voyons un exemple concret :

```
evaluation.
IF val < 100 THEN
   DISPLAY pla-inf
ELSE
   DISPLAY pla-sup
END-IF.

* Du code...
* Encore du code...

MOVE 150 TO val.
* Et vous avez besoin de réutiliser le code de l'évaluation alors vous appelez v
PERFORM evaluation.
```

Dans ce cas vous aurez un message d'erreur, mais si vous retournez sur la console vous verrez bien :

```
Superieur ou inferieur

La valeur de 'val' est superieure ou egale a 100
```

On est d'accord, dans cet exemple ça n'a pas de réel intérêt mais c'est simplement pour vous montrer leur utilisation.

2.3.2.0.3 Imbrication Bien sûr, vous pouvez imbriquer vos conditions de cette manière :

```
cobol hl_lines="4"       IF x < 50 THEN           * ...       ELSE            IF a > 1
* ...           ELSE                * ...           END-IF       END-IF
```

Remarquez qu'on a une condition un peu plus complexe ici, j'ai mis `OR` (OU). J'aurais très bien pu mettre `AND` (ET) si j'avais voulu mais ici c'est juste un exemple.

2.3.2.0.4 Petite astuce Si vous avez testé ce code, vous avez probablement été confronté à un souci d'espace dû à la taille de la zone de saisie, du coup vous êtes obligé de faire un retour à la ligne comme je l'ai fait :

```
01 pla-sup.
   02 LINE 6 COL 10 VALUE "La valeur de 'val' est superieure ou".
   02 VALUE " egale a 100".
```

Pour pouvoir continuer mon texte j'ai dû réécrire à la 3e ligne `02 VALUE " egale a 100"`, voici une petite astuce pour les utilisateurs de NetExpress qui permet d'écrire cela autrement avec le signe - (moins) dans la colonne 7 (comme pour les commentaires) :

```
01 pla-sup.
    02 LINE 6 COL 10 VALUE "La valeur de 'val' est superieur ou
-"egale a 100".
```

Figure 2.3 –

2.3.2.0.5 Retour aux booléens J'espère que vous n'avez pas oublié ce que je vous ai expliqué plus haut, car nous allons y revenir ! :-°

On a vu comment déclarer nos booléens, c'est bien mais les utiliser c'est encore mieux et maintenant que vous avez le nécessaire pour les utiliser, ça serait dommage de s'en priver.

"'cobol hl_lines="31 34 37 40" * Entête du programme

```
77 note PIC 99v99.
88 passable VALUE 10 THRU 11.99.
88 assez_bien VALUE 12 THRU 13.99.
88 bien VALUE 14 THRU 16.
88 parfait VALUE 20.

SCREEN SECTION.
* On créé nos plages de test
01 pla-pass.
   02 LINE 7 COL 10 'C''est passable'.

01 pla-ab.
```

02 LINE 7 COL 10 'C''est pas mal'.

```
01 pla-b.
   02 LINE 7 COL 10 'C''est bien'.

01 pla-p.
   02 LINE 7 COL 10 'C''est parfait'.

01 pla-nul.
```

02 Line 7 COL 10 'Il n''y a pas de mention'.

```
PROCEDURE DIVISION.

MOVE 20 TO note.

IF passable THEN
   DISPLAY pla-pass
ELSE
```

IF assez_bien THEN DISPLAY pla-ab ELSE IF bien THEN DISPLAY pla-b ELSE IF parfait DISPLAY pla-p ELSE * Si aucune des conditions est à TRUE, alors il n'y a pas de mention DISPLAY pla-nul END-IF END-IF END-IF END-IF.

```
STOP RUN.
```

"'

Regardez les ligne surlignées, pour vérifier si un booléen est à `TRUE`, il suffit simplement de recopier le nom qu'on lui a attribué lors de nos déclarations dans une condition. Là, la plage affichée sera pla-p qui correspond à "C'est parfait" puisque note vaut 20 ~~(comme toutes les notes d'élèves normalement constitués)~~.

À mon avis l'exemple est assez parlant pour ne pas avoir besoin de vous donner des explications supplémentaires, mais si jamais quelque chose n'est pas clair, n'hésitez pas à relire le chapitre.

[[information]] | Dans la déclaration des plages d'affichage, vous avez peut-être remarqué des doubles guillemets simples ; il ne s'agit pas d'une erreur, mais d'une manière de pouvoir mettre des apostrophes dans une phrase.

2.3.3 Choix multiples

Les choix multiples sont comparables au switch dans les autres langages, voici donc leur syntaxe (ça rappelle entre autres le PL/SQL) :

```
EVALUATE choix
WHEN 1
* Action 1 quand la variable choix vaut 1...
WHEN 2
* Action 2 quand la variable choix vaut 2...
WHEN 3
* Action 3 quand la variable choix vaut 3...
WHEN OTHER
* Action par défaut pour toutes les autres valeurs de choix...
END-EVALUATE.
```

Donc je pense qu'il n'y a rien de spécial à dire, on va évaluer la valeur de choix à chaque `WHEN`. On retrouve la condition d'entrée à tous les `WHEN` et à la fin on met `WHEN OTHER` pour prendre tous les cas que l'on a prévu dans notre application. :)

Maintenant, si vous le voulez bien, direction les boucles !

2.4 Les boucles

Les boucles sont un des fondements de la programmation, alors voyons comment ça se passe avec les spécificités de COBOL.

2.4.1 Boucler N fois

Cette forme de boucle est la plus simple que vous pouvez rencontrer : elle permet de répéter une ou plusieurs instructions un certain nombre de fois. Ce nombre vous devez le définir vous-mêmes, voyons ça avec un exemple :

```
* Entête du code avec déclaration d'une variable numérique x PIC 99

INITIALIZE x.

PERFORM 5 TIMES
   COMPUTE x = x + 5
END-PERFORM

* Et on affiche x pour connaitre sa valeur :
DISPLAY pla-x.
```

`PERFORM` peut se traduire par “effectuer” donc là nos instructions seront effectuées 5 fois, vous remarquerez que comme les conditions, il n’y a pas de point dans les instructions d’une boucle, vous obtiendrez donc :

```
Valeur de la variable x : 25
```

Rien de plus simple, mais vous verrez que la suite est moins drôle. :diable :

2.4.2 Boucle booléenne

Maintenant nous allons passer à une autre sorte de boucle qui n’est pas beaucoup plus compliquée que l’autre. Celle-ci permet de boucler tant que la condition d’arrêt n’est pas remplie.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. boucle.
DATA DIVISION.

77 i PIC 999.
```

```
SCREEN SECTION.

01 pla-ligne.
   02 LINE i COL 10 VALUE "Ligne numero ".
   02 PIC zz FROM i.

PROCEDURE DIVISION.
MOVE 1 TO i.

PERFORM UNTIL i > 10
   DISPLAY pla-ligne
   COMPUTE i = i + 1
END-PERFORM

STOP RUN.
```

Ce qui se traduit en français par : tant que i est strictement inférieur à 10, alors on itère et on ajoute 1 à `i` à chaque fois.

J'aimerais également attirer votre attention sur la plage d'affichage, vous avez remarqué quelque chose ? Non ? Eh bien vous auriez dû ! :p

Pour l'instruction qui donne le numéro de la ligne où doit apparaitre ma phrase, j'ai utilisé la valeur de la variable `i` pour que les lignes s'écrivent les unes en dessous des autres, et éviter ainsi qu'elle s'impriment les unes sur les autres.

[[attention]] | Surtout vérifiez bien vos conditions d'arrêts, sinon vous risquez de partir dans une boucle infinie !

Vous obtiendrez alors :

```
Ligne numero 1
Ligne numero 2
Ligne numero 3
Ligne numero 4
Ligne numero 5
Ligne numero 6
Ligne numero 7
Ligne numero 8
Ligne numero 9
Ligne numero 10
```

[[information]] | Vous pouvez bien entendu utiliser `AND` et `OR` dans votre condition si vous avez besoin de faire plus complexe.

2.4.3 Boucle avancée

2.4.3.0.1 Incrémentation particulière On va voir une nouvelle manière d'incrémenter nos variables : cette syntaxe va nous permettre de ne pas faire de `COMPUTE` pour incrémenter une valeur, lors de l'itération d'une boucle avec `UNTIL`. Voilà la bête :

```
IDENTIFICATION DIVISION.
PROGRAM-ID. boucleAvancee.
DATA DIVISION.

77 i PIC 999.

SCREEN SECTION.

01 pla-ligne.
   02 LINE i COL 10 VALUE "Ligne numero ".
   02 PIC zz FROM i.

PROCEDURE DIVISION.
MOVE 42 TO i.

PERFORM VARYING i FROM 5 BY 2 UNTIL i > 20
   DISPLAY pla-ligne
END-PERFORM.

STOP RUN.
```

Et voilà le résultat :

```
Ligne numero  5

Ligne numero  7

Ligne numero  9

Ligne numero 11

Ligne numero 13

Ligne numero 15

Ligne numero 17

Ligne numero 19
```

Cette boucle permet de faire une incrémentation par 2 tant que `i` n'est pas supérieur à 20, j'ai mis 2 mais j'aurais pu prendre n'importe quelle valeur à la place. Pour les anglophobes, `VARYING` peut se traduire par "variant", donc il s'agit du choix de la variable qui va... varier.

Cependant, si vous avez bien regardé le code, à la ligne 14 on met 42 dans `i`. Et pourtant la boucle commence à 5 ! C'est à cause du mot `FROM` que `i` est remis à 5.

2.4.3.0.2 Les mots TEST BEFORE/AFTER

On va continuer sur l'exemple que je vous ai donné plus haut. Vous allez remplacer la ligne :

```
      PERFORM VARYING i FROM 5 by 2  UNTIL i > 20
```

Par :

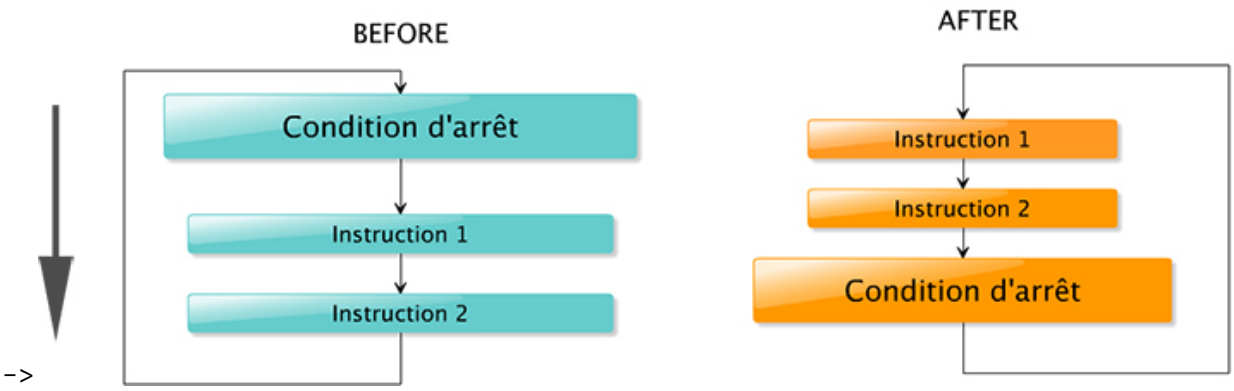
```
      PERFORM TEST AFTER VARYING i FROM 5 by 2  UNTIL i > 20
```

Compilez, et regardez le résultat :

```
Ligne numero  5

Ligne numero  7

Ligne numero  9

Ligne numero 11

Ligne numero 13

Ligne numero 15

Ligne numero 17

Ligne numero 19

Ligne numero 21
```

Vous avez vu ? On a une itération de plus ! Essayez de remplacer `AFTER`, par `BEFORE` et vous verrez que le nombre d'itérations est le même que sans les mots `TEST BEFORE`, on peut donc considérer que ça revient au même de ne pas le mettre.

Je vais vous donner un schéma pour résumer le fonctionnement d'une boucle avec `BEFORE` ou `AFTER` :



Test Avant/Après <-

Pour faire l'analogie avec d'autres langages, on peut dire que le test `BEFORE` ressemble au traditionnel "while", et `AFTER` à "do... while".

J'espère que vous êtes prêt, parce que je vais vous interroger, et pas plus tard qu'au prochain chapitre ! :-°

2.5 TP : Plus ou moins

Jeunes gens, il est temps de mettre en pratique ce que vous avez appris jusqu'à présent !

Ce TP est récurrent dans de nombreux tutoriels, mais c'est un bon exercice qui vous fait travailler les bases. Nous allons faire le jeu du "plus ou moins". Certes ce n'est certainement pas une application commune du langage COBOL, mais c'est plus amusant que de vous demander une application de comptabilité et de calculer je-ne-sais-quoi. :-°

2.5.1 Préparation

2.5.1.0.1 Pré-requis Certaines notions sont indispensables pour réussir ce TP, si vous n'êtes pas sûrs de vous, n'hésitez pas à revenir quelques chapitres en arrière !

Nous allons donc avoir besoin :

— de connaitre la syntaxe de base ;
— d'avoir bien compris les conditions ;
— et de savoir utiliser les boucles !

2.5.1.0.2 Consignes Pour ceux qui ne connaissent pas déjà le principe, voilà ce que vous devez faire.

On demande à notre joueur d'entrer **un chiffre entre 1 et 100**, et l'objectif est que le joueur trouve le chiffre fixé **de manière aléatoire** par l'ordinateur. Pour l'aider, on va lui indiquer si les chiffres qu'il tape au fur et à mesure sont **inférieurs ou supérieurs** au nombre à trouver.

Si le chiffre n'est pas le bon, on lui redemande jusqu'à qu'il le trouve. Et quand il l'aura trouvé, on lui affichera un message de félicitations pour lui dire qu'il a gagné la partie.

2.5.1.0.3 Dernières astuces Il vous manque encore un petit quelque chose pour arriver à faire le TP, on va voir comment générer un nombre aléatoire. Dans cet exemple, on génère un nombre entre 1 et 100 :

```
* Ce code se place dans PROCEDURE DIVISION.
* seed est une variable au format suivant : 9(8)
MOVE FUNCTION CURRENT-DATE(9:8) TO seed.
COMPUTE nbAleatoire = FUNCTION RANDOM (seed) * 100 + 1.
```

[[information]] | La fonction `RANDOM` génère un nombre entre 0 et 1 exclu. On a également besoin d'un paramètre pour avoir un nombre vraiment aléatoire. Ensuite on multiplie ce nombre par 100, ce qui donne un chiffre entre 0 et 99, alors on ajoute 1 pour avoir un chiffre entre 1 et 100.

2.5.1.0.4 Aide supplémentaire Si jamais vous pensez avoir bien compris ce qui vous est demandé, vous pouvez ne pas lire cette partie et vous lancer dans le code !

Sinon je vais vous aider à définir ce qu'il est nécessaire d'avoir dans votre code tout en restant assez bref, mais encore une fois, essayez de faire sans !

[[secret]] | Vous avez craqué ? Bon je vais essayer de vous guider : | | | | – Deux variables entières, une qui stock le nombre aléatoire et l'autre le nombre entré par l'utilisateur | – Au minimum 4 plages dont une de saisie pour récupérer l'entrée de l'utilisateur | – Une fois le nombre entré, on le compare à celui de l'ordinateur et selon le cas on informe l'utilisateur | – Il faut aussi une boucle pour redemander plusieurs fois à l'utilisateur de taper un nombre s'il se trompe | | | Il n'y a vraiment rien d'exceptionnel mais si vous n'avez toujours aucune idée de ce qu'il faut faire, je vous conseille de relire les parties qui vous posent problème. ^^

2.5.2 Correction

Vous avez fini ? Vous êtes sûrs ? Attention ne trichez pas, ce n'est pas dans votre intérêt. ;)

Je vais vous proposer **une** des solutions possibles afin de réaliser notre mini-jeu.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. PlusOuMoins.

DATA DIVISION.
WORKING-STORAGE SECTION.

77 seed PIC 9(8) VALUE 0.
77 nbAleatoire PIC 999.
77 nbEntree PIC 999.

SCREEN SECTION.

1 pla-titre.
    2 BLANK SCREEN.
    2 LINE 2 COL 15 VALUE 'Jeu du plus et du moins !'.

1 pla-plus.
    2 LINE 4 COL 5 VALUE 'C''est plus !'.

1 pla-moins.
    2 LINE 4 COL 5 VALUE 'C''est moins'.

1 pla-trouve.
    2 LINE 4 COL 5 VALUE 'Bravo ! Vous avez trouve !'.

1 pls-nb.
    2 LINE 6 COL 5 VALUE 'Veuillez entrer un nombre : '.
    2 PIC zzz TO nbEntree REQUIRED.


PROCEDURE DIVISION.

INITIALIZE nbEntree.
MOVE FUNCTION CURRENT-DATE(9:8) TO seed.
```

```
DISPLAY pla-titre.

COMPUTE nbAleatoire = FUNCTION RANDOM (seed) * 100 + 1.

PERFORM UNTIL nbEntree = nbAleatoire
   DISPLAY pls-nb
   ACCEPT pls-nb

   IF nbEntree > nbAleatoire THEN
       DISPLAY pla-moins
   ELSE
       DISPLAY pla-plus
   END-IF
END-PERFORM.

DISPLAY pla-trouve.

STOP RUN.
```

2.5.2.0.1 Améliorations possibles Voici une petite liste d'idées pour améliorer votre petit jeu, encore une fois c'est un TP récurrent donc les idées restent plus ou moins les mêmes :

- Proposer différents intervalles de valeurs possibles
- Afficher le nombre d'essais à l'utilisateur, une fois le nombre trouvé
- Demander à refaire une partie quand la partie est finie
- ...

2.5.3 Bonus : un peu de couleur

Jeunes gens, laissez-moi vous donner une petite astuce en guise de récompense après une séance de dur labeur ! Je vais vous apprendre à mettre un peu de couleur dans votre console.

Voilà le secret :

```
01 pla-erreur.
     02 LINE 5 COL 10
     "     C'est plus grand !     "
     FOREGROUND-COLOR 15
     BACKGROUND-COLOR 4.
```

Vous l'aurez compris, ceci est un message d'erreur écrit en blanc sur fond rouge, comme un message d'erreur plus ou moins traditionnel.
On a donc la propriété `FOREGROUND-COLOR` qui correspond à la couleur du texte, et `BACKGROUND-COLOR` qui correspond au fond du texte.

[[attention]] | Attention à la disposition des points, il n'y a pas de point après le texte cette fois, mais après la dernière instruction de couleur !

[[information]] | Bien sûr, vous n'êtes pas obligés d'utiliser les 2 propriétés à chaque fois, une suffit. ;)

Les couleurs sont au nombre de 16, entre 0 et 15. Et comme je vous aime bien, je vous ai préparé un tableau récapitulatif des couleurs :

Code	Couleur
0	Noir
1	Bleu
2	Vert
3	Bleu clair
4	Marron
5	Violet
6	Kaki
7	Gris clair
8	Gris
9	Bleu clair
10	Vert clair
11	Cyan
12	Rouge
13	Rose
14	Jaune
15	Blanc

Voilà ! Maintenant il ne vous reste plus qu'à donner de la couleur à notre mini-jeu ! :soleil :

Voilà pour les notions essentielles ! Même avec de bonnes bases dans d'autres langages, il faut avouer que COBOL ne ressemble vraiment pas à la majorité (syntaxiquement parlant).

Sur ce, si vous êtes encore d'attaque, vous pouvez déjà entamer la prochaine partie !

3 Notions avancées

Ici nous aborderons certains aspects plus complexes que nous offre Cobol. De bonnes bases sont requises, n'hésitez pas à retourner sur la première partie si vous vous sentez trop justes.

3.1 Les tableaux

Nous attaquons cette partie avec une autre notion : les tableaux. Ces derniers désignent une suite finie d'éléments du même type.

3.1.1 Déclaration

La déclaration d'un tableau se fait dans la `DATA DIVISION`.

Pour préciser qu'une variable est un tableau, il faut utiliser le mot-clef `OCCURS` suivi d'un nombre représentant la taille du tableau. Il contient obligatoirement un nombre fini d'éléments représentant la taille du tableau. Ces éléments peuvent être des entiers, des réels, des chaînes, etc.

Nous allons voir les différents tableaux possibles à travers quelques exemples.

3.1.1.0.1 Tableau unidimensionnel

Voici l'exemple le plus classique que vous allez trouver :

```
* En tête...
WORKING-STORAGE SECTION.

01 tableau.
   02 entier PIC 9 OCCURS 10.
```

Résumons avec un schéma :

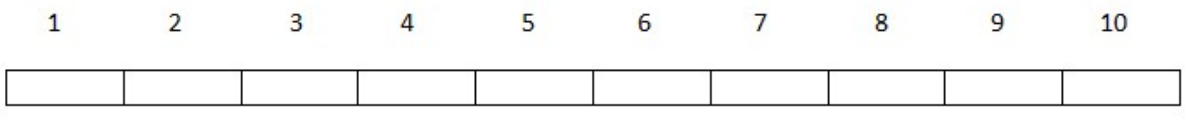


Tableau d'entiers à 1 dimension et 10 entrées <-

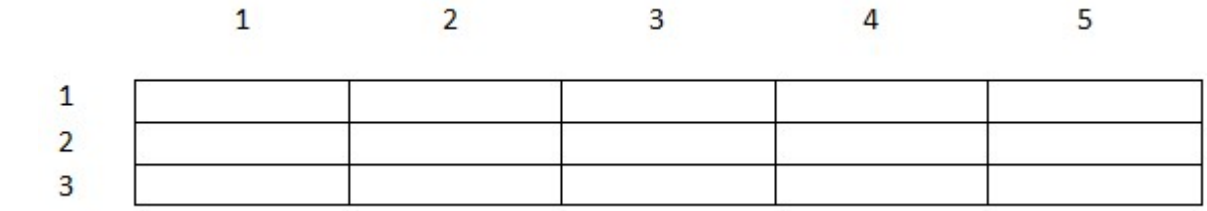
Vous voyez que sur le schéma, mes cellules sont numérotées de 1 à 10, et non de 0 à 9. **Contrairement à la majorité des langages, en Cobol un tableau commence à l'indice 1 et non 0.**

3.1.1.0.2 Tableau multidimensionnel

Voilà un exemple de tableau bidimensionnel :

```
01 tab.
   02 ligne OCCURS 3. * Le nombre de lignes
      03 cellule PIC 9 OCCURS 5.* Le nombre de colonnes
```

Vous pouvez créer jusqu'à 6 dimensions en imbriquant le mot clé `OCCURS` à chaque ligne, mais au-delà de 3, gérer son tableau devient une tâche difficile. Voilà à quoi cela ressemble :



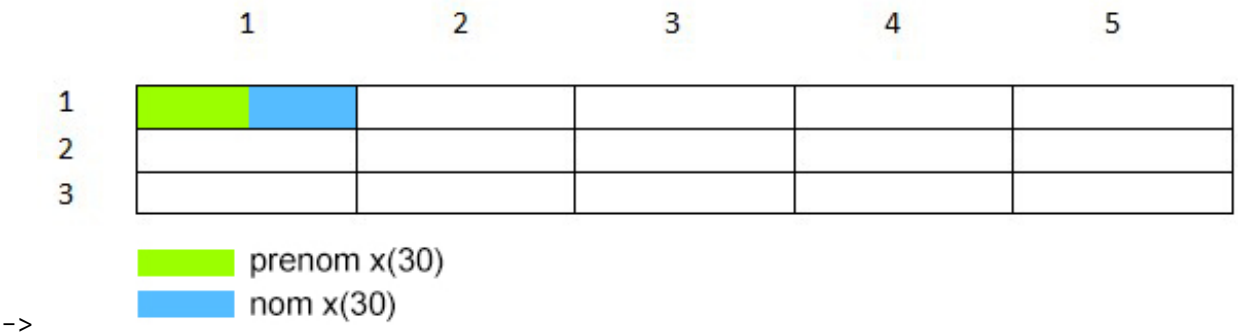
->

Tableau à 2 dimensions <-

3.1.1.0.3 Tableau de structure

```
01 tab.
   02 ligne-carre OCCURS 3.
      03 cellule OCCURS 5.
           04 prenom PIC x(30).
           04 nom PIC x(30).
```

Dans cet exemple, chaque cellule contient une structure à la place d'une simple variable ; pour faire simple, c'est comme si la cellule était divisée en plusieurs parties contenant chaque information. Encore une fois, rien ne vaut une bonne illustration :



->

Tableau de structure à 2 dimensions <-

3.1.2 Affectation

Bon, déclarer nos tableaux c'est bien, mais les utiliser c'est mieux ! :D

On va prendre l'exemple le plus simple avec une dimension, nous allons demander à l'utilisateur jusqu'à combien il veut compter et nous remplirons le tableau selon ce qu'il désire.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. Tableau.

DATA DIVISION.
WORKING-STORAGE SECTION.

77 n PIC 99.
77 i PIC 99.

01 tab.
   02 entier PIC 99 OCCURS 99.

SCREEN SECTION.

01 pls-n.
   02 BLANK SCREEN.
   02 LINE 5 COL 8 VALUE 'Valeur de n : '.
   02 PIC 99 TO n REQUIRED.

01 pla-tab.
   02 BLANK SCREEN.
   02 LINE 2.
   02 OCCURS 99.
        03 LINE + 1 COL 5 PIC zz FROM entier.

PROCEDURE DIVISION.
INITIALIZE tab.

DISPLAY pls-n.
ACCEPT pls-n.

PERFORM TEST AFTER VARYING i FROM 1 BY 1 UNTIL i = n
   MOVE i TO entier(i)
END-PERFORM.

DISPLAY pla-tab.

STOP RUN.
```

3.1.2.0.1 Sortez vos loupes !

On va analyser ce code de manière un peu plus approfondie.

Pour la partie située dans la `WORKING-STORAGE SECTION`, les explications sont données dans la partie traitant la déclaration donc je ne vais pas y revenir, il vous suffit de lire le début du chapitre.

Voyons plutôt du côté de `pla-tab` :

```
cobol hl_lines="4-5"       01 pla-tab.          02 BLANK SCREEN.          02 LINE 2.
02 OCCURS 99.               03 LINE + 1 COL 5 PIC zz FROM entier.
```

OCCURS permet de choisir le nombre de ligne que l'on veut afficher lors de l'affichage de notre

tableau. Donc là, on va afficher le tableau complet, soit 99 lignes.

La ligne suivante, permet de récupérer ligne par ligne le contenu de la variable **entier**. Vous pouvez également constater la petite subtilité qui permet de sauter une ligne à chaque affichage avec `LINE + 1`.

Passons au `PROCEDURE` :

```
PROCEDURE DIVISION.
INITIALIZE tab.

DISPLAY pls-n.
ACCEPT pls-n.

PERFORM TEST AFTER VARYING i FROM 1 BY 1 UNTIL i = n
   MOVE i TO entier(i)
END-PERFORM.

DISPLAY pla-tab.

STOP RUN.
```

Une fois que l'utilisateur a choisi le nombre qu'il veut, nous remplissons le tableau jusqu'à la valeur saisie. Nous faisons un `TEST AFTER` afin d'arriver jusqu'à l'indice n, sans lui nous irions seulement jusqu'à l'indice n-1 (rappelez-vous le schéma dans le chapitre sur les boucles).

Ensuite, nous ajoutons le numéro de l'itération dans le tableau grâce à `MOVE i TO entier(i)`.

[[information]] | J'ai utilisé `MOVE` pour attribuer une valeur à `entier(i)`, mais on peut aussi le faire avec `COMPUTE`.

[[question]] | Je ne comprends pas très bien là… Pourquoi `entier(i)` et pas `pla-tab(i)` comme dans les autres langages ?

Voilà une autre particularité du Cobol : on accède au contenu d'un tableau via le nom de la cellule, et non celui du tableau ! Donc entier(1) vaut 1, entier(2) vaut 2, etc.

Ensuite `DISPLAY pla-tab` pour afficher notre tableau, voici le résultat pour n = 10 :

```
1
2
3
4
5
6
7
8
9
10
```

3.1.2.0.2 Autre méthode Jusqu'à maintenant nous avons afficher notre tableau d'un bloc, mais vous pouvez aussi le faire ligne par ligne comme ceci :

“‘cobol hl_lines=“14” * En tete et déclaration... 01 pla-ligne. 02 LINE i. 02 COL 5 PIC zz FROM entier(i).

```
PROCEDURE DIVISION.
INITIALIZE tab.

DISPLAY pls-n.
ACCEPT pls-n.

PERFORM TEST AFTER VARYING i FROM 1 BY 1 UNTIL i = n


MOVE i TO entier(i)     DISPLAY pla-ligne END-PERFORM.


STOP RUN.
```

“‘

Le résultat est exactement le même mais c'est simplement pour vous montrer une autre manière de procéder. ;)

3.1.2.0.3 Taille des tableaux Pour l'instant je vous ai uniquement montré comment faire des tableaux à taille fixe, mais il est également possible de faire des tableaux dont la taille varie.

```
01 tab.
   02 default PIC 99.
   02 val PIC 9 OCCURS 1 TO 50 DEPENDING default.
```

De manière plus globale, la syntaxe peut se représenter de la manière suivante : `OCCURS x TO y DEPENDING z`. x symbolise la taille minimum du tableau, y la taille maximum et z la taille par défaut.

Donc z doit se situer entre x et y, sous forme plus mathématique cela se traduit par :

$$x <= z <= y$$

[[information]] | Je ne l'ai pas vraiment abordé de manière précise, mais pour les tableaux à plusieurs dimensions, l'accès à une cellule se fait comme ceci : `cellule(i,j,k)`

3.1.3 Opérations

Certaines fonctions permettent d'effectuer des opérations sur les tableaux, nous allons en voir deux.

3.1.3.0.1 Faisons le tri ! La première opération que nous allons voir c'est comment trier nos tableaux. Nous allons reprendre l'exemple de tout à l'heure en le modifiant un petit peu :

“‘cobol hl_lines=“9” PROCEDURE DIVISION. INITIALIZE tab.

```
  MOVE 15 TO n.
  PERFORM TEST AFTER VARYING i FROM 1 BY 1 UNTIL i = n
      MOVE i TO entier(i)
  END-PERFORM.

  SORT entier DESCENDING.
  DISPLAY pla-tab.
```

“‘

En exécutant notre programme, l'affichage sera :

```
15
14
13
12
11
10
9
8
7
6
5
4
3
2
1
```

Je pense que vous l'aurez compris, `SORT entier DESCENDING` permet de trier les éléments de notre tableau par ordre décroissant. Ici nous utilisons `DESCENDING` car nos éléments ont été ajoutés par ordre croissant.

Mais dans le cas où les entrées sont dans un ordre aléatoire et que l'on veut afficher notre tableau dans l'ordre, il suffit d'utiliser l'opération inverse, à savoir `ASCENDING`.

3.1.3.0.2 La recherche L'un des avantages des tableaux, c'est que l'on peut facilement rechercher l'élément qui nous intéresse grâce à une fonction toute faite.

Pour pouvoir faire une recherche il faut que notre tableau soit indexé, lors de sa déclaration nous allons ajouter ceci :

```
01 tab.
   02 entier PIC 99 OCCURS 99 INDEXED BY indice.
```

Et notre recherche dans `PROCEDURE DIVISION` se fait comme ceci :

```
SET indice TO 1.
SEARCH entier
   AT END
       DISPLAY "Element introuvable..."
   WHEN entier(indice) = 5
       DISPLAY "Element " entier(indice) " trouve ! ".
```

Pour pouvoir commencer notre recherche correctement, il faut initialiser l'indice de départ. Donc avec `SET indice TO 1` nous positionnons notre curseur de recherche à la cellule 1. Ensuite il faut indiquer dans quelle cellule on veut chercher notre élément, ce qui correspond à `SEARCH entier`.

La clause `AT END` est optionnelle, mais je la mets ici, pour signaler à notre utilisateur si la recherche ne renvoie aucun résultat, ce qui affichera "Element introuvable..." à l'écran.

Ensuite, à la manière d'une condition multiple avec le mot `WHEN`, nous allons énoncer nos conditions. Ici je cherche si 5 se trouve dans notre tableau entier. Si un élément est trouvé alors on affiche "Element 05 trouve !", sachant qu'*indice* joue le rôle de variable d'itération.

Nous avons fait le tour en ce qui concerne les tableaux en COBOL, comme d'habitude, on se retrouve au prochain chapitre.

3.2 Fonctions et sous-programmes

L'un des concepts que l'on apprend en informatique est la programmation modulaire. Dans le cas du COBOL, cette manière de programmer permet d'écrire des fonctions et/ou des sous-programmes afin d'obtenir un code réutilisable.

Il arrive très souvent que l'on doive créer de grosses applications. Ainsi grâce à l'utilisation des fonctions et/ou des sous-programmes, on peut en répartir le développement.

Donc dans ce chapitre, nous allons voir l'utilisation des fonctions intrinsèques (fonctions déjà prédéfinies), la création d'une fonction, et pour terminer la création et l'utilisation de sous-programmes.

3.2.1 Les fonctions intrinsèques

Les fonctions intrinsèques sont des fonctions toute prête à l'emploi, cela évite de réécrire bêtement des fonctions qui existent déjà. ;)

Voyons ça avec un exemple d'utilisation :

"'cobol hl_lines="30-31" IDENTIFICATION DIVISION. PROGRAM-ID. fonction.

```
WORKING-STORAGE SECTION.
01 mot PIC A(30).
01 nombre PIC 9.
01 racine PIC 9.

SCREEN SECTION.
01 plg-aff-titre.
```

02 BLANK SCREEN. 02 LINE 1 COL 10 'Utilisation des fonctions.'.

```
01 plg-saisie.
```

02 LINE 3 COL 1 'Tapez un mot en minuscule : '. 02 PIC A(30) TO mot REQUIRED. 02 LINE 4 COL 1 'Entrez un nombre entre 0 et 9 : '. 02 PIC z TO nombre REQUIRED.

```
01 plg-res.
```

02 LINE 7 COL 1 'Voici votre mot en majuscule : '. 02 PIC A(30) FROM mot. 02 LINE 8 COL 1 'La racine carre du nombre est : '. 02 PIC 9 FROM racine.

```
PROCEDURE DIVISION.
```

DISPLAY plg-aff-titre plg-saisie. ACCEPT plg-saisie.

MOVE FUNCTION UPPER-CASE (mot) TO mot. MOVE FUNCTION SQRT (nombre) TO racine.

DISPLAY plg-res.

```
GOBACK.
```

"'

Dans cet exemple, on demande à l'utilisateur d'entrer un mot en minuscule et un entier entre 0 et 9. Ensuite, on utilise la fonction `UPPER-CASE` pour mettre toutes les lettres du mot en majuscule. Et, on utilise la fonction `SQRT` pour obtenir la racine carré du nombre entré. Si le résultat n'est pas juste, alors le résultat est tronqué.

Donc, on peut voir que pour utiliser une fonction, c'est très simple. Il suffit d'écrire le mot `FUNCTION` suivi du nom de la fonction en question.

```
Utilisation des fonctions.

Tapez un mot en minuscule : canard
Entrez un nombre entier entre 0 et 9 : 9

Voici votre mot en majuscule : CANARD
La racine carre du nombre est : 3
```

3.2.1.1 Quelques fonctions utiles

Nous vous avons préparé quelques tableaux de fonctions intrinsèques, mais nous ne détaillerons pas leur utilisation précise.

Nom	Description
COS	Fonction cosinus
TAN	Fonction tangente
LOG	Fonction Logarithme naturel
LOG10	fonction Log base 10
SIN	Fonction Sinus
FACTORIAL	Factoriel n
SQRT	Fonction racine
SUM	Fait la somme de plusieurs valeurs
MOD	Modulo 2 : renvoit le reste d'une division (10 module 3 = 1 par exemple)
RANDOM	Génère un nombre entre 0 et 1 exclu

3.2.1.1.1 Fonctions mathématiques

Nom	Description
LENGTH	Retourne la longueur d'une chaine de caractères sous forme d'un entier
UPPER-CASE	Permet de passer une chaine de caractères en majuscules
LOWER-CASE	Permet de passer une chaine de caractères en minuscules
REVERSE	Inverse une chaine de caractères

3.2.1.1.2 Fonctions sur les caractères

Nom	Description
CURRENT-DATE	Retourne la date actuelle sous la forme AAAAMMJJHHMMSSCC
WHEN-COMPILED	Donne la date de compilation

3.2.1.1.3 Les dates

3.2.2 Création d'une fonction

3.2.2.1 Forme générale d'une fonction

L'utilisation des fonctions prédéfinies c'est bien, mais écrire les siennes c'est encore mieux.
Une fonction possède des paramètres formels. Une fois que la fonction a été appelée, celle-ci donne un résultat.

Voici la forme générale d'une fonction.

```
$set repository "update on"
FUNCTION-ID. nomFonction.
[file-control.]
[data division.]
[file section.]
[working-storage section.]

LINKAGE SECTION.

*> ici déclaration des paramètres en entrées
*> puis déclaration du paramètre résultat

[screen section.]

PROCEDURE DIVISION [USING liste des paramètres en entrée] GIVING paramètre résul

*> ici toutes les instructions

END FUNCTION nomFonction.
```

Tout d'abord, j'ai fait exprès de mettre entre crochets les sections qui n'ont pas d'importance ici. Pour écrire une fonction, on doit utiliser une nouvelle section : la `LINKAGE SECTION`, qui se trouve dans la `DATA DIVISION`.

Dans la `LINKAGE SECTION`, il faut mettre toutes les variables qui seront les paramètres formels de la fonction. Les variables déclarées dans cette section seront liées entre la fonction, et le programme appelant cette fonction, bien souvent le programme principal.

Une fonction peut avoir un ou plusieurs paramètres en entrée, et un seul résultat en sorti. Une fonction peut ne pas avoir de paramètres en entrée, mais renverra obligatoirement un résultat.

Sinon, vous pouvez déclarer vos variables comme vous savez déjà le faire. Il faut juste que les variables définies dans la `LINKAGE SECTION` soient identiques à celles des variables originales définies dans le programme appelant.

[[attention]] | Attention ! La clause `VALUE` ne peut pas être utilisée pour l'initialisation des variables déclarer dans la `LINKAGE SECTION`.

Ensuite, il y a deux nouvelles clauses, `USING` et `GIVING`, qui apparaissent sur la même ligne que la `PROCEDURE DIVISION`.

On écrit la clause `USING` suivie du ou des paramètres en entrée nécessaire(s) pour la fonction. Vous remarquerez qu'il y a des crochets entourant la clause `USING` et ses paramètres d'entrée. Je pense que vous avez deviné pourquoi.

Oui, on n'a pas besoin de l'écrire si la fonction n'a pas de paramètre en entrée. De plus, lorsque la clause `USING` est utilisée, l'ordre dans lequel est écrit les variables est important : il doit correspondre à l'ordre utilisé lors de l'appel de la fonction.

La seconde clause `GIVING` est suivie du nom de la variable résultat qui a été déclarée dans la `LINKAGE SECTION`.

[[question]] | Mais c'est quoi ce truc `\$set repository "update on"` tout en haut ?

J'allais justement y venir ! :)

Pour qu'une fonction puisse être utilisable, elle doit être compilée et enregistrée dans un répertoire, d'où le **repository**.

L'enregistrement de la fonction s'effectue grâce à la commande suivante : `\$set repository "update on"`. Mais cette commande va enregistrer par défaut la fonction dans le répertoire courant.

Si vous voulez enregistrer la fonction dans un autre endroit, il faut utiliser une commande supplémentaire : `\$set rdfpath "_chemin complet vers le répertoire_"`. Cette commande est à mettre avant la commande précédente. On obtient donc :

```
$set rdfpath "chemin complet vers le répertoire"
$set repository "update on"
*> définition du reste de la fonction
```

Voici la fonction. Je préfère enregistrer la fonction dans le répertoire courant.
À compiler en premier !!

```
$set repository "update on"
FUNCTION-ID. calculSomme.

LINKAGE SECTION.
01 param1 pic 99.
01 param2 pic 99.
01 paramRes pic 99.

PROCEDURE DIVISION USING param1 param2 GIVING paramRes.

COMPUTE paramRes = param1 + param2.

END FUNCTION calculSomme.
```

Et voilà le programme principal utilisant la fonction.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. progPrinc.
REPOSITORY.
FUNCTION calculSomme.

DATA DIVISION.
WORKING-STORAGE SECTION.
01 entier1 PIC 99.
01 entier2 PIC 99.
01 res PIC 99.
```

```
SCREEN SECTION.
01 plg-aff-titre.
    02 BLANK SCREEN.
    02 LINE 1 COL 20 'Somme de deux entiers'.

01 plg-saisie.
    02 LINE 4 COL 1 'Entrez le nombre 1 : '.
    02 PIC zz TO entier1 REQUIRED.
    02 LINE 5 COL 1 'Entrez le nombre 2 : '.
    02 PIC zz TO entier2 REQUIRED.

01 plg-resultat.
    02 line 8 col 1 'La somme des deux entiers est : '.
    02 PIC 99 FROM res.

PROCEDURE DIVISION.

INITIALIZE entier1 entier2 res.
DISPLAY plg-aff-titre plg-saisie.
ACCEPT plg-saisie.

MOVE FUNCTION calculSomme(entier1,entier2) TO res.
DISPLAY plg-resultat.

GOBACK.
END PROGRAM progPrinc.
```

3.2.3 Les sous-programmes

Dans cette partie, nous allons voir la création de sous-programmes. Dans un projet important, au lieu de réécrire plusieurs fois le même code, on fera plutôt plusieurs appels à un même sous-programme. C'est le principe même de la programmation procédurale. Un sous-programme possède des paramètres qui, comme pour une procédure, permettent d'échanger des données avec un programme appelant.

Écrire un sous-programme n'est pas très compliqué, c'est même très proche de l'écriture des fonctions, puisque l'on y retrouve le même principe de construction. À la différence qu'un sous-programme peut avoir un point d'entrée principal, et un ou plusieurs points d'entrées secondaires.

Je vais vous introduire les différentes notions petit à petit.

3.2.3.0.1 Définition générale d'un sous-programme

```
IDENTIFICATION DIVISION.
PROGRAM-ID. nom du sous-programme.

[file-control.]
[data division.]
```

```
[file section.]

[working-storage section.]
*> Possibilité de déclarer une variable


LINKAGE SECTION.
*> déclaration des paramètres en entrée
*> sans oublier le paramètre résultat


[screen section.]

*>point d'entrée principal
PROCEDURE DIVISION [USING BY REFERENCE ou BY VALUE paramètre en entrée] [RETURNI


*>point de sortie
GOBACK
ou
EXIT PROGRAM [GIVING variable déclaré dans la WSS]



*>point d'entrée secondaire
ENTRY 'id-entrée' [USING BY REFERENCE ou BY VALUE paramètre en entrée].

END PROGRAM nom du sous-programme.
```

On retrouve bien sûr la `LINKAGE SECTION` qui garde le même rôle que pour les fonctions. À savoir qu'il n'y a pas besoin d'enregistrer un sous-programme dans un répertoire comme on devait le faire avec les fonctions. Il suffit juste de le compiler avant de l'utiliser.

À partir du point d'entrée principal, vous pouvez voir cette ligne :

```
PROCEDURE DIVISION [USING BY REFERENCE ou BY VALUE paramètre en entrée] [RETURNING par
```

3.2.3.0.2 Passage de paramètre On retrouve la clause `USING` qui garde ici aussi le même rôle. Il y a une nouveauté qui apparaît, c'est le passage des paramètres. Il peut être envisagé par adresse ou par valeur.

`BY REFERENCE` : c'est-à-dire par adresse. Un paramètre transmis par adresse peut être partagé entre le programme appelant et le sous-programme appelé. Ce qui veut dire que toute modification du paramètre envoyé au sous-programme est répercutée au niveau du paramètre du programme appelant. Lors de l'appel, le paramètre du sous-programme se voit attribuer une adresse qui est celle du paramètre du programme principal.

`BY VALUE` : c'est-à-dire par valeur. Lors de l'appel, l'adresse du paramètre du programme appelant sera différente de celle du paramètre du sous-programme. Dans ce cas, au moment de l'appel,

la valeur du paramètre du programme principal est directement affectée au paramètre du sous-programme comme une copie. Donc, contrairement au passage par adresse, le passage par valeur permet de modifier une variable sans que cela n'affecte la variable utilisée dans le programme principal. De plus, dans ce type de transfert de paramètres, la variable ne doit pas dépasser 2 octets.

Si rien n'est spécifié au niveau du mode de passage de paramètres, alors c'est la transmission par adresse qui sera effectuée par défaut. D'ailleurs, aujourd'hui il est beaucoup moins courant de préciser le passage de paramètres. Tout simplement parce qu'on n'est plus à l'époque où l'on manquait de mémoire...

Après le passage de paramètres, on peut voir l'instruction `RETURNING`. Cette instruction fait la même chose que `GIVING` sauf que celle-ci concerne les sous-programmes.

[[attention]] | Il se pourrait que vous ayez à utiliser juste la phrase : `PROCEDURE DIVISION RETURNING` *variable* .
| Ce genre de phrase n'est à utiliser que dans les sous-programmes. En effet, utiliser cette phrase dans le programme principal de votre projet pourrait entraîner des résultats imprévisibles.

3.2.3.0.3 Appel d'un sous-programme

```
CALL id-sous-programme [USING BY REFERENCE ou BY CONTENT paramètre en entrée]

[EXCEPTION instruction-impérative1]

[NOT EXCEPTION instruction-impérative2]

[RETURNING variable]

END-CALL
```

Au moment de l'appel d'un sous-programme, on peut décider de la méthode du passage des paramètres comme on l'a vu précédemment.

[[question]] | Mais c'est quoi `BY CONTENT` ?! Encore un autre méthode de passage de paramètres ?

Non, pas du tout. Vous n'avez pas besoin de vous inquiéter, en fait cela correspond à la même chose que l'explication que je vous ai donnée pour `BY VALUE` plus haut.

[[information]] | En 2002 est apparue une nouvelle norme avec l'adaptation du COBOL vers le langage orienté objet.
| Pour ceux qui ont déjà programmé avec des langages orientés objet comme le Java, ou le C# entre autres, le concept de la gestion des exceptions est similaire.

Je ne vais pas m'étendre sur l'explication du fonctionnement des exceptions. Il faut juste savoir qu'il peut arriver que l'appel du sous-programme échoue. Cela peut se produire si ce dernier ne peut pas se charger dynamiquement, ou s'il n'y pas assez de place en mémoire. Dans ce cas-là, une exception est levée, et donc le code se trouvant après le mot-clé `EXCEPTION` sera exécuté.

Si dans le cas contraire, l'appel du sous-programme a fonctionné, alors les instructions écrites après le mot-clé `NOT EXCEPTION` seront exécutées, et ensuite la main sera passée au sous-programme.

Sachez que la gestion des exceptions en COBOL n'est en général pas très utilisée. Personnellement, je n'ai jamais eu besoin de les utiliser, mais vous savez que cela existe maintenant.

L'instruction `RETURNING` peut aussi être utilisée dans la phrase de `CALL`. Cela permet tout simplement de recevoir le résultat du sous-programme. Dans le cas où le sous-programme est quitté avec une instruction du type `EXIT PROGRAM GIVING` *variable*.

Lorsque l'appel est terminé, il ne faut pas oublier le mot clé `END-CALL`.

3.2.3.0.4 Point d'entrée secondaire

Dans la définition générale d'un sous-programme, on peut voir qu'il peut y avoir un ou plusieurs points d'entrées secondaires. Un point d'entrée secondaire est défini comme suit : `ENTRY 'id-entrée' [USING BY REFERENCE ou BY VALUE paramètre en en`

On utilise le mot clé `ENTRY` suivi du nom du point d'entrée entre apostrophes. On peut utiliser aussi la clause `USING` et préciser le mode de passage des paramètres comme on l'a vu précédemment avec le point d'entrée principal.

À chaque fois que vous définissez un point d'entrée secondaire, il ne faut pas oublier d'écrire l'instruction `GOBACK` pour indiquer la sortie du sous-programme.

Exemple :

```
ENTRY 'id-entrée' [USING BY REFERENCE ou BY VALUE paramètre en entrée].
      instruction 1
      instruction 2

      GOBACK.

ENTRY 'id-entrée' [USING BY REFERENCE ou BY VALUE paramètre en entrée].
      instruction 1
      instruction 2

      GOBACK.
```

[[attention]] | On n'a pas le droit d'utiliser le mot-clé `RETURNING` dans un point d'entrée secondaire.

3.2.3.0.5 Sortie d'un sous-programme

Que l'on se trouve au niveau du point d'entrée principal, ou d'un point d'entrée secondaire, on peut quitter un sous-programme de la même manière.

Pour sortir d'un sous-programme, on peut utiliser l'instruction `GOBACK`. On peut aussi utiliser l'instruction `EXIT PROGRAM` en rajoutant si cela est nécessaire la clause `GIVING` pour renvoyer le résultat du sous-programme.

Il est tout de même préférable de quitter un sous-programme en utilisant `GOBACK`.

```
GOBACK ou EXIT PROGRAM [GIVING variable]
```

3.2.3.0.6 Exemple 1 Voici un premier exemple de sous-programme très simple qui effectue la somme de deux entiers. J'ai choisi de faire un sous-programme n'utilisant que le point d'entrée principal pour le moment.

“‘cobol hl_lines=“12 16” IDENTIFICATION DIVISION. PROGRAM-ID.somme.

```
DATA DIVISION.
WORKING-STORAGE SECTION.
01 res PIC 999.

LINKAGE SECTION.
01 entier1 PIC 99.
01 entier2 PIC 99.

PROCEDURE DIVISION USING entier1 entier2.

   COMPUTE res = entier1 + entier2

EXIT PROGRAM GIVING res.
END PROGRAM somme.
```

“‘

Dans la première ligne surlignée, j'ai seulement utilisé la clause `USING` pour le passage de paramètres sans préciser le mode de passage.

Dans la seconde ligne surlignée, comme je n'ai pas utilisé la clause `RETURNING` au niveau du point d'entrée principal, j'ai dû utiliser l'instruction `EXIT PROGRAM GIVING` *res* pour que le sous-programme donne le résultat au programme principal. *res* étant la variable qui a été déclarée dans la WSS.

“‘cobol hl_lines=“31” IDENTIFICATION DIVISION. PROGRAM-ID.progPrinc.

```
WORKING-STORAGE SECTION.
01 entier1 PIC 99.
01 entier2 PIC 99.
01 res PIC 999.

SCREEN SECTION.
01 plg-aff-titre.
```

02 BLANK SCREEN. 02 LINE 1 COL 10 ‘Utilisation d’‘un sous-programme’.

```
01 plg-saisie.
   02 LINE 3 COL 1 'Entrez un nombre entier : '.
   02 PIC zz TO entier1 REQUIRED.
```

02 LINE 4 COL 1 ‘Entrez un nombre entier : ’. 02 PIC zz TO entier2 REQUIRED.

```
01 plg-res.
```

02 LINE 7 COL 1 'La somme des deux entiers est : '. 02 PIC 999 FROM res.

```
PROCEDURE DIVISION.
```

INITIALIZE entier1 entier2.

DISPLAY plg-aff-titre plg-saisie. ACCEPT plg-saisie.

CALL 'somme' USING entier1 entier2 RETURNING res END-CALL

DISPLAY plg-res.

```
GOBACK.
END PROGRAM progPrinc.
```

"'

Dans le programme principal, j'ai surligné la ligne la plus importante : l'appel du sous-programme. Vous pouvez voir que je n'ai rien fait de compliqué. Je n'ai ni précisé le mode de transmission des paramètres ni utilisé les exceptions. Et pourtant, ce code marche très bien. Grâce à l'instruction `RETURNING`, le résultat est directement envoyé dans la variable *res*.

Il est quand même à noter que le nom du sous-programme appelé doit être mis entre apostrophes (*simple quote* pour les habitués).

3.2.3.0.7 Exemple 2 On pourrait imaginer un sous-programme calculant les caractéristiques d'un rectangle. Pour cet exemple, j'ai choisi de créer un sous-programme où l'on passera seulement par deux points d'entrées secondaires. Un point d'entrée permettrait de calculer le périmètre d'un rectangle, et le second calculerait la surface.

Voici le sous-programme :

"'cobol hl_lines="13 16" IDENTIFICATION DIVISION. PROGRAM-ID.caractRect.

```
LINKAGE SECTION.
01 longueur PIC 99.
01 largeur PIC 99.
01 perimetre PIC 9(3).
01 surface PIC 9(4).

PROCEDURE DIVISION.
GOBACK.

ENTRY 'perimetre' USING longueur largeur perimetre.
```

COMPUTE perimetre = (longueur + largeur) * 2 GOBACK.

```
ENTRY 'surface' USING longueur largeur surface.
```

COMPUTE surface = longueur * largeur GOBACK.

```
  END PROGRAM caractRect.
```

“‘

Le code n’a rien de compliqué, il faut juste veiller à bien mette l’instruction `GOBACK` pour indiquer la sortie du sous-programme pour chaque point d’entrée créé.

Et voici le programme principal :

“‘cobol hl_lines=“30” PROGRAM-ID. progPrinc.

```
  WORKING-STORAGE SECTION.
  01 longueur PIC 99.
  01 largeur PIC 99.
  01 perimetre PIC 9(3).
  01 surface PIC 9(4).

  SCREEN SECTION.
  01 plg-aff-titre.
```

02 BLANK SCREEN. 02 LINE 1 COL 10 ‘Caracteristique rectangle’.

```
  01 plg-saisie.
     02 LINE 3 COL 1 'Entrez la longueur du rectangle : '.
```

02 PIC zz TO longueur. 02 LINE 4 COL 1 ‘Entrez la largeur du rectangle : ’. 02 PIC zz TO largeur.

```
  01 plg-res.
```

02 LINE 8 COL 1 ‘Le perimetre du rectangle est : ’. 02 PIC z(3) FROM perimetre. 02 LINE 9 COL 1 ‘La surface du rectagle est : ’. 02 PIC z(4) FROM surface.

```
  PROCEDURE DIVISION.
```

DISPLAY plg-aff-titre plg-saisie. ACCEPT plg-saisie.

CALL ‘caractRect’.

CALL ‘perimetre’ USING longueur largeur perimetre END-CALL. CALL ‘surface’ USING longueur largeur surface END-CALL.

DISPLAY plg-res.

```
  END PROGRAM progPrinc.
```

“‘

Quand vous voulez utiliser un sous-programme qui a un ou plusieurs points d’entrées secondaires, il faut d’abord appeler le point d’entrée principal du sous-programme, comme je l’ai fait ici au niveau de la ligne surlignée. Après seulement vous pouvez appeler le sous-programme à partir des points d’entrée secondaire comme vous savez le faire. :)

On a fait le tour des fonctions et des sous-programmes, de leur création à leur utilisation. Vous êtes maintenant parés pour faire de gros projets en COBOL.

3.3 Les chaines de caractères

Les chaines de caractères sont tellement classiques qu'on ne peut pas passer à côté, on les a déjà beaucoup utilisées mais vous ne connaissez pas encore tout ce qu'il est possible de faire !

[[information]] | Afin de condenser un peu les codes donnés, je ne vais pas vous redonner les plages d'affichage et de saisie à chaque fois, c'est à vous de les créer maintenant, vous devriez en être capable. ;)

3.3.1 Déclaration et caractéristiques

Comme vous le savez maintenant, les chaines de caractères se déclarent de la manière suivante :

```
* En-tête...
WORKING-STORAGE SECTION.

77 chaine PIC x(int longueur).
```

3.3.1.0.1 Les constantes On va parler un peu des **constantes**. Comme les autres langages, Cobol peut en tirer parti. Rappelons-le, le principe d'une constante est que sa valeur est constante à travers notre code.

```
* En-tête...
WORKING-STORAGE SECTION.

78 ma-constante VALUE 'Ma chaine de caracteres qui ne change jamais'.
```

Comme vous le voyez, nos constantes sont déclarées en 78, et non en 77 comme les autres variables. Ensuite comme d'habitude, le nom suivi du mot-clé `VALUE` puis notre chaîne entre apostrophes (ou guillemets simples).

Je ne vous en ai pas encore parlé pour ne pas vous trop vous bourrer la tête d'un coup, mais une constante ne contient pas nécessairement une chaîne de caractères. On peut y mettre un chiffre comme ceci :

```
* En-tête...
WORKING-STORAGE SECTION.

78 tva VALUE 19.6.
```

[[information]] | Pour afficher la valeur d'une constante, il suffit d'utiliser `DISPLAY`.

3.3.1.0.2 Comparaison de chaînes J'aimerais vous dire un petit mot sur les comparaisons. Il vous arrivera peut-être de vouloir comparer des chaînes de caractères entre elles de cette manière :

```
IF ch1 = ch2 THEN
   DISPLAY 'Chaines identiques'
ELSE
   DISPLAY 'Chaines differentes'
END-IF.
```

C'est tout à fait possible, et cela ne dépend pas de la longueur physique que vous avez choisie lors de la déclaration de vos variables, mais bien du contenu de celles-ci.

3.3.1.0.3 Affectation Nous n'en avons jamais parlé clairement, mais l'attribution d'une valeur dans une variable de type string (chaîne de caractères) se fait via le mot clé `MOVE` :

```
MOVE 'Ma chaine' TO str.
```

Mais vous pouvez également copier le contenu d'une variable pour le mettre dans une autre :

```
MOVE str1 TO str2.
MOVE constante TO str.
```

3.3.2 Manipulations avancées

3.3.2.0.1 La concaténation Comme tous les langages, il est possible de mettre des chaînes de caractères bout à bout pour former une nouvelle chaîne. Dans un premier temps, on va déclarer les variables que l'on va utiliser :

```
77 ch1 PIC x(20) VALUE 'J''aime'.
77 ch2 PIC x(20) VALUE 'le Cobol !'.
77 ch3 PIC x(40).
```

Ensuite, on va simplement les mettre les unes à côté des autres :

```
STRING ch1 ch2 INTO ch3.
```

Ce qui va vous retourner :

```
J'aime               le Cobol !
```

Si vous avez pris les mêmes valeurs que moi, vous avez sûrement remarqué qu'il y a beaucoup d'espaces entre nos 2 mots. Ceci est dû à la taille que l'on a choisie lors de la déclaration de nos variables ; si vous diminuez la taille de *ch1* à 10, il y aura moins d'espaces entre les mots.

Vous pouvez également restreindre vos concaténations en choisissant des caractères qui vont délimiter vos chaînes. Voyons cela avec un exemple :

```
STRING ch1 ch2 DELIMITED ' ' INTO ch3.
```

Ici j'ai choisi de délimiter les chaines par leur espace, donc dès qu'il y a un espace dans une des parties, on passe à la suivante à concaténer, le résultat sera :

```
J'aime le
```

Comme il y a un espace entre "le" et "Cobol !" alors la chaîne est coupée !

[[attention]] | Le contenu de `DELIMITED` est sensible à la casse, donc faites attention ! ;)

Il est aussi possible de connaître la longueur de la nouvelle chaîne, on va donc ajouter une nouvelle variable numérique :

```
77 ptr PIC 99.

*...

MOVE 1 TO ptr.
STRING ch1 ch2 DELIMITED 'C' INTO ch3 POINTER ptr.
```

Lors de l'affichage de `ptr` le résultat obtenu est :

```
Nouvelle chaine : j'aime    le
Valeur de ptr : 14
```

En réalité `POINTER` ne va pas compter uniquement les caractères normaux mais aussi les espaces. Donc si on compte la longueur de *ch1* plus la longueur de *ch2* jusqu'au "C" inclus, le résultat est donc 14 (10 + 4).

[[information]] | Les structures en Cobol sont considérées comme des chaines de caractères. Cette chaine étant le résultat de la concaténation de toutes les variables de la structure.

3.3.2.0.2 La dé-concaténation

Comme on peut concaténer, l'idéal c'est de pouvoir faire l'inverse, et c'est ce que l'on va faire avec l'aide de `UNSTRING` ! La syntaxe est très proche de celle de tout à l'heure :

```
UNSTRING chaineComplete DELIMITED ' ' INTO chaine1 chaine2 chaineN
```

Vous pouvez également appliquer les mêmes opérations avec la clause `POINTER`.

3.3.2.0.3 Test booléen Vous pouvez également faire d'autres choses, comme les tests booléens pour tester le contenu de votre chaîne. On peut par exemple savoir si la chaîne ne contient que du numérique, de l'alphabétique en majuscule, en minuscule, etc...

```
IF maChaine ALPHABETIC THEN
* Si la chaine ne contient que des lettres et des espaces
END-IF.

IF maChaine NUMERIC THEN
* Si la chaine ne contient que des chiffres et un signes
END-IF.

IF maChaine ALPHABETIC-UPPER THEN
* Si la chaine ne contient que des lettres en majuscules et des espaces
END-IF.

IF maChaine ALPHABETIC-LOWER THEN
* Si la chaine ne contient que des lettres en minuscules et des espace
END-IF.
```

Vous pouvez également ajouter la clause `NOT` juste après le nom de la variable testée, comme ceci :

```
IF maChaine NOT ALPHABETIC THEN
* Si la chaine ne contient pas que des lettres et des espaces
END-IF.

* ...
```

3.3.3 Quelques fonctions

3.3.3.0.1 REVERSE Cette fonction permet d'inverser l'ordre des lettres dans une chaîne de caractères :

```
INITIALIZE ch1 ch2.
MOVE FUNCTION REVERSE (chaine1) TO chaine2.
```

Ici, j'ai simplement mis le contenu inversé de *chaine1* dans *chaine2*. Mais j'aurais pu mettre le contenu de *chaine1* dans *chaine2* pour éviter de passer par une variable intermédiaire.

Cette fonction peut toujours vous être utile si on vous demande de regarder si un mot est un palindrome, comme kayak... :-°

3.3.3.0.2 NUMVAL `NUMVAL` permet d'effectuer une sorte de *cast* à partir d'une chaîne de caractères vers une valeur numérique. Car comme vous pouvez vous en douter, effectuer des calculs via `COMPUTE` est impossible avec une chaîne de caractères, cette fonction va le rendre possible d'une certaine manière !

```
MOVE 42 TO chaine.
INITIALIZE entier.

MOVE FUNCTION NUMVAL (chaine) TO entier.
```

Donc *entier* prendra bien la valeur numérique représentée dans la variable *chaine*.

3.3.3.0.3 MIN/MAX Celle-ci permet de récupérer le minimum ou le maximum d'une chaîne. Autrement dit, vous allez passer une liste de chaînes à une des deux fonctions et elle va vous retourner le minimum ou le maximum selon l'ordre alphabétique.

```
77 ch1 PIC x(20) VALUE 'abc'.
77 ch2 PIC x(20) VALUE 'abcd'.
77 ch3 PIC x(20).
* ...
INITIALIZE ch3.
MOVE FUNCTION MAX (ch2 ch1) TO ch3.
```

Si on affiche le contenu de *ch3* avec cet exemple, c'est "abcd" qui sera affiché car c'est la plus grande des deux valeurs. Si on avait utilisé `MIN`, ce serait "abc" qui aurait été affiché.

3.3.3.0.4 LENGTH Si vous êtes un développeur aguerri, vous devriez connaître cette fonction qui permet de récupérer la longueur d'une chaîne de caractères. Vous pourriez être tenté d'utiliser la fonction `LENGTH`, ce qui est normal.

Mais il y a un piège ! Je m'explique :

```
IDENTIFICATION DIVISION.
PROGRAM-ID. fonction.

WORKING-STORAGE SECTION.
01 mot PIC A(30).
01 nbLettres PIC 99.

SCREEN SECTION.
01 plg-aff-titre.
   02 BLANK SCREEN.
   02 LINE 1 COL 10 'Fonction LENGTH.'.

01 plg-saisie.
   02 LINE 5 COL 1 'Taper un mot : '.
   02 PIC A(30) TO mot REQUIRED.

01 plg-res.
   02 LINE 8 COL 1 'Le nombre de lettres du mot est : '.
   02 PIC 99 FROM nbLettres.
```

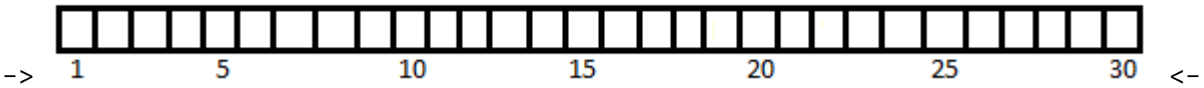
```
       PROCEDURE DIVISION.
          DISPLAY plg-aff-titre plg-saisie.
          ACCEPT plg-saisie.

          MOVE FUNCTION LENGTH (mot) TO nbLettres.

          DISPLAY plg-res.

       GOBACK.
```

```
Fonction LENGTH

Taper un mot : programmation

Le nombre de lettres du mot est : 30
```

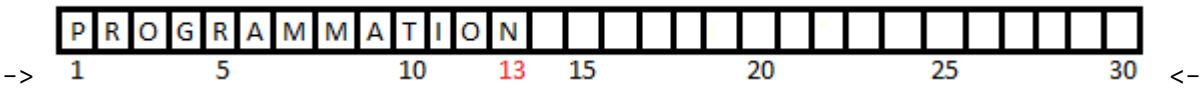
[[question]] | Quoi ?! C'est quoi ce résultat ?! o_O

Eh bien voilà, l'utilisateur entre le mot "programmation". Le résultat est 30, alors que ce mot contient 13 lettres…

D'abord, on a déclaré une variable mot pouvant contenir jusqu'à 30 caractères. Donc, c'est comme si on avait un tableau de 30 cases contenant que des cases vides.



Ensuite, l'utilisateur a saisi un mot de 13 caractères. Voici donc ce qui se passe, le mot est stocké dans le tableau, mais toutefois, il reste des cases vides.



On a donc, le mot PROGRAMMATION contenu dans les cases de 1 à 13. Puis, des blancs de la case 14 à 30. En fait la fonction `LENGTH` donne la valeur de la longueur totale de la chaîne y compris les blancs.

Alors dans ce cas, pourquoi avoir créée une telle fonction ? Je ne sais pas ! :D

[[information]] | Il existe aussi l'instruction `LENGTH OF` qui fait la même chose que la fonction `LENGTH`.

[[question]] | Mais alors il est impossible de connaître la taille réelle d'une chaîne de caractères ?

Bien sûr ! Il suffit d'utiliser l'instruction `INSPECT` qui permet entre autres de compter le nombre de caractères d'une chaîne.

Voici un exemple dans un programme.

"'cobol hl_lines="28" IDENTIFICATION DIVISION. PROGRAM-ID.LongMot.

```
  DATA DIVISION.
```

```
WORKING-STORAGE SECTION.
01 chaine PIC x(50).
01 nb PIC 99.

SCREEN SECTION.
01 plg-aff-titre.
   02 BLANK SCREEN.
```

02 LINE 1 COL 20 'Longueur d''un mot'.

```
01 plg-saisie.
```

02 LINE 4 COL 1 'Entrez un mot : '. 02 PIC X(50) TO chaine REQUIRED.

```
01 plg-res.
```

02 LINE 6 COL 1 'La longueur du mot est : '. 02 PIC 99 FROM nb.

```
PROCEDURE DIVISION.
```

INITIALIZE nb

DISPLAY plg-aff-titre plg-saisie. ACCEPT plg-saisie.

INSPECT chaine TALLYING nb FOR CHARACTERS BEFORE SPACE

DISPLAY plg-res

```
GOBACK.
END PROGRAM LongMot.
```

"'

La ligne importante de ce programme est bien sûr celle-ci : `INSPECT chaine TALLYING nb FOR CHARACTERS BEFORE SPACE.`

L'instruction va compter tous les caractères du mot un à un, et s'arrêter dès le premier espace qu'il aura trouvé. C'est à cela que sert la commande `FOR CHARACTERS BEFORE SPACE`. La clause `TALLYING` permet de placer le résultat de l'instruction dans la variable *nb*.

Donc pour synthétiser un peu tout ça, voici ce qu'il faut retenir :

- la fonction `LENGTH` ne permet pas de connaitre la "vraie" longueur d'une chaine, mais sa longueur physique ;
- les constantes se déclarent en 78, et non en 77 ;
- il est possible de comparer des chaines dans vos conditions.

4 Conclusion

À partir de maintenant vous pouvez considérer que vous avez acquis les bases du langage, pour le reste à vous de jouer. :)

COBOL a fait l'objet de nombreuses mises à jour, maintenant il est possible de faire des interfaces graphiques, de l'orienté objet ou même de faire du web et tout ce que ça comporte (XML, base de données, ...) !

4.1 Remerciements

Un grand merci à :

- Janmary pour son implication, ses conseils et pour avoir partager son expérience avec nous
- Ralys pour ses relectures attentives et ses conseils
- Coyote pour avoir migré le tutoriel vers Zeste de savoir, ses relectures et la validation
- Et aux lecteurs qui nous ont aidés à améliorer le tutoriel lors de la bêta.