Laurent Sebilleau & Daniel Varlet

Aller plus loin en AppleScript

© Laurent Sebilleau & Daniel Varlet - 2002
Tous droits réservés pour tous pays.

Dans l'esprit de la licence Open Source, vous pouvez librement copier et diffuser cet ouvrage sous sa forme intégrale et originale avec les réserves suivantes :

1- Cette autorisation n'entraîne aucune cession des droits d'auteur. Ces droits restent la propriété exclusive des auteurs. Toute modification de forme ou de fond reste soumise à l'autorisation des auteurs et devra obligatoirement comprendre la présente notice.

2- Les copies et diffusions doivent se faire gratuitement. Le diffuseur est cependant en droit de percevoir un dédommagement pour les frais de diffusion et de reproduction.

3- Toute incorporation dans un produit logiciel payant, même sous la forme originale ne peut se faire qu'avec l'accord préalable des auteurs. Cette disposition ne vise que les logiciels qui exploiteraient l'ouvrage comme données. L'ajout pur et simple de l'ouvrage sous sa forme originale à la distribution d'un logiciel payant est autorisée.

4- Les auteurs ne garantissent en aucune manière que les informations contenues dans ce manuel sont correctes, et le lecteur les utilise sous sa responsabilité et à ses risques et périls. Malgré le soin apporté à sa rédaction, cet ouvrage ne saurait remplacer les documentations officielles des constructeurs de matériel et éditeurs de logiciel sur les produits décrits. Les auteurs ne sauraient en aucune manière être tenus pour responsables des dommages directs ou indirects de toute nature pouvant survenir à la suite de l'utilisation de ces informations.

Apple et AppleScript sont des marques déposées d'Apple Computers Inc.

Introduction.

Cet ouvrage est né de la rencontre entre un programmeur AppleScript (D.Varlet) et un programmeur venant d'autres horizons (L.Sebilleau). Il recueille donc les informations que nous avons échangées, débarrassées des commentaires désobligeants et des tournures familières qui sont d'usage dans ces communications amicales et informelles. Les questions soulevées sur la liste AppleScript francophone ont également été pour nous une grande source d'inspiration.

Ces informations sont largement accessibles dans maints ouvrages et sites Internet, et bien souvent sous une forme plus complète et plus rigoureuse que ce que l'on va lire, mais, comme elles se trouvent assez dispersées, il nous a paru utile de les rassembler ici en essayant de les rendre accessibles au plus grand nombre.

C'est la raison pour laquelle cet ouvrage n'est pas un traité mais une collection un peu hétéroclite de compléments divers. Le souci de simplicité nous a conduit parfois à schématiser des choses en réalité plus complexes comme le fonctionnement d'un micro-processeur ou les structures de données utilisées par AppleScript. On gardera donc à l'esprit qu'il s'agit d'un schéma en gros exact, permettant de comprendre des principes, mais incapable de rendre compte des moindres détails[1] .

Pour toutes précisions, on se reportera donc à la documentation de référence : Inside Macintosh et les docs Intel et Motorola pour les micro-processeurs.

Nous tenons à remercier ici tous ceux qui ont bien voulu nous faire profiter de leurs avis et commentaires en acceptant de relire la version beta[2] , et plus largement tous les contributeurs de la liste AppleScript francophone. Sans eux, cet ouvrage n'aurait jamais vu le jour.

Nous espérons que vous prendrez autant de plaisir à le lire que nous avons eu à le rédiger.

---

[1] Par exemple, on ne trouvera dans ce qui suit aucune mention du registre d'état d'un micro-processeur, ce qui serait naturellement une lacune inexcusable s'il s'agissait d'un traité de programmation en assembleur.

[2] Et parmi eux, tout spécialement Nicolas Descombes qui nous a aidé de multiples manières.

La mémoire.

Un micro ordinateur est constitué d'un micro processeur, d'une mémoire et de périphériques. La mémoire est un composant passif, c'est à dire qu'elle ne peut changer d'état spontanément : c'est le micro processeur qui provoque toutes les modifications de la mémoire : ils forment donc un couple inséparable : la mémoire est l'espace de travail du micro processeur, et le micro processeur la machine qui modifie la mémoire. La mémoire et le micro processeur sont reliés par une série de lignes électriques imprimées sur la carte mère que l'on appelle le bus.

Mémoire vive et morte.

Il faut distinguer la mémoire vive (RAM) de la mémoire morte (ROM). La mémoire morte est une zone non modifiable par le micro processeur. En compensation, son contenu ne disparaît pas lorsque l'on coupe l'alimentation de l'appareil. Elle peut donc contenir des informations permanentes, et tout particulièrement ce dont l'appareil a besoin pour démarrer.

La mémoire vive est volatile, c'est à dire que son contenu disparaît lorsqu'on coupe l'alimentation électrique. Par contre, elle peut être modifiée par le micro processeur. C'est par conséquent son espace de travail privilégié : toutes vos applications actives, tous vos documents ouverts se trouvent en RAM.

Le disque dur de son côté, contient dans les fichiers, exactement les mêmes suites d'octets qu'on retrouve en RAM. Lorsque vous lancez une application ou ouvrez un fichier, le système commence par copier en RAM l'image qui se trouve sur le disque dur. Lorsque vous sauvez un document, le système copie sur le disque dur la zone de RAM où il se trouve. Les échanges avec le disque dur ne transforment pas les informations : il s'agit de copies pures et simples dans un sens ou dans l'autre.

Description de la mémoire.

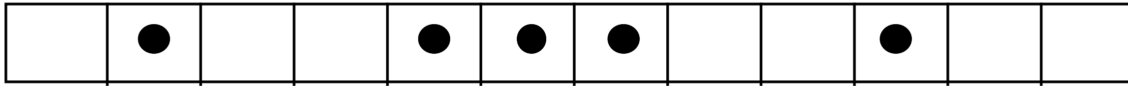
Fondements.

Dans un µordi, la mémoire est une longue série de petites cases individuelles appelées bits. Avoir 128 Mo de mémoire c'est simplement avoir une suite continue de 1 073 741 824 bits[3] . Chacune de ces cases peut avoir deux états distincts, en général codés 0 et 1 (mais on pourrait aussi bien mettre + et -, ouvert/fermé, etc…). Ici, on a utilisé des points noirs et rien du tout.

---

[3] Du point de vue des puces, ces bits sont répartis entre différents circuits intégrés sur les barrettes mémoire, mais pour le programmeur comme pour le micro processeur, les frontières sont imperceptibles et invisibles.

La mémoire se présente comme ça :



Vous voyez ce que cela représente ? Vous avez de la chance, pas moi…

Il est fondamental de comprendre que la mémoire ne contient jamais rien d'autre que ces combinaisons d'états : elle ne contient jamais de texte, d'images ou de nombres, mais des combinaisons de bits qui représentent ces nombres ou ces images selon une convention.

Ces bits sont en fait groupés au moins par huit, ce qui constitue un octet qui peut donc avoir 256 états différents (2 ^ 8 comme les matheux le vérifieront aisément), et même maintenant par seize ou trente deux (mot et double mot). On groupe les bits pour accélérer les opérations de lecture et d'écriture : les faire par paquets de huit, seize ou trente deux est plus rapide que de les faire un par un.

Une autre raison est qu'on utilise rarement les bits individuellement car ils ne permettent pas de représenter beaucoup d'information, mais seulement un système avec deux possibilités (ou deux états distincts). Un octet avec ses 256 états permet de représenter bien plus de choses, comme par exemple, tous les caractères de l'alphabet.

La mémoire se présente donc comme une suite ordonnée d'octets. Chacun de ces octets est identifié par un numéro d'ordre. Ils se suivent donc un peu comme les maisons dans une rue. Il faut simplement imaginer une rue très longue où toutes les maisons se trouvent du même côté. Comme les maisons, les octets ont donc une adresse attribuée une fois pour toute qui est la position qu'ils occupent dans la rue.

Conventions de représentation des données.

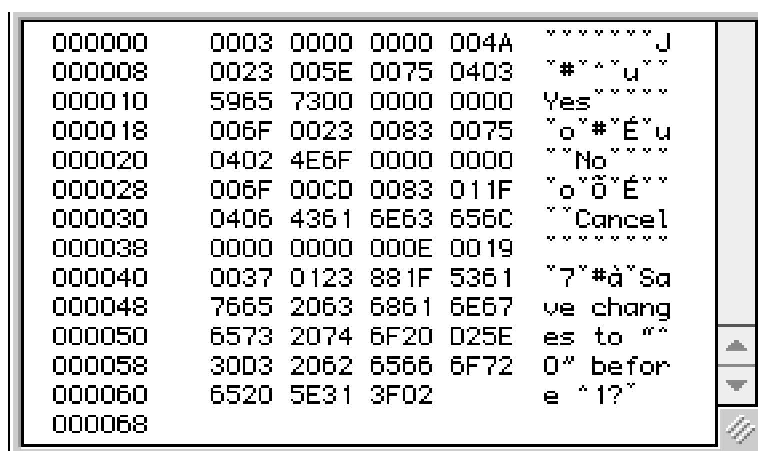
On représente donc les données par ces combinaisons de bits groupées en nombre variable, ce qui suppose toujours l'existence d'une convention (le code ASCII, les différents types de bitmaps graphiques, etc…) qu'elle soit générale et standardisée ou particulière à l'application[4] . Comme il s'agit d'une convention, elle est toujours fondamentalement arbitraire, ce qui se traduit en général par l'existence de plusieurs conventions concurrentes. Tout le monde a entendu parler du code ASCII qui sert à représenter les caractères typographiques, mais il existe une autre convention moins connue appelée EBCDIC qui fait la même chose. Le même caractère est associé à des octets différents, mais les deux fonctionnent aussi bien.

La conséquence c'est que le même octet peut selon les cas, représenter un

[4] Ceci explique pourquoi on regroupe les bits par 8,16 ou 32. Un bit à lui seul ne permet de représenter que des données très simples : le plus souvent on a besoin d'un nombre de combinaisons plus élevées. Par exemple les caractères alphanumériques qu'on a longtemps représenté sur un octet (256 caractères différents) et qu'on représente de plus en plus sur un mot de 16bits (Unicode).

nombre, un caractère alphabétique, un point d’une image, une instruction de programme, ou tout autre chose, et qu’il est tout à fait impossible en l’examinant à lui tout seul de savoir ce qu'il représente.

C’est le programme, donc le contexte qui le détermine. Voici à titre d’exemple un bloc de données brutes :



La première colonne est l’adresse relative des octets dans le bloc (autrement dit la position des octets, non pas dans la mémoire, mais comptée à partir du début du bloc. Notez que la numérotation commence à 0 et que ces numéros sont en hexadécimal : en troisième ligne 000010 = 16).

Les quatre colonnes suivantes sont le contenu de la mémoire, toujours en hexadécimal, les octets étant regroupés par deux pour plus de compacité. Chaque octet est représenté par deux chiffres exactement.

La dernière colonne est une <u>tentative</u> d’interpréter ces octets comme des caractères typographiques. C’est pratique, parce que cela permet de repérer du premier coup d'œil s'il existe des chaînes de caractères lisibles dans le bloc : ici “Yes” “No” “Cancel”. Mais c'est seulement une tentative : elle ne peut donner un résultat satisfaisant qu'à la condition que le bloc de données soit réellement censé représenter un texte. Sinon, on observe ce qui se produit ici : quelques mots identifiables peut-être, et des valeurs qui n'ont pas de sens comme texte.

Ce n'est pas pour autant une erreur, mais il nous manque simplement la convention pour décoder ce bloc.

(N.B. cette fenêtre a été obtenue en ouvrant une ressource avec ResEdit et en sélectionnant l’article de menu “Open using Hex editor” qui vous affiche les données brutes).

Si vous demandez à ResEdit d'ouvrir la ressource normalement, il vous dira que ce machin est en fait la description d’une boîte de dialogue, et, connaissant la convention, il vous affiche ça à la place :

Save changes to “0” before 1?

Top: 42 Height: 142
Left: 34 Width: 316

ResEdit reconnaît la convention de codage (format) des boîtes de dialogue, donc sait interpréter tel octet comme une dimension de la boîte, tel autre comme l’identificateur d’un bouton décrit ailleurs, tel groupe comme l'intitulé des boutons de cette boîte : ce sont les mots "Yes" "No" "Cancel" que nous avons reconnu dans le bloc. Avec de la patience et de l’obstination, vous pourriez même reconstituer cette convention (ou format) en essayant de voir à quoi correspond tel octet à telle position[5] .

Le micro processeur travaille sur les octets dont il connaît les valeurs, mais il ignore les conventions, donc ce qu'ils sont censés représenter.

Il s’ensuit donc que le µP peut parfaitement exécuter des opérations “absurdes” comme additionner les points d’une image. Il prendra les octets et les additionnera en les considérant comme des nombres et même si le résultat est un beau caca à l’écran, la machine elle-même ne peut détecter d’erreur, car du point de vue du micro processeur, il n'y en a pas. Il sait additionner des octets comme s'ils étaient des nombres, et c'est au programmeur de savoir si cette opération a un sens en fonction de la convention qu'il utilise dans ce bloc de données.

Représentation des nombres.

L’exemple le plus frappant est sans doute la représentation des nombres :

Les 256 états possibles d’un octet peuvent représenter les nombres de 0 à 255 (nombres sans signes) aussi bien que les nombres de -127 à +128 (nombres signés) et il existe d’ailleurs deux manières(complément à un et complément à deux)de représenter ces nombres signés. Il convient d'y ajouter la représentation BCD.

A chacune de ces conventions correspond un jeu d'instructions arithmétiques

[5] Je vous déconseille vivement de vous faire les dents sur cet exemple qui est assez compliqué.

appropriées qu'il faut utiliser à bon escient si on veut obtenir un résultat correct[6] .

Cela dit, on utilise très souvent l'une de ces conventions numériques pour représenter l'état des octets eux-mêmes, mais les octets ne "sont" pas des nombres, il est parfois important de s'en rappeler. La convention est simple, elle consiste à représenter les bits par des 0 ou des 1.

L'état d'un octet peut donc être représenté par huit chiffres ayant pour valeur 0 ou 1. Ces huit chiffres peuvent être considérés comme un nombre écrit en base 2. C'est ce qu'on appelle la représentation en binaire. Par exemple :

00000100 = 4 00000110 = 6 00001010 = 10, etc…

Pour des raisons de compacité et de lisibilité, on utilise plutôt la représentation hexadécimale (nombres en base seize). Un chiffre en base seize peut représenter quatre chiffres en base 2, donc quatre bits à la fois. Les lettres A B C D E F représentent les chiffres qui ne figurent pas dans notre système de numération, c'est à dire 10, 11, 12, 13, 14, 15. Ainsi les nombres qui nous ont servi d'exemple s'écrivent :

00000100 = 04 00000110 = 06 00001010 = 0A, etc…

Absence de limites et de bornes en mémoire.

Il s'ensuit également qu'il n'existe aucune manière de définir intrinsèquement la limite d'une zone de données, comme les étagères d'une bibliothèque déterminent des limites physiques que l'on ne peut dépasser. La seule chose que l'on puisse faire est d'affecter un octet à ce rôle, donc lui donner une valeur spéciale, mais qui ne sera spéciale que dans la convention utilisée. Dans une autre convention, cet octet pourra avoir une toute autre signification et ne plus jouer un rôle de borne !

Pour prendre un exemple : le stockage en mémoire de la chaîne de caractères "Hello".

Je vais naturellement inscrire dans des cases mémoire adjacentes, les codes ASCII des cinq caractères, mais il me faut un moyen de dire que la chaîne est constituée de cinq caractères, sinon, rien n'empêche le programme de prendre aussi les octets suivants, de les considérer comme des codes ASCII, et ceci jusqu'à la fin de la mémoire allouée à l'application[7] . Pour le micro processeur, rappelons le, il n'y a que des octets tous semblables, et non certains qui contiennent des caractères et d'autres non.

Il y a deux solutions classiques :

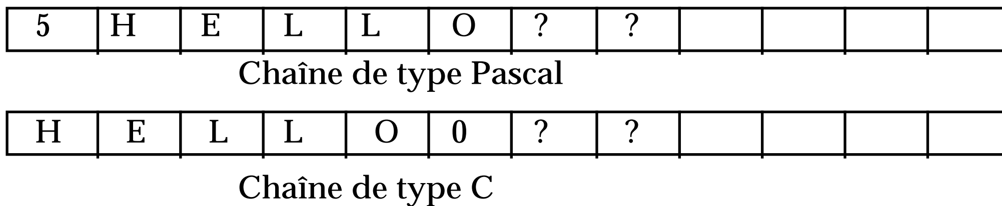
- soit placer au début de la chaîne (avant le caractère "H") le nombre de caractères dans le premier octet (chaînes Pascal).

---

[6] On trouvera en annexe des renseignements plus précis sur ces représentations.

[7] Ou jusqu'à ce que le programme se plante, c'est selon.

- soit placer après le dernier un code ASCII ne correspondant à aucun caractère utilisable, 0 par exemple (qui est un caractère de contrôle), c'est la chaîne de type C[8] .



5	H	E	L	L	O	?	?				

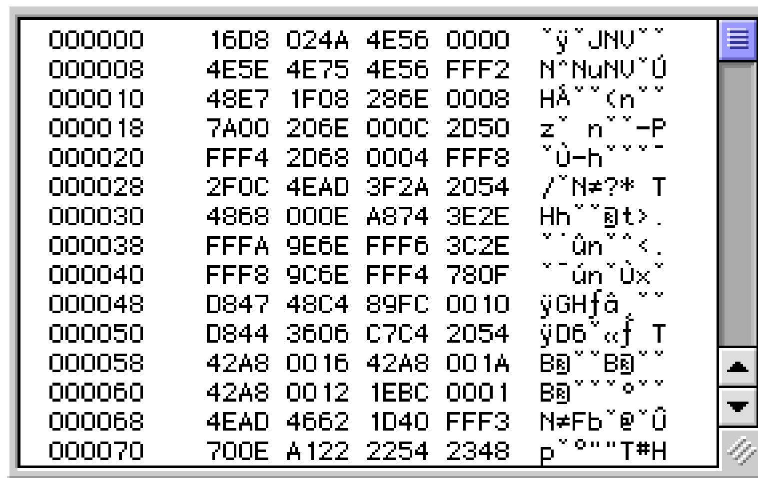
Chaîne de type Pascal

H	E	L	L	O	0	?	?				

Chaîne de type C

Représentation des programmes.

Ajoutons enfin que le programme lui même est stocké sous la même forme, les différents états des octets représentant alors des instructions en langage machine et leurs paramètres. Voici à quoi ressemble un segment de code en mémoire :



000000	16D8	024A	4E56	0000	˘ÿ˘JNV˘˘
000008	4E5E	4E75	4E56	FFF2	N^NuNV˘Û
000010	48E7	1F08	286E	0008	HÂ˘˘(n˘˘
000018	7A00	206E	000C	2D50	z˘ n˘˘-P
000020	FFF4	2D68	0004	FFF8	˘Ù-h˘˘˘¯
000028	2F0C	4EAD	3F2A	2054	/˘N≠?* T
000030	4868	000E	A874	3E2E	Hh˘˘®t>.
000038	FFFA	9E6E	FFF6	3C2E	˘˙ûn˘˜<.
000040	FFF8	9C6E	FFF4	780F	˘¯ún˘Ùx˘
000048	D847	48C4	89FC	0010	ÿGHƒâ¸˘˘
000050	D844	3606	C7C4	2054	ÿD6˘«ƒ T
000058	42A8	0016	42A8	001A	B®˘˘B®˘˘
000060	42A8	0012	1EBC	0001	B®˘˘˘º˘˘
000068	4EAD	4662	1D40	FFF3	N≠Fb˘@˘Û
000070	700E	A122	2254	2348	p˘°""T#H

Voici donc de quoi se nourrit votre processeur favori. [9] C'est ce qu'on appelle le langage machine, le seul qui soit compréhensible par le micro processeur. Là encore, on peut demander à ResEdit d'interpréter si on sait de quoi il s'agit, ce qui donne ceci :

[8] Les points d'interrogation dans le schéma représentent des octets dont l'état est inconnu et n'a pas d'importance.
[9] La tentative d'interpréter ceci comme des caractères alphanumériques échoue complètement bien entendu.

Offset	Addr	Opcode	Operand	Comment	Hex
+0000	00000C	LINK	A6,#$FFF2		4E56 FFF2
+0004	000010	MOVEM.L	D3-D7/A4,-(A7)		48E7 1F08
+0008	000014	MOVEA.L	$0008(A6),A4		286E 0008
+000C	000018	MOVEQ	#$00,D5		7A00
+000E	00001A	MOVEA.L	$000C(A6),A0		206E 000C
+0012	00001E	MOVE.L	(A0),-$000C(A6)		2D50 FFF4
+0016	000022	MOVE.L	$0004(A0),-$0008(A6)		2D68 0004 FFF8
+001C	000028	MOVE.L	A4,-(A7)		2F0C
+001E	00002A	JSR	"CODE,6"+$2B28		4EAD 3F2A
+0022	00002E	MOVEA.L	(A4),A0		2054
+0024	000030	PEA	$000E(A0)		4868 000E
+0028	000034	_GetPort		; A874	A874
+002A	000036	MOVE.W	-$0006(A6),D7		3E2E FFFA
+002E	00003A	SUB.W	-$000A(A6),D7		9E6E FFF6
+0032	00003E	MOVE.W	-$0008(A6),D6		3C2E FFF8
+0036	000042	SUB.W	-$000C(A6),D6		9C6E FFF4
+003A	000046	MOVEQ	#$0F,D4		780F
+003C	000048	ADD.W	D7,D4		D847
+003E	00004A	EXT.L	D4		48C4
+0040	00004C	DIVS.W	#$0010,D4		89FC 0010
+0044	000050	ADD.W	D4,D4		D844
+0046	000052	MOVE.W	D6,D3		3606
+0048	000054	MULS.W	D4,D3		C7C4

L'opération que vient de faire ResEdit s'appelle un désassemblage. C'est la réussite de l'opération qui montre qu'il s'agit de langage machine exécutable, et on peut dire qu'elle est réussie parce que la colonne opcode montre une suite régulière d'instructions. Le désassemblage de n'importe quelle zone de mémoire ne contenant pas de code est truffé de ??? qui signalent une opération inconnue du processeur.

Dans la colonne de droite, vous voyez les mêmes valeurs hexadécimales que dans la fenêtre des données brutes et vous pouvez ainsi apprendre que “48E7 1F08” signifie en fait : MOVEM.L D3-D7/A4,-(A7). C'est encore très ésotérique, mais tout le monde admettra que c'est un peu plus facile à lire qu'une suite de nombres hexa.

C'est de l'assembleur, autrement dit un décalque exact du langage machine sous une forme un peu plus adaptée aux êtres humains[10] .

L'important est ceci : quelque soit le langage que vous utilisiez, il est toujours traduit par un compilateur et/ou un interpréteur en langage machine, autrement dit cette suite d'octets totalement incompréhensibles sauf par le micro processeur.

Mais le processeur n'a aucun moyen de vérifier qu'il n'est pas en train d'essayer d'exécuter une zone contenant une image au lieu d'une suite d'instructions. La plupart du temps, il finit par détecter une erreur (un code ne représentant aucune instruction, ou une instruction munie de paramètres absurdes), mais il peut fort bien exécuter d'abord une série d'opérations que personne n'aura jamais prévu ni programmé.

Ceci peut naturellement donner des catastrophes : sur les anciennes machines, le processeur allait en général jusqu'au bout de la mémoire, là où se trouvaient les pilotes de périphériques, et on disposait d'une demie seconde environ pour ouvrir le lecteur de disquettes avant que le contenu de la disquette insérée ne soit brutalement

[10] Dans les temps héroïques, il y a eu des gens qui programmaient directement en langage machine en rentrant directement les valeurs binaires des octets avec de petites roues à molettes. Il fallait vraiment être motivé…

écrasé par un accès aléatoire et intempestif. Maintenant rassurez-vous, c'est du passé, les micro processeurs modernes disposent de contrôles intégrés pour ne pas sortir des limites.

On peut donc imaginer facilement que le premier souci d'un programmeur en assembleur est d'organiser soigneusement son espace mémoire de manière à éviter ces erreurs classiques et qui produisent de si beaux plantages :

- tenter d'exécuter des données comme si c'était un programme (erreurs de type 2 et 3 sur le Mac),
- effectuer des déplacements de blocs qui écrasent des données essentielles au déroulement du programme,
- ou encore tenter d'interpréter une image comme un texte (faire une recherche de chaîne avec les text items delimiters dans une image fonctionne parfaitement, mais ne donne aucun résultat vraiment intéressant).

Les langages dits évolués nous mettent en principe à l'abri de ce genre de mésaventures en prenant eux-mêmes en charge l'organisation des données et du programme. Ils le font de deux manières :

- en s'occupant eux-mêmes de localiser les données en mémoire (variables et structures) sans qu'elles se recouvrent ou se surchargent.
- En typant ces données.

Nous reviendrons sur le typage qui est essentiel, mais le principe est simple : en obligeant le programmeur à déclarer ce que représente telle ou telle variable (nombre, caractère alpha, etc…), les compilateurs et les interpréteurs sont à même de détecter si des opérations absurdes ou mal dimensionnées sont appliquées aux données. Ces déclarations permettent donc aux compilateurs de suppléer à l'aveuglement du micro processeur qui ne sait pas ce qu'il traite et d'anticiper un certain nombre d'erreurs à la conception même du programme.

Micro processeur.

Notions de base.

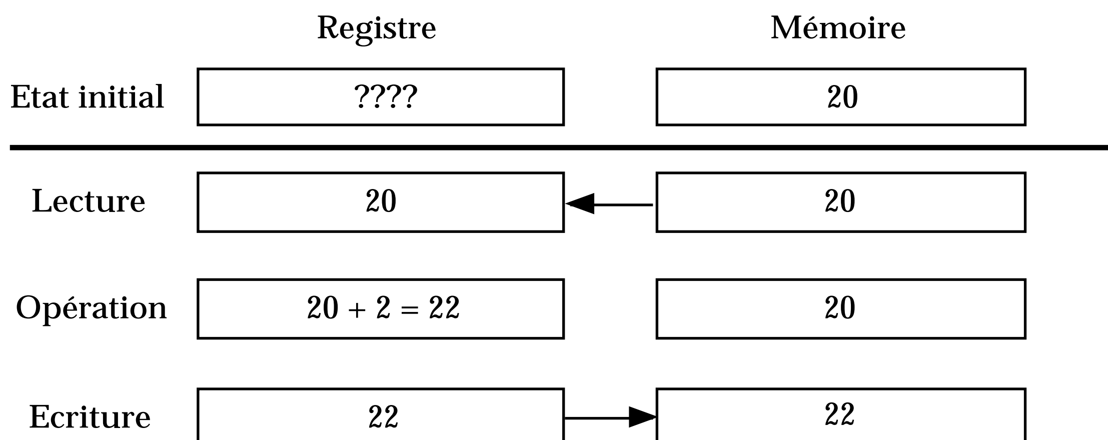
La petite bête qui tourne dans nos machines est fondamentalement composée de registres .

Un registre est exactement comme une cellule de mémoire constitué de bits (autrefois huit, maintenant 32, voire 64).

On peut donc copier l'état d'un registre dans une mémoire (écriture en RAM) ou au contraire copier l'état d'une mémoire (lecture en RAM ou ROM) dans un registre. En fin de compte, on se retrouvera avec un registre et une mémoire exactement dans le même état. Ce n'est pas un transfert comme si on déversait le contenu d'une casserole dans une autre : la source de la copie ne change pas.

Les micro processeurs ne peuvent pas travailler sur la mémoire autrement que par ces lectures/écritures[11] .

Donc si l'on veut modifier une donnée, il faut en général commencer par la charger dans un registre (lecture), la modifier par une opération (une addition par exemple), puis la stocker là où on l'a lue (écriture). Voici un schéma représentant une de ces opérations :



Les lignes Lecture, Opération et Ecriture correspondent chacune exactement à une instruction en langage machine. Elle sont nécessaires car le micro processeur ne peut exécuter l'addition directement dans la mémoire.

Les points d'interrogation montrent que l'état du registre est inconnu au départ. Ceci correspond au fait que l'état d'un registre ne se modifie pas spontanément, ou ne se "vide" pas au bout d'un certain temps. A la fin de cette opération, le registre contient 22 et le contenu restera 22 tant qu'on n'y touche pas.

En fait, le comportement est exactement le même que celui du presse-papier : la

[11] C'est de moins en moins vrai, mais je n'en tiens pas compte pour simplifier l'exposé.

lecture correspond à l'opération “copier” et l'écriture au “coller” : on peut coller autant de fois qu'on veut, le contenu du presse papier ne change pas tant qu'on n'y copie pas autre chose.

Les micro processeurs modernes contiennent plusieurs registres identiques de ce type qu'on appelle registres de données. (8 pour le 680X0 et aussi pour le PowerPC). Dans nos exemples ils seront notés D0, D1, … D7.

Ces trois opérations constituent un programme. Le µP a besoin de savoir où exactement lire et écrire les données en mémoire. Rappelons que chaque octet en mémoire possède un numéro d'ordre qui ne varie jamais : ce numéro est ce qu'on appelle une adresse et fonctionne à peu près comme l'adresse d'une maison dans une rue. Dans les ordinateurs, il n'y a qu'une seule rue, très longue et toutes les maisons sont rangées du même côté.

Le programme que nous avons vu se présente en assembleur de la façon suivante :

```
move memoire, D0   ; ceci est la lecture, la copie se fait de la gauche vers la droite
add   #2, D0       ; addition, le # signifie que 2 est un nombre et pas une adresse
move  D0, memoire  ; écriture
```

memoire représente ici l'adresse de la cellule mémoire à modifier. C'est donc un nombre que je suis censé connaître si j'ai au préalable organisé mes données. C'est ce qu'on appelle une adresse absolue, autrement dit le numéro exact de la cellule mémoire.

Ce mode de désignation de la cellule mémoire est suffisant pour modifier une donnée isolée, mais il n'est pas pratique lorsque l'on travaille avec des boucles.

Prenons l'exemple suivant :

j'ai une chaîne de caractères qui commence à l'adresse 00020 et je souhaite la modifier en ajoutant 32 à chacun de ces caractères[12] . Je ne peux pas simplement mettre ce programme dans une boucle :

```
faire autant de fois que de caractères

 move 00020, D0
 add   #32, D0
 move  D0, 00020

fin
```

car ceci ne donnerait pas le résultat cherché :

je modifierai un certain nombre de fois le contenu de la même cellule mémoire. Il faut donc qu'à chaque pas, l'adresse de cette mémoire change, donc l'incrémenter de 1.

Où se trouve cette adresse ? Dans le code en langage machine lui-même, juste après les deux octets qui contiennent le code de l'opération **move**. Je pourrais donc

---

[12] Cette opération n'est absurde qu'en apparence : ajouter 32 au code ascii d'une majuscule donne le code ascii de la minuscule correspondante. Si vous avez bien compris qu'il n'y a pas de lettres en mémoire, mais seulement des octets qui peuvent être interprétés indifféremment comme des caractères ou des nombres, ceci ne devrait pas vous poser de problème.

imaginer de procéder de la manière suivante en supposant que je connaisse l'adresse où est rangée cette valeur 00020.

Supposons qu'elle se trouve en 00400[13] :

```
faire autant de fois que de caractères

 move  00020, D0
 add   #32, D0
 move  D0, 00020
 move  00400, D0
 add   #1, D0
 move  D0, 00400

fin
```

Ce n'est pas pratique, et cela présente un inconvénient majeur : en fin de routine, 00400 ne contient plus l'adresse du début de la chaîne, mais celle de la fin.

Si je relance la routine, ça n'ira pas. En plus, c'est parfaitement incompréhensible.

Comme le problème se pose pour toutes les boucles, il existe naturellement un outil plus approprié, c'est le registre adresse.

Un registre adresse est identique à un registre de données, mais les opérations que l'on peut faire sur son contenu ne sont pas les mêmes.

Par contre, on peut dire au micro processeur que l'adresse d'une cellule mémoire se trouve dans un de ces registres d'adresse et non directement écrit dans le code. Il y a aussi plusieurs registres adresses dans les micro processeurs, ici nous parlerons des registres A0 à A7.

On procédera donc de la façon suivante :

```
faire autant de fois que de caractères

     move  #00020, A0  ; je charge dans A0 l'adresse de ma cellule mémoire
     move  (A0), D0    ; lecture, j'utilise le contenu de A0 comme adresse
     ; s'il n'y avait pas les parenthèses, je copierai le contenu du registre A0
     ; lui-même et non la cellule mémoire dont il contient l'adresse.
     add   #32, D0
     move  D0, (A0)+   ; notez le '+' ! c'est l'incrémentation automatique
fin
```

Revenons en détail là dessus car ce mécanisme est fondamental, y compris dans les langages évolués où il sous-tend la notion de pointeur et de référence.

Dans les schémas suivant, nous essayons de montrer les différentes possibilités que possède un micro processeur.

Le machin appelé sélecteur d'adresses n'existe pas en réalité, en tous cas, il n'est pas accessible au programmeur. C'est une manière de montrer où le micro

---

[13] Il est tout à fait possible d'écrire en assembleur un programme qui se modifie lui-même (après tout ce n'est encore qu'une série d'octets que l'on peut modifier). Si cette tentation surgissait, résistez courageusement et intéressez-vous à Lisp qui est fait pour ça. Vous n'y gagneriez que d'innombrables nuits sans sommeil et une réputation de programmeur plus que douteuse.

processeur va chercher l'adresse de la valeur qu'il est censé copier.

Les flèches rouges montrent le fonctionnement de cette sélection d'adresse et les bleues le fonctionnement de la copie proprement dite.

La partie à droite est une représentation de la mémoire. Il y a une zone de données et une zone où se trouve le programme proprement dit.

Codeop représente le code de l'instruction étudiée, il est différent pour chaque opération, mais sa valeur précise n'a pas d'intérêt.

Instruction : move #00020, A0

D0

A0 00020

Sélecteur d' adresses

H 00020
E 00021
L 00022
L 00023
O 00024
0 00025

Données

Programme

codeop
00020 00400

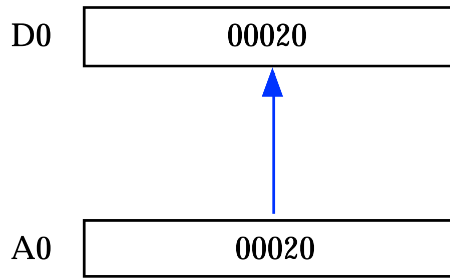
Processeur

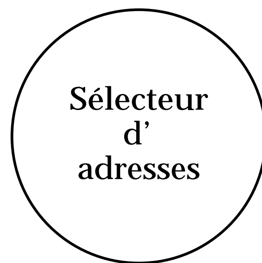
Mémoire

Notez le # dans l'instruction. Ceci indique qu'il faut considérer 00020 comme la donnée à charger, sinon, cette valeur est considérée comme l'adresse absolue de la donnée à charger.

Voyons maintenant l'usage du registre adresse A0 : le premier exemple est un simple transfert entre registres, le sélecteur d'adresses n'a pas besoin de renseignement supplémentaire.

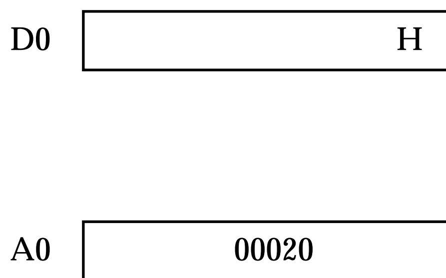
Instruction: move A0, D0

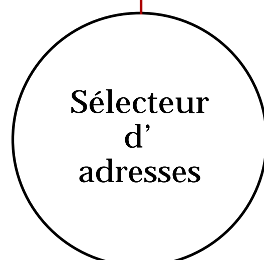




Le suivant est plus intéressant, notez les parenthèses autour de A0.

Instruction: move (A0), D0





Et enfin le dernier qui montre le dispositif de post incrémentation. Le registre A0 est incrémenté après l’écriture en mémoire :

Instruction: move D0, (A0)+

D0 h

A0 00021

Sélecteur d’ adresses

Données

h 00020
E 00021
L 00022
L 00023
O 00024
0 00025

Programme

codeop 00400

Processeur

Mémoire

On voit que le registre A0 est ainsi “prêt” pour le prochain pas de la boucle.

Post incrémentation suppose que l’on peut aussi pré incrémenter, et c’est exact. On verra dans la suite qu’il existe d’autres modes d’adressage, c’est à dire d’autres manières de donner l’adresse d’une donnée à lire ou écrire en mémoire.

L’important est de bien comprendre qu’il existe deux mécanismes :

- le premier est de donner l’adresse comme une constante,

- le deuxième est de donner un emplacement (registre ou adresse) où l’on pourra trouver l’adresse de la donnée.

Voici un autre exemple qui traite de données composites.

Tant qu’il s’agit de données simples qui tiennent sur 4 octets, le problème ne se pose pas car les registres de données peuvent les contenir toutes entières.

Mais supposons que nous traitions une liste de personnes avec pour renseignements un n° d’ordre et un âge, tous les deux stockés sur un octet. Nous avons construit cette liste sous forme d’éléments contigus, simplement rangés les

uns derrière les autres (donc des paquets de deux octets qui se suivent).

?	0	23	1	54	2	12	?								

1er enreg. 2é enreg. 3è enreg.

adresse de base de la liste

Pour faire une boucle qui va explorer l'âge des personnes, nous avons besoin de connaître l'adresse du début de la liste qui est aussi celle du premier enregistrement :

1- Charger adresse du début de liste en A0

Ceci fait, il existe une forme spéciale d'instruction permettant d'atteindre, non pas l'octet pointé par A0, mais un octet qui se trouve un peu plus loin. Ainsi nous savons que l'âge n'est pas le premier octet de l'enregistrement, mais le second. Pour l'atteindre, on écrira donc :

```
move   1(A0), D0
```

Le schéma illustre le fonctionnement de l'instruction :

Instruction: move 1(A0), D0

Données

0	00020
23	00021
1	00022
54	00023
2	00024
12	00025

D0 23

A0 00020 +1

Sélecteur d' adresses

codeop Programme 00400

Processeur Mémoire

La donnée effective est en 00021, mais le registre adresse contient celle du début de l'enregistrement. On voit comment on peut utiliser ceci pour construire des

structures de données complexes et y accéder facilement[14] .

On pourrait modifier le registre A0, mais si un accès est nécessaire ultérieurement sur une autre donnée, il faudra tenir compte du fait qu'il ne contient plus l'adresse du début.

Imaginez par exemple que l'on accède à l'âge seulement quand une condition est remplie, puis au numéro :

```
test si l'âge du capitaine
  ajouter 1 au registre A0
  bla bla bla
fin de test
  utiliser A0 pour obtenir le numéro ; on voit qu'ici selon que le test est vrai ou
  ; non, A0 va contenir une valeur différente !
  ; Si j'utilise l'adressage avec déplacement, le problème ne se pose pas.
```

Pour passer à l'enregistrement suivant :

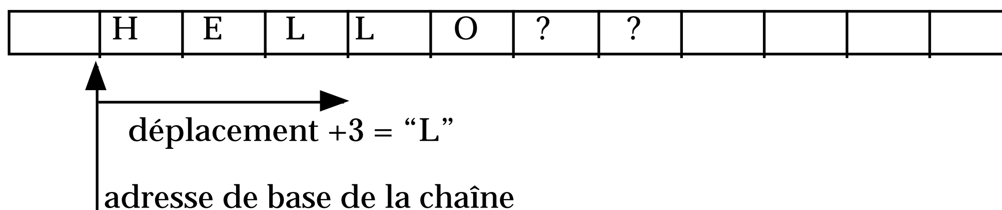
```
adda        #2, A0
```

Pourquoi 2 ? parce que c'est la longueur de l'enregistrement.

Si notre affaire contenait davantage de données, nous ajouterions ici la longueur de l'enregistrement. Et pour accéder à chaque donnée individuelle, nous ferions un accès relatif au début de l'enregistrement (contenu dans le registre adresse A0 qui sert de base) en spécifiant un déplacement égal à la position de la donnée (comptée en octets) dans l'enregistrement[15] .

Cas classique d'une chaîne de caractères :

on voit que le µP n'a pas besoin d'une boucle pour trouver le xième caractère d'une chaîne : il récupère l'adresse de début de la chaîne, ajoute x au registre adresse (ou deux fois x, si c'est de l'Unicode), et hop ! C'est donc un accès ultra rapide, bien plus rapide que de passer une chaîne en revue en y cherchant un délimiteur.



Le µP possède de nombreux modes d'adressage plus ou moins complexes, où l'adresse finale est le résultat d'un calcul effectué au vol dans des registres internes. A partir d'une adresse dite de base (qui peut être une constante ou le contenu d'un

[14] C'est probablement comme cela que sont construits les records en AppleScript. En C et langages dérivés, on a les struct.

[15] En anglais, déplacement se dit offset. Ceux qui connaissent l'instruction correspondante en AppleScript verront tout de suite la correspondance: c'est effectivement ce mécanisme base + déplacement qui est mis en œuvre. On voit aussi qu'il a beaucoup de chances d'être très efficace car c'est une instruction du processeur.

registre adresse), on ajoute différents déplacements qui peuvent être des constantes ou le contenu d'autres registres.

Nous ne décrirons pas ces modes d'adressage possibles, car il n'y a aucun moyen de persuader un compilateur d'utiliser l'un ou l'autre.

En conclusion.

L'important est de constater la chose suivante :
le µP est extrêmement efficace et rapide sur des données en format fixe, autrement dit, tant que les données sont rangées dans des blocs de longueur fixe, connue à l'avance, et immuable.

Il est nettement moins à l'aise sur des formats où la longueur des données est variable : c'est par exemple le cas des chaînes de caractères.
Le nom d'une personne est de longueur variable. S'il entre dans une donnée composite, on peut avoir intérêt à réserver une zone de longueur fixe (mettons 60 caractères) suffisamment grande pour contenir tous les noms imaginables de manière à garder le bénéfice d'un format fixe (il suffit d'utiliser un déplacement de 60 octets pour "sauter" le nom et atteindre la donnée suivante, quelque soit la longueur réelle de ce nom). Il y a naturellement de la place perdue, mais les performances sont nettement meilleures.
C'est une situation très générale : les structures de données les plus compactes sont en général les moins efficaces, et les plus efficaces le sont au prix d'un certain gaspillage de mémoire.

Jeu d'instructions du micro processeur.

En dehors de ces instructions de lecture/écriture en mémoire qui sont fondamentales, les micro processeurs savent aussi faire des opérations sur les valeurs contenues dans leurs registres :

- Opérations arithmétiques : additions soustractions, multiplications et divisions, changements de signes etc…[16.]
- Opérations logiques au niveau bit : ET, OU inclusif et exclusif, négation, etc…
- Instructions de contrôle d'exécution et de test. Donc les boucles et les si. Les processeurs ne sont pas très riches dans ce domaine, mais ils possèdent les éléments de ces instructions indispensables. Comme cela fonctionne à peu près comme dans les langages évolués, nous ne nous y attarderons pas. Il n'y a pas d'instruction "au cas où".
- Des tas de machins super utiles au programmeur en assembleur et qui ne concernent vraiment que lui. D'autres registres aussi dont je refuse

---

[16] Signalons que les opérations en virgule flottante sont l'apanage d'un élément particulier du micro processeur doté de registres spéciaux et d'instructions particulières. Autrefois, cet ensemble était dans un circuit séparé appelé coprocesseur.

énergiquement de parler ici.

Disons un mot de la manière dont fonctionnent les instructions de contrôle d'exécution. La compréhension du mécanisme n'est pas essentielle, mais les curieux impénitents y trouveront sans doute de quoi les mettre en appétit.

De la même manière que les adresses sont utilisées pour l'organisation des données, ces adresses sont utilisées dans l'exécution d'un programme. Il existe un registre spécial appelé pointeur de programme qui contient l'adresse de la prochaine instruction à exécuter. Si celle-ci est une simple opération, on passe simplement à la case mémoire suivante, mais il arrive (en cas de boucle ou de test) que l'on revienne en arrière ou que l'on saute plus loin. Ces instructions de saut contiennent l'adresse de la prochaine instruction à exécuter.

exemple d'instruction:

```
 beq   1295        ; beq est un si: si la condition est remplie on "saute" à
                   ; l'adresse 1295
 move  A1,D1       ; sinon on exécute cette instruction placée en séquence
 etc…

------- ensuite, à l'adresse 1295
1295   move  A2,D1 ; après le saut l'exécution reprend ici.
 etc…
```

Sous-programmes.

Les appels de <u>sous-programme</u> méritent une mention particulière : ils contiennent une adresse (celle du sous-programme) et le µP est capable de stocker quelque part l'adresse à partir de laquelle le sous-programme a été appelé (pour pouvoir y revenir à la fin du sous-programme). Il le fait dans une structure de donnée particulière appelée la pile, située elle aussi en mémoire que nous allons décrire en détail.

Procédures et fonctions.

Avant cela précisons quelques points de vocabulaire. Le mécanisme d'appel de sous-programme est unique pour le micro processeur à quelques détails près. Or ceci correspond dans les langages évolués à différentes notions : routine, procédure, fonction, handler (ou gestionnaire), etc...

Ces distinctions sont introduites par les langages évolués. Routine n'a pas de sens précis : c'est simplement un bout de code qui se suffit à peu près à lui-même et qu'on peut appeler.

Fonction et procédure sont deux variantes : la fonction se distingue de la procédure par le fait qu'on récupère un résultat : le nom de la fonction peut donc être utilisé en lieu et place de la valeur qu'elle fournit. Par exemple :

```
set y to my carre(x)
```

```
on carre(z)
  return z*z
end carre
```

L'appel de la fonction carre() renvoie un résultat qui sera stocké dans la variable y. Les procédures ne renvoient aucun résultat. Le type d'appel est donc plutôt :

```
my connectToInternet()
```

La distinction est tellement ténue que la plupart des langages de programmation ne la connaissent pas. Ils ont des fonctions, et si l'on a besoin d'une procédure, on programme une fonction qui ne retourne aucun résultat.

Procédures et fonctions appartiennent normalement à la même application. Lorsqu'il s'agit d'appels d'une application à une autre, le mécanisme n'est plus forcément utilisable, en raison des dispositifs de protection de la mémoire. Pour les appels au système, il existe un mécanisme spécial dont nous ne parlerons pas car il est systématiquement encapsulé de façon transparente dans les langages évolués.

Apple events.

Par contre, AppleScript dispose d'un système d'appel qui lui est propre et qui consiste à générer des Apple events. Un Apple event est un événement pour le système (en particulier pour la partie du système qui les traite : l'event Manager). Ces événements sont l'enregistrement de choses comme les clics souris, les touches enfoncées au clavier, etc…

Le système s'occupe en permanence de collecter les événements qui sont pour la plupart des signaux en provenance de l'utilisateur. Quand il en détecte un, il stocke une description précise de l'événement (date en clics, nature : clic souris, touche clavier, touches de modification enfoncées, localisation du curseur écran, etc…)

Ceci fait, il stocke ces renseignements dans une file d'attente par ordre chronologique..

Handlers ou gestionnaires.

Les applications en fonctionnement interrogent l'Event Manager pour obtenir de lui les événements qui les concernent, et c'est ainsi qu'elles interagissent avec l'usager. Une application pilotée par événements (la quasi totalité des applis Mac) est ainsi constituée d'une série de routines destinées à répondre à chaque événement qui peut survenir. Ces routines sont ce qu'on appelle des handlers. Elles sont destinées à être lancées en réponse à un événement extérieur.

L'une des caractéristiques de cette file d'attente, c'est que le programmeur peut y ajouter des événements "fictifs", c'est à dire qui n'auront pas été réellement produits par l'usager. A quoi ça sert ? Les programmes du type QuickKeys ou AppleWorks sont capables d'enregistrer des séquences d'événements utilisateurs (macros), et c'est ainsi qu'ils les reproduisent à la demande : en expédiant à l'Event

Manager une copie des événements originaux. Les applications ne voient aucune différence avec une interaction réelle.

L'Apple event est une classe particulière d'événements qui permet d'utiliser le mécanisme pour communiquer entre applications.

Comme il ne s'agit pas d'interaction avec l'usager, les handlers ne sont pas les mêmes. Les handlers prévus pour répondre aux Apple events se trouvent dans le dictionnaire des applis (au sens AppleScript du terme), et il en existe une "suite obligatoire" que toutes les applis sont tenues de mettre en œuvre. Les scripts AppleScript possèdent eux aussi des handlers qui peuvent non seulement être appelés à l'intérieur du script lui même (ils se comportent alors comme une fonction), ou depuis un autre script ou même une autre application si elle sait générer des Apple events. Dans ce cas, le dictionnaire de votre script va être l'ensemble des handlers qu'il contient : ils seront chargés de répondre aux Apple events qui peuvent être adressés à votre script. C'est pour cela qu'ils sont appelés handlers, même s'ils ressemblent énormément à des fonctions.

Enfin, voici la raison de l'instruction "tell". Tell désigne la cible par défaut. Qui peut être n'importe quel objet d'un script.. À la différence des événements utilisateurs normaux que le système sait comment diriger sur la bonne appli (celle qui possède la fenêtre active, ou alors le système quand on clique en dehors, pour changer d'appli de premier plan), le système ne connaît pas a priori l'application destinataire d'un Apple event : il faut le lui préciser, d'où l'instruction "tell".

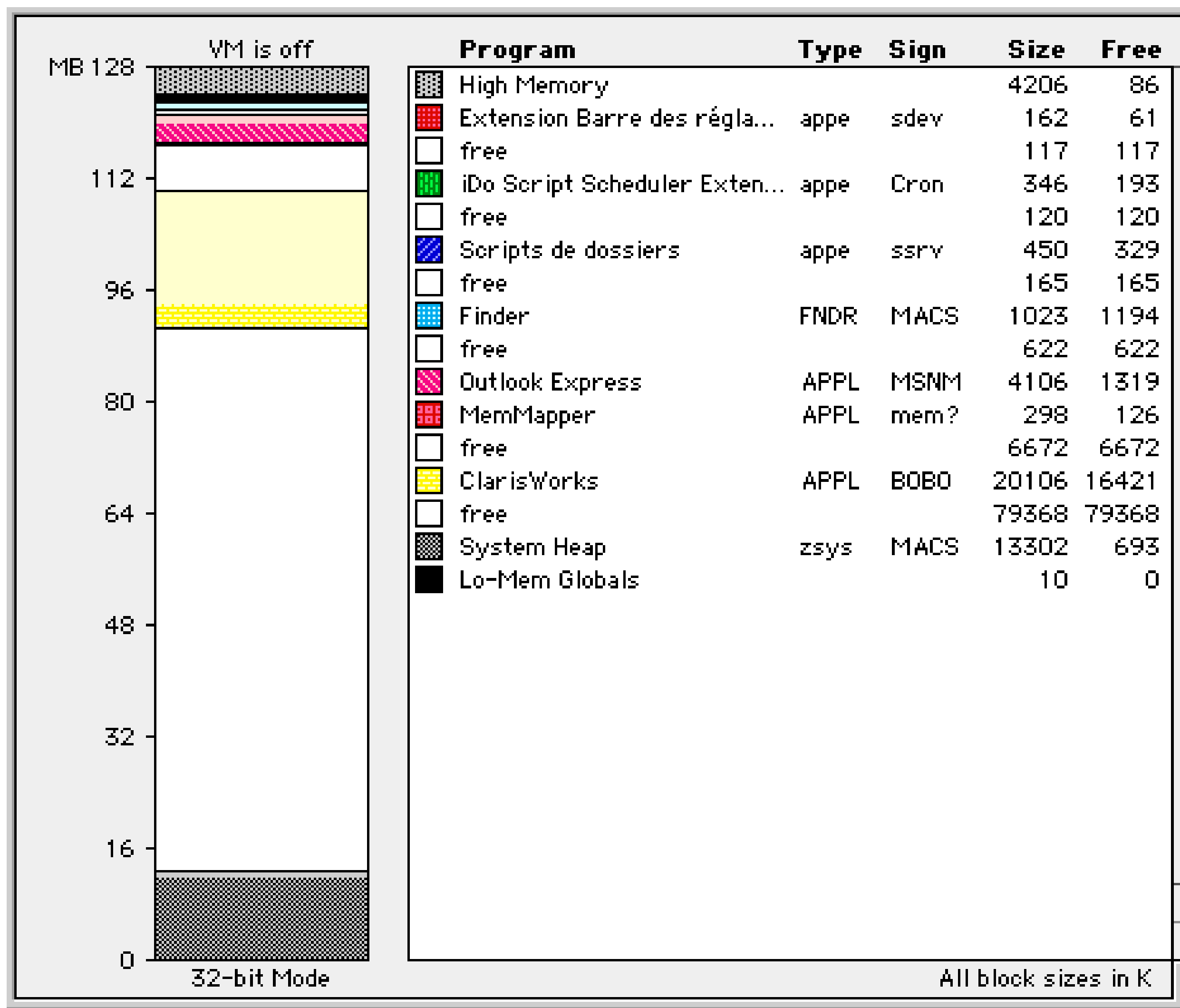
Organisation de la mémoire.

Mémoire globale.

Pour information, voici comment se présente l'installation d'un programme en RAM lorsque le système le charge et le lance. L'illustration suivante donne une carte d'utilisation de la RAM un peu plus détaillée que "à propos de votre ordinateur…" puisqu'elle ne donne pas seulement la liste des applications lancées, mais également celle de certaines extensions et tableaux de bord.

Dans tous les cas, le système tente de réserver pour l'application ou l'extension la quantité de mémoire qu'elle a demandé (la valeur préférée dans la fenêtre "Infos" pour les applications). A l'application ensuite de se débrouiller pour y stocker aussi ses données et documents [17] .

[17] Le "nouveau gestionnaire Macintosh" permet aux applications qui savent s'en servir de dépasser cette limitation.



Program	Type	Sign	Size	Free
High Memory			4206	86
Extension Barre des régla…	appe	sdev	162	61
free			117	117
iDo Script Scheduler Exten…	appe	Cron	346	193
free			120	120
Scripts de dossiers	appe	ssrv	450	329
free			165	165
Finder	FNDR	MACS	1023	1194
free			622	622
Outlook Express	APPL	MSNM	4106	1319
MemMapper	APPL	mem?	298	126
free			6672	6672
ClarisWorks	APPL	BOBO	20106	16421
free			79368	79368
System Heap	zsys	MACS	13302	693
Lo-Mem Globals			10	0

Le système lui même est organisé de la même manière qu'une application : il possède sa zone de variables globales (Lo-Mem Globals), son tas (System Heap) et sa pile (High Memory).

Ensuite on peut voir la proportion de mémoire réellement utilisée par une application : par exemple Claris Works dispose de 20Mo et il lui reste 16 Mo de libre (colonnes size et free).

On peut aussi constater que la mémoire est actuellement fragmentée, c'est à dire que la mémoire libre est répartie en un certain nombre de blocs de différentes tailles (marqués free). Si l'on cherche à lancer une nouvelle appli, la mémoire disponible est en fait la taille du plus grand bloc disponible (ici 79 Mo). Le système est incapable de déplacer Claris Works pour y adjoindre le petit bloc de 6 Mo qui se trouve au dessus.

VM off signifie que la mémoire virtuelle est désactivée et 32 bits mode que le mode d'adressage sur 32 bits est actif.

Le mode 32 bits est un souvenir du passé. Les premiers processeurs 68000 sur les premiers Macs avaient déjà des registres adresses de 32 bits, donc capables en principe de gérer jusqu'à 4 Go de RAM. Cette quantité paraissait si phénoménale à l'époque[18] que Motorola n'avait même pas prévu de broches externes pour les trois bits de poids fort du bus, réduisant ainsi la capacité d'adressage physique réelle à 500 Mo (déjà pas mal, alors que l'immense majorité des machines disposait de

[18] Le concurrent 8086 chez Intel n'avait que 1 Mo de capacité d'adressage.

moins de 10 Mo).

Ceci revient à dire que les trois derniers bits d'une adresse étaient en fait ignorés : on pouvait y mettre n'importe quelle valeur, le processeur sélectionnait toujours la même case mémoire.

Ces trois bits ignorés par le processeur dans les adresses ont été utilisés par des programmeurs pour y stocker des informations diverses. Par exemple, le gestionnaire de mémoire des premiers systèmes s'en servait pour marquer le verrouillage des blocs mémoire.

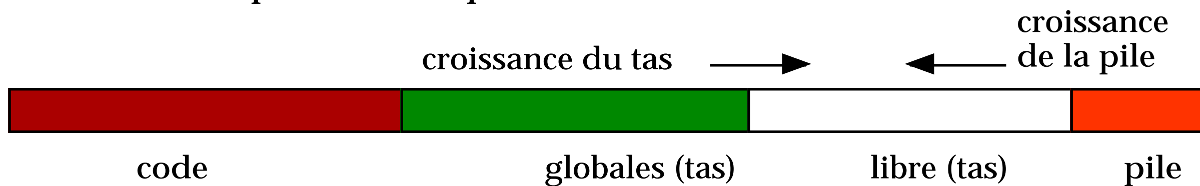
Par la suite, ces bits ont été câblés, si bien que ces adresses qui autrefois désignaient une seule et même case mémoire quelque soit la valeur de leurs bits de poids fort, ont commencé à désigner des adresses physiquement différentes (qui en général étaient situées au delà de la mémoire physique réellement disponible). D'où des erreurs : certaines applications anciennes sont devenues incapables de fonctionner correctement.

Mémoire allouée à l'application.

Le bloc mémoire attribué par le système va être coupé en trois morceaux qui contiendront respectivement : le code exécutable, les données globales et dynamiques et la pile[19] .

Le code exécutable est le programme lui-même, en langage machine. Les données globales sont les variables globales du programme. S'y ajoutent les données qui sont créées au vol (dynamiquement) en fonction des besoins de l'utilisateur[20] .

Cette zone est souvent appelée tas (anglais : heap). Elle suit le code objet. La pile par contre est implémentée à partir de la fin du bloc.



On voit donc clairement que les limites du tas et de la pile ont tendance à se rapprocher au fur et à mesure de l'utilisation du programme[21] . Qu'est-ce qui les empêche de se recouvrir et de se surcharger ? Et bien rien, ou pratiquement rien, sinon la prévoyance du programmeur[22] .

Parfois, le système "s'aperçoit" que la pile a débordé et probablement écrasé

[19] Bien sûr c'est souvent plus compliqué, mais en gros c'est ça et ce quelque soit le système d'exploitation.

[20] Les globales d'une application seront par exemple la définition des menus qui est toujours présente. Quand l'utilisateur crée ou charge un document, les données correspondantes sont installées ou créées au vol dans le tas (et éventuellement détruites ensuite): c'est ce qu'on appelle une allocation dynamique.

[21] Et la loi de Murphy (ou *in french* de l'emmerdement maximum) permet d'affirmer que cette tendance a de très fortes chances de s'actualiser.

[22] Et on sait ce que ça vaut.

des données essentielles. Il arrête alors le programme avec un message (anglais : stack overflow). Parfois, il ne se rend compte de rien[23] et le programme plante avec une erreur de type 1,2 ou 3 (sur Mac).

La première signale que le µP tente d'accèder à une zone mémoire qui n'existe pas (le numéro est correct, mais vous n'avez pas assez de RAM),

la deuxième signale que le µP tente d'utiliser une adresse impaire (ce qu'il n'a en général pas le droit de faire pour des raisons hardware relevant de l'économie sordide et plutôt complexes à expliquer).

et la dernière que le programme contient une instruction inconnue ou illégale.

Pas de panique. Il est absolument impossible de provoquer l'une de ces trois erreurs directement à partir d'un langage évolué, donc ce n'est pas à proprement parler une erreur de programmation.

Mais l'une de ces erreurs apparaît en général très vite lorsque le pointeur de programme du µP est chargé avec une adresse aléatoire.
En C, on peut à la rigueur bricoler ce genre de situation.
En AppleScript, c'est impossible, mais il suffit que tas et pile se recouvrent : dans ce cas, modifier le contenu d'une variable peut remplacer une adresse de retour valide par n'importe quoi. La solution serait d'augmenter la mémoire allouée à l'application pour mettre le château de sable (les globales) à l'abri des vagues insidieuses (la pile). Malheureusement, en AppleScript, la taille de la pile ne peut être modifiée de cette façon : quelque soit la mémoire allouée à l'application, la taille de la pile est toujours la même.

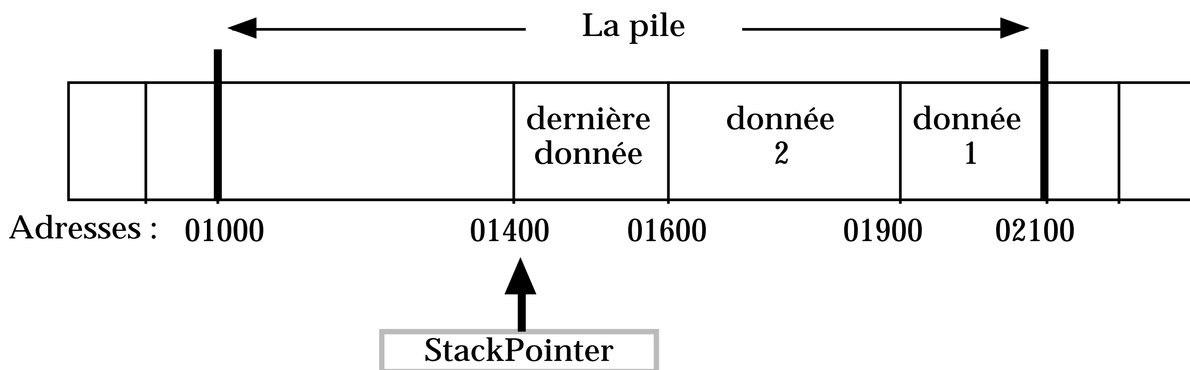
[23] Parce que la pile déborde en douce quand il ne regarde pas et revient sagement à une taille plus raisonnable après avoir tout ravagé. C'est un peu comme les vagues qui menacent en permanence un château de sable.

La pile

Définition.

Les piles (anglais : stack) sont des structures de données générales aussi appelées LIFO pour “last in, first out” autrement dit “dernier élément entré, premier sorti”. Elles ressemblent à des piles d’assiettes d’où leur nom[24] .

Elles sont associées à deux instructions fondamentales : empiler (une donnée), et dépiler (la donnée placée au sommet de la pile).
Elles sont implémentées sous la forme d’un bloc de mémoire que l’on remplit à partir deson extrémité finale, à reculons, c'est à dire en progressant vers les adresses les moins élevées (en revenant vers les premiers numéros de la rue).

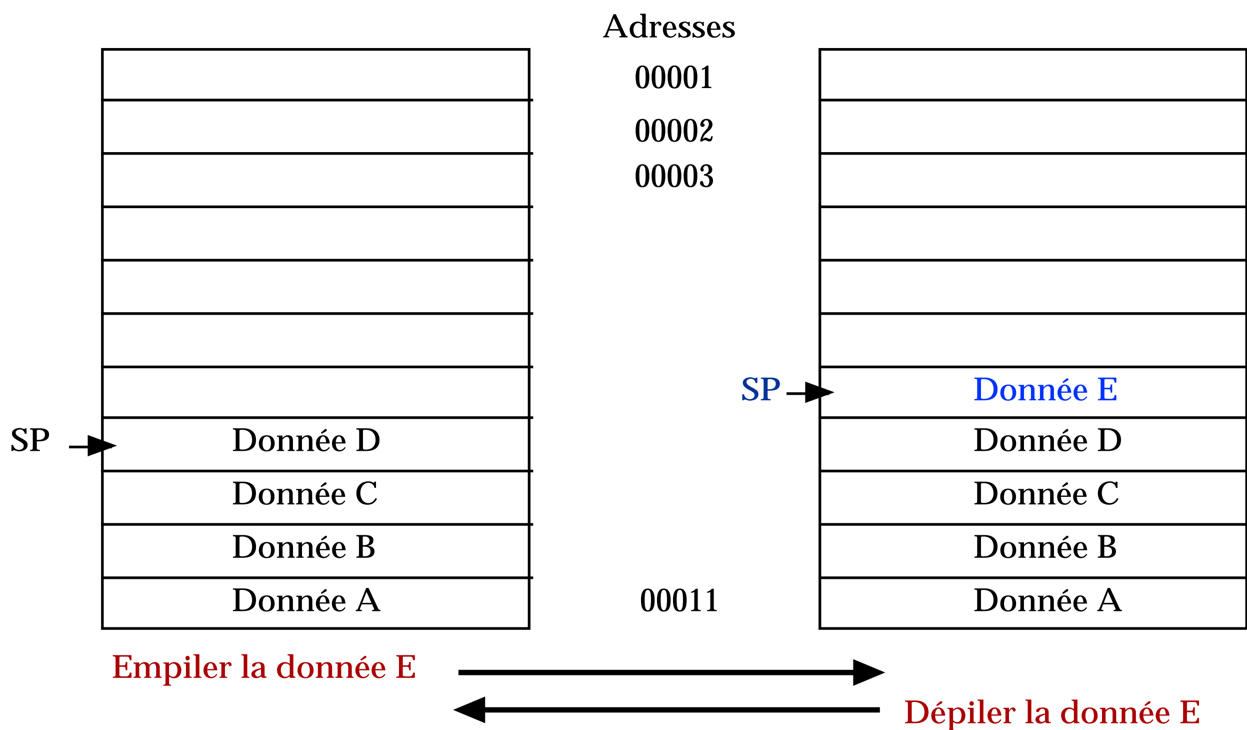


Un pointeur (appelé pointeur de pile, SP en abrégé), permet de connaître l’adresse de la dernière donnée entrée dans la pile.

Dépiler consiste donc à récupérer la donnée pointée par SP, puis à incrémenter SP de la taille de cette donnée.

Empiler consiste à décrémenter SP de la taille de la donnée à ranger (pour faire de la place, sinon on écraserait la donnée précédente), puis de stocker la donnée à l’adresse pointée par SP.

[24] Sauf si vous êtes légèrement tordu, vous prenez normalement l’assiette placée au dessus de la pile et c’est aussi là que vous rangez les assiettes propres.



La pile du micro processeur.

Tous les µP disposent d'un registre adresse spécial appelé pointeur de pile qui pointe vers une zone mémoire appelée *la* pile (THE pile) pour la distinguer des piles supplémentaires que vous auriez envie d'implémenter.

Cette pile est utilisée par le µP pour le stockage des adresses de retour de sous-programmes. En effet, lorsqu'on appelle une fonction,

my fonction()
instruction suivante

tout se passe comme si le corps de la fonction était inséré entre l'appel et l'instruction suivante du programme (notée en rouge). Le compilateur n'insère pas réellement le corps de la fonction à cet endroit car cela ne serait pas économique. Au lieu de cela, il indique à quelle adresse va se trouver la fonction à exécuter (ailleurs dans le programme).

Lorsque l'exécution parvient à la fin du corps de la fonction, elle doit reprendre ou revenir à instruction suivante. Cette adresse ne peut être donnée une fois pour toute dans le code, puisqu'une fonction peut être appelée plusieurs fois dans le programme à des endroits différents.
C'est la raison pour laquelle au moment de l'appel, le µP enregistre dans la pile l'adresse de instruction suivante. Arrivé à la fin du corps de la fonction, il récupère cette adresse dans la pile pour savoir où reprendre l'exécution.

L'instruction JSR adr. a pour effet d'empiler l'adresse de l'instruction suivant cette instruction d'appel, puis d'effectuer un saut à adr. en transférant la valeur adr. dans le pointeur de programme (registre PC). adr désigne donc l'emplacement

où commence la fonction appelée.

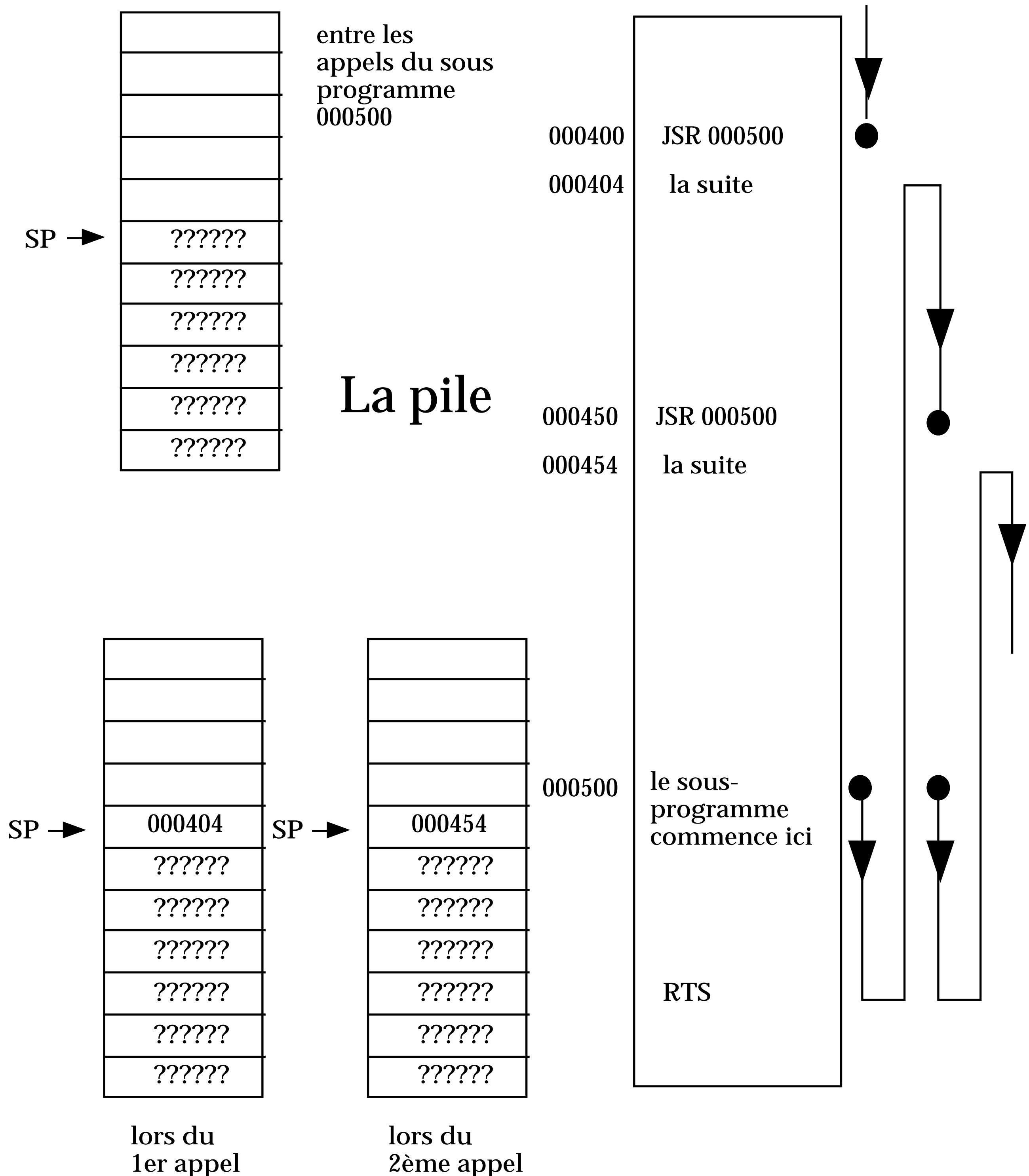
L'instruction RTS = retour marque la fin de la fonction. Elle est l'exact équivalent de l'instruction "return" en AppleScript[25] . Elle n’a besoin d’aucun paramètre : c'est un dépiler dans PC. On récupère ainsi l'adresse empilée au moment de l'appel et on la transfère dans le pointeur de programme qui, rappelons-le, désigne la prochaine instruction à exécuter.

Le schéma suivant montre l'exécution d'un programme qui fait deux appels successifs à la même fonction ou sous-programme, située à l'adresse 000500. Les flèches sur la droite montrent dans quel ordre l'exécution à lieu. Les gros points montrant à quel endroit il y a saut à l'adresse spécifiée par JSR. (donc l'adresse de la fonction).

A gauche on a représenté l'état de la pile en dehors du sous-programme et lors des deux appels successifs, juste avant le RTS. On voit que le pointeur de pile désigne l'adresse de retour, c'est à dire l'adresse qui suit immédiatement le JSR ayant effectué l'appel.

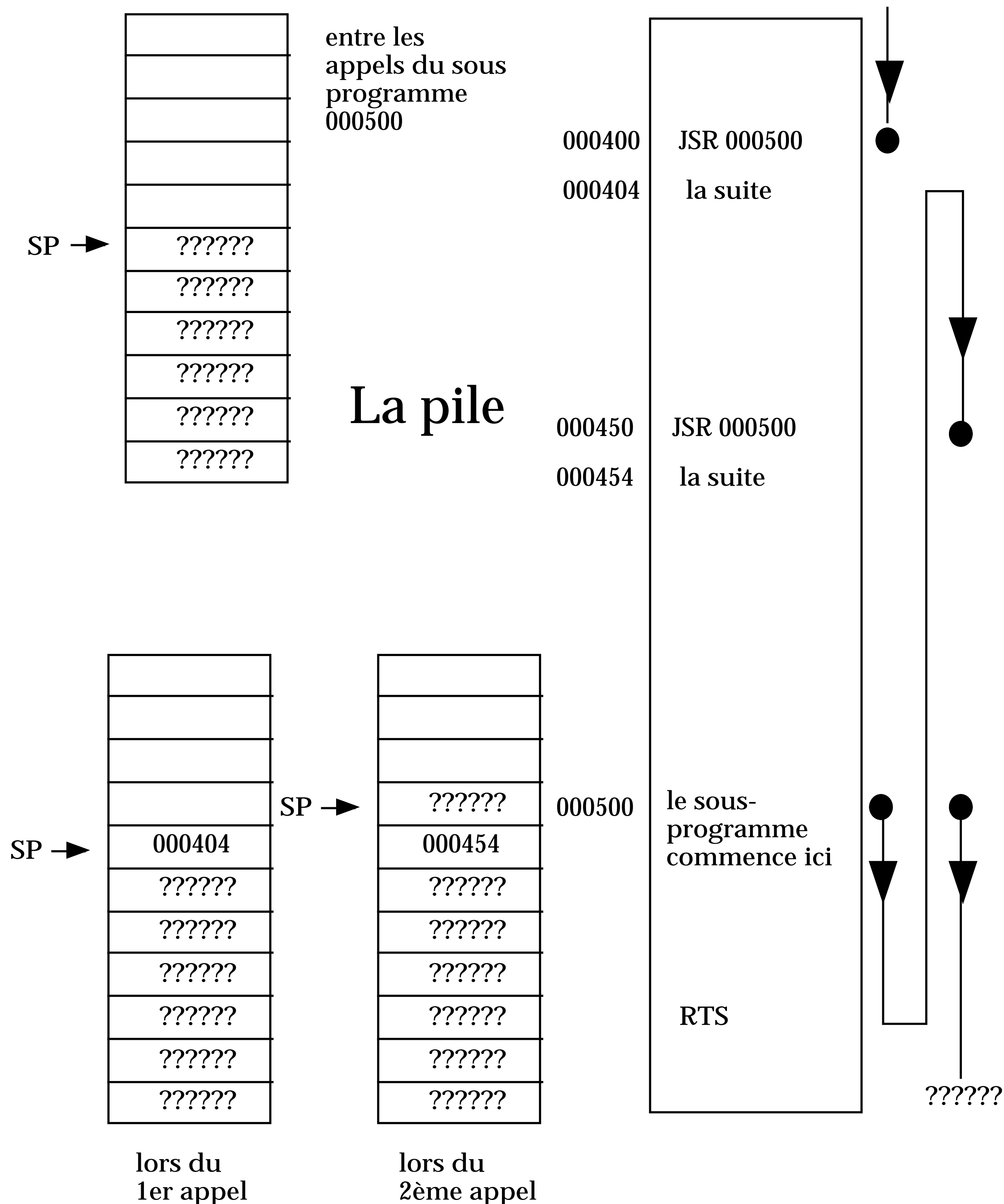
Cette adresse est naturellement différente à chaque fois.

[25] Hormis le fait qu'elle n'est pas obligatoire en AppleScript et qu'elle l'est en assembleur. En fait le compilateur AppleScript en ajoute une à la place du "end" qui termine la fonction.



On voit tout de suite un truc marrant : le sous-programme peut contenir des instructions empiler ou dépiler, mais il vaut mieux pour tout le monde que l'adresse de retour soit vraiment au sommet de la pile lorsque le RTS est exécuté sinon l'exécution se poursuit ailleurs (la donnée pointée par le pointeur de pile est interprétée comme adresse de retour, sans aucun état d'âme).

Dans l'exemple suivant, il y a une légère différence avec le précédent : lors de la deuxième exécution, le programmeur a oublié de dépiler une donnée, juste au dessus de l'adresse de retour empilée par JSR. C'est cette donnée qui sera utilisée. Inutile de dire que le résultat sera inattendu :



La gestion de la pile réclame donc pas mal de rigueur lorsqu'on programme en assembleur et personne ne sera étonné qu'un dispositif aussi explosif ne soit directement manipulable par aucun langage de programmation évolué.

Pourquoi ne pas réserver la pile à ces adresses de retour uniquement et en faire une structure de données privées pour éviter complètement les erreurs ?

Parce que la pile est le seul moyen d'implémenter des variables automatiques et d'assurer la réentrance des procédures. C'est ce que nous allons voir maintenant.

Variables et compilateurs.

Implémentation des variables.

Voilà qui nous intéresse plus directement en tant que programmeur en langage évolué. Une variable est un couple nom/valeur. Lorsque l'on écrit :

```
set x to 1
```

ou en C

```
integer x = 1;
```

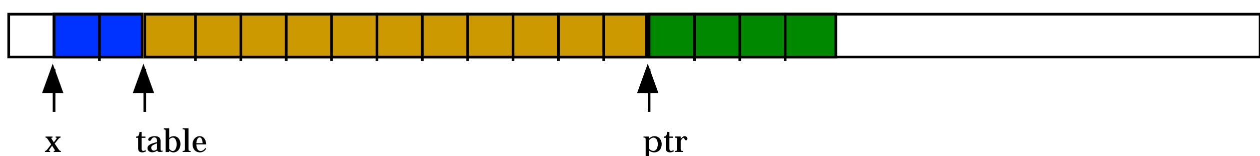
le nom de la variable est x et sa valeur (actuelle) 1.
La valeur d'une variable peut changer, mais chaque nom différent désigne une variable différente.
En fait, chaque variable est une zone mémoire réservée au stockage de sa valeur.

Comme nous l'avons vu, le µP travaille sur des adresses et non des noms de variables, donc les compilateurs effectuent les opérations suivantes : lorsque la variable est déclarée (en AppleScript, la première apparition de la variable suffit), le compilateur alloue à cette variable une portion de mémoire et utilisera l'adresse de cette zone dans le code objet à chaque fois que la variable est mentionnée.

C'est donc une substitution : quand le compilateur rencontre la variable 'marignan' et qu'il décide arbitrairement (c'est son boulot) de la ranger à l'adresse 1515, il remplacera toute mention de la variable marignan par 1515.

Une sorte de dictionnaire est donc géré par le compilateur au fur et à mesure des allocations : ce dictionnaire comprend les noms des variables et les adresses des zones mémoire qui lui seront associées lors des exécutions. Lorsque le code objet a été créé, ce dictionnaire n'est plus utile et n'est pas conservé : dans le code objet. On ne retrouve donc pas les noms des variables[26] dans le code objet, mais des adresses qui ont été calculées par le compilateur.

Celui-ci réalise donc une sorte de carte d'occupation de la mémoire réservée au programme et remplit ce bloc au fur et à mesure des besoins.



Dans l'exemple, on a déclaré un entier 'x', une suite de 11 caractères 'table' et

[26] Sauf si (c'est agaçant, il y a toujours des mais !) vous avez demandé de compiler avec des infos de debug. Option inconnue en AppleScript, mais disponible en C et autres.

une référence (ou pointeur) qui occupe 4 octets 'ptr'. Le compilateur utilise les adresses 1, 3 et 14 en lieu et place de 'x', 'table' et 'ptr'.

On voit tout de suite pourquoi la plupart des langages imposent une déclaration ou un typage précis des variables : c'est tout simplement parce que le compilateur a normalement besoin de savoir combien d'octets sont nécessaires pour stocker la valeur d'une variable :

un entier (un, deux ou parfois quatre octets) n'occupe pas la même place qu'un nombre flottant (sept octets) ou une chaîne de caractères (???).

Dans les langages à typage fort (comme Pascal, C, Java, etc…), chaque variable se présente comme une étagère parfaitement rigide dans laquelle il est impossible de “forcer” une donnée de taille différente.

Il est tout à fait impossible de stocker directement un nombre en virgule flottante dans une variable déclarée comme entier, parce que, en général, sept octets ne “rentrent” pas dans quatre.

Comment faire dans le cas d'une chaîne dont la longueur n'est pas connue d'avance ? On réserve une zone suffisamment largement dimensionnée en “espérant” que cela suffira pour tous les besoins du programme[27] .

Dans les langages à typage faible (comme AppleScript où l'on peut ranger dans une variable exactement ce que l'on veut), cette allocation de mémoire n'est pas faite une fois pour toute par le compilateur, mais au moment de l'exécution. C'est ce qu'on appelle l'allocation dynamique.

Le compilateur se contente d'allouer des références[28] . Naturellement, si cette technique est plus souple, elle n'est pas gratuite, car le programme devra faire toute une gestion de mémoire qui pénalise l'exécution par rapport aux langages à typage fort.

Le processus de compilation.

Puisque nous en sommes à parler de cette allocation mémoire, décrivons un peu plus le travail du compilateur et de ses joyeux complices : celui-ci va donc remplacer les variables par des adresses. Celles-ci ne seront pas les adresses définitives, car l'emplacement mémoire auquel le programme va être chargé est différent selon les exécutions. Le compilateur va donc calculer des adresses relatives au début du bloc mémoire réservé à l'application (c'est à dire comptées comme si le premier octet alloué portait le numéro 0).

Il laissera également “en blanc” les adresses qu'il ne connaît pas (les références aux variables ou aux procédures dites “externes” qui font partie du même

[27] Cet espoir ne peut manquer d'être régulièrement déçu, si bien qu'il vaut beaucoup mieux que le programme vérifie si la chaîne qu'il prétend écrire n'est pas trop longue. En C par exemple, il est parfaitement possible de stocker une chaîne de caractères dans un bloc trop petit pour la recevoir. Que se passe-t-il alors ? Les variables suivantes sont écrasées, ce qui provoque en général des désordres plus ou moins graves. En anglais on appelle ça “buffer overflow”, terme que les familiers du hack et des failles de sécurité reconnaîtront tout de suite.

[28] Les références sont étudiées en détail plus loin.

programme, mais ont été compilées séparément[29]), et celles-ci seront remplies par l'éditeur de liens (anglais : linker) qui s'exécute le plus souvent juste après la compilation.

Le compilateur ajoute en plus du code standard qui n'a pas été écrit par le programmeur mais permet un lancement propre (initialisation du programme) et diverses routines indispensables au fonctionnement (comme la gestion de mémoire en AppleScript ou en Java).
Ce code supplémentaire est souvent appelé run time[30] .

Le produit du compilateur est appelé "code objet", et celui de l'éditeur de liens "exécutable".

Enfin, le système, lors du lancement, va attribuer un bloc mémoire au programme et le chargeur (anglais : loader) va calculer les adresses définitives en mémoire à partir de l'origine maintenant connue du bloc alloué. Elles ne sont donc valables que pour cette exécution. Le chargeur s'occupe aussi des librairies dynamiques (dll) et autres segments de code quand ils existent. Le mécanisme exact est assez complexe, mais le principe reste le même.

Ouf ! Maintenant le programme est réellement opérationnel et peut être exécuté pour réaliser toutes les opérations prévues (et parfois imprévues) par son concepteur.

Signalons un point important : à partir du code exécutable en langage machine, on ne peut remonter "à l'envers" jusqu'au programme source en langage évolué. D'abord parce que le dictionnaire qui fait correspondre les adresses et les noms de variables et de fonctions n'est plus disponible : le compilateur peut l'incorporer au code objet si on le lui demande mais c'est seulement une option.

Ensuite parce que l'on ne peut pas dire à coup sûr que tel ou tel bloc d'instructions en langage machine correspond à une instruction précise dans le langage évolué : diverses constructions syntaxiques peuvent donner le même résultat. D'autre part, bien des compilateurs optimisent le code objet, c'est à dire suppriment des instructions redondantes ou même retouchent l'ordre des opérations.
Quand on compile, il y a donc une perte d'information par rapport à la version source.

En AppleScript, c'est normalement différent, car la version source est enregistrée dans les scripts et applis compilés (ressource "scrpt"), à moins que vous ne demandiez l'enregistrement sous forme de script "exécutable seulement". Dans ce cas, le fichier produit est un exécutable qui ne permet pas de remonter à la version source, comme ceux que nous venons de décrire. Pour les applications et scripts ordinaires, une compilation de ce source contenu dans la ressource "scrpt" a

[29] Cette étape n'existe pas en AppleScript car il est impossible de compiler un script en plusieurs fois. Elle est par contre importante dans les gros projets où l'on ne souhaite pas recompiler la totalité du source à chaque modification. Certains de ces projets peuvent même utiliser des langages différents. Dans ce cas, la compilation séparée est indispensable.
[30] Désolé, je ne connais pas d'équivalent français. Run time signifie "au moment de l'exécution".

lieu au vol, avant l'exécution. Pour les scripts courts, ceci n'entraîne pas de retard notable, mais on gagne du temps lorsque les scripts sont longs à les compiler comme “exécutables seulement”.

Variables automatiques.

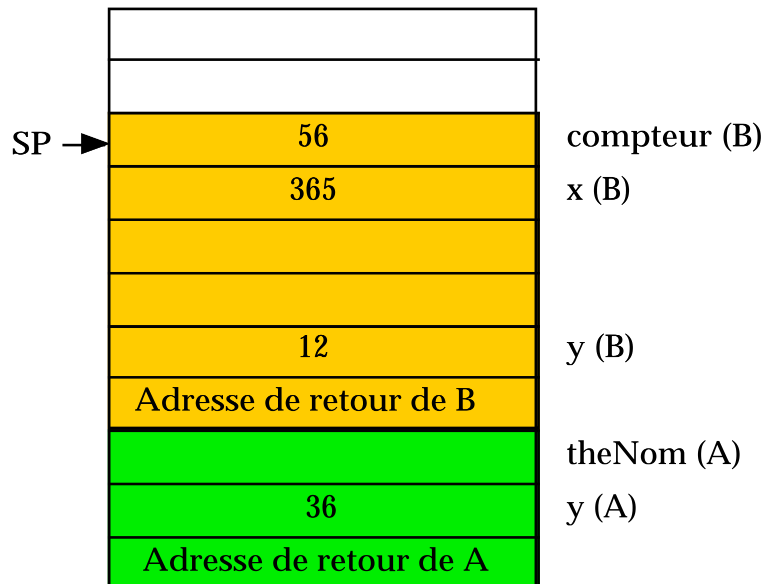
Nous avons parlé de l'allocation des variables globales. Il en existe une autre sorte appelée automatiques ou locales[31] . La distinction est importante car les variables auto sont allouées dans la pile.

Lors de l'appel d'un handler ou d'une routine, le compilateur va ajouter à l'instruction d'appel de sous-programme l'empilement de toutes les variables locales ou automatiques utilisées dans la routine.

En fin de handler, il dépile toutes ces variables de manière à restaurer la pile dans l'état initial avec l'adresse de retour au sommet, comme nous l'avons vu plus haut dans l'appel des sous-programmes.

L'adresse des variables locale ne sera donc pas une case mémoire précise (comme pour les globales), mais un déplacement par rapport à la valeur du pointeur de pile.

Voici un exemple de pile :



le programme principal a appelé une routine A qui déclare deux variables locales, y et theNom, puis cette routine appelle une routine B qui en déclare trois autres, compteur, x et y. Nous avons donné ces exemples de noms de variable pour préciser les idées, mais naturellement, ils ne figurent que dans le code source.

A la compilation, ces noms sont remplacés par des positions relatives par rapport à SP :

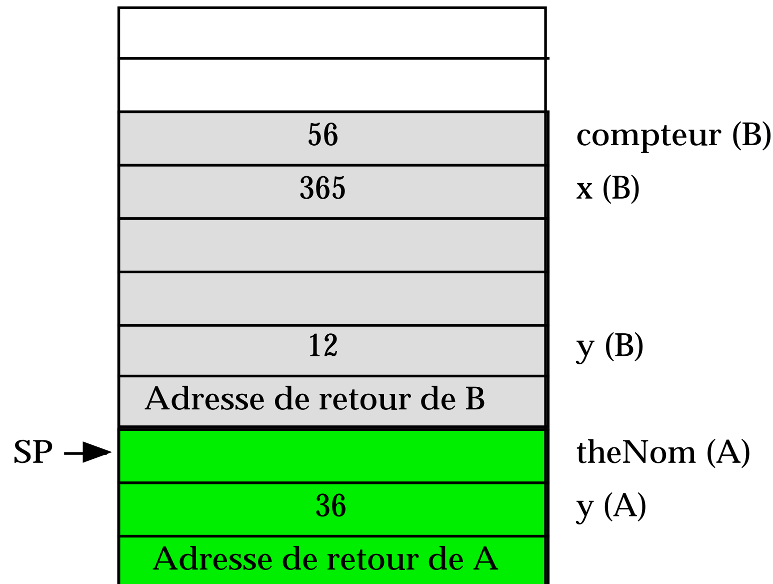
Dans le corps de la routine B :

compteur: SP

[31] Les deux sont synonymes.

x: SP + 1
y: SP + 4
Dans le corps de la routine A :
theNom: SP
y: SP + 1

On remarquera que ces positions relatives sont communes par exemple aux variables x de la routine B et y de la routine A. Mais lorsque la routine B finit de s'exécuter, la pile devient :



ce qui montre que SP + 1 ne désigne plus la même case mémoire que dans le corps de la routine B. Nous avons représenté sur fond gris les variables locales de la routine B pour montrer qu'une fois celle-ci terminée, les valeurs ne sont pas effacées pour autant. Cependant, elles ne sont plus utilisables, car un nouvel usage de la pile (appel d'une autre routine par exemple) ne manquera pas de les surcharger.

On voit donc que la mémoire située au dessus du SP est perpétuellement menacée d'être engloutie par un nouvel appel de sous-programme. Elle ne peut donc être utilisée de manière fiable[32] .
Ceci a deux conséquences importantes :

La première est que les variables locales n'existent (et ne sont donc utilisables) que pendant l'exécution de la procédure qui “les crée”. En dehors de ce moment, les cases mémoires utilisées existent bien entendu, mais contiennent des valeurs qui peuvent changer à tout moment en dehors du contrôle du programme. D'ailleurs, aucun langage évolué ne permet de les utiliser en dehors du bloc ou handler dans lequel elle sont définies.

La deuxième est que les variables locales sont différentes lorsque la même procédure est appelée plusieurs fois simultanément. Le premier cas est la réentrance : une procédure peut être appelée une nouvelle fois avant que le

[32] Raison pour laquelle il faut manipuler avec précaution les pointeurs et références sur ces variables. Le pointeur peut rester valide alors que la variable a disparu (mais non son emplacement mémoire, bien entendu).

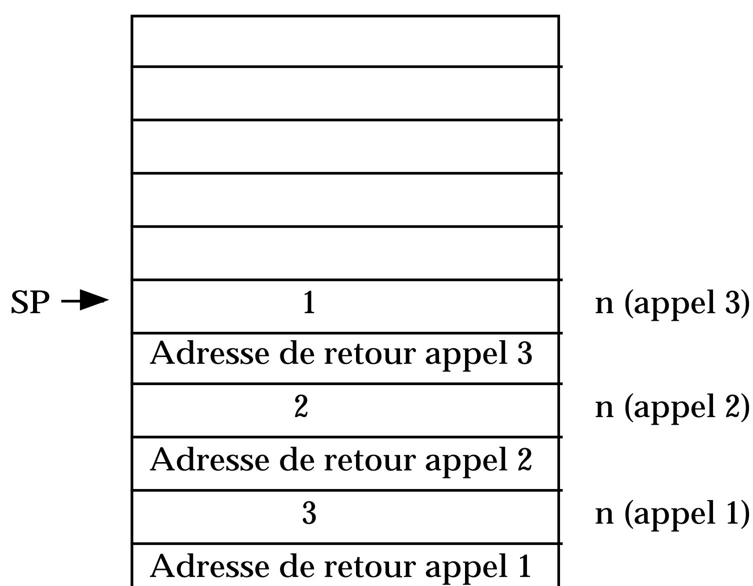
l'exécution du premier appel soit terminé. Si elle fait appel à des variables globales, la deuxième exécution, surchargeant ces variables, risque de perturber la première et réciproquement. Si la procédure ne fait appel qu'à des locales, les deux exécutions utiliseront en fait des zones mémoire différentes, situées à des endroits différents dans la pile.

Il en est de même pour les procédures récursives (qui s'appellent elles-mêmes). Chaque nouvel appel alloue sur la pile des zones mémoires différentes, si bien qu'il existe un jeu de variables automatiques complet par appel.

Voici un exemple du fonctionnement sur la procédure factorielle[33] .
La procédure factorielle s'écrit récursivement de la façon suivante :

```
on fact(n)
  if n = 1 then
        return 1
  else
        return n * (my fact(n - 1))
  end if
end fact
```

Si l'on appelle ce handler avec le paramètre 3, il y aura deux autres appels récursifs de fact() et la pile présentera l'aspect suivant au cours du dernier appel :



On voit assez clairement sur cet exemple qu'il y a en fait trois variables locales portant le même nom, mais de valeurs bien distinctes puisqu'elles occupent des emplacements mémoire différents.

On voit aussi que malgré son élégance indéniable, la récursivité est rarement une méthode à conseiller : remplir la pile avec la suite des nombres de 1 à n, entrelardés d'autant d'occurences de la même adresse de retour n'est pas un modèle d'efficacité. On voit plus généralement pourquoi les procédures récursives sont dangereuses : elles font grossir la pile de manière difficile à prévoir. A réserver donc

[33] Malgré son classicisme, cet exemple n'est pas une manière recommandable de calculer des factorielles. De plus en AppleScript, cette fonction est incorrecte car elle conduit toujours à un débordement de pile si on lui passe un paramètre qui n'est pas un entier strictement positif.

aux cas où le nombre d'appels imbriqué est petit et clairement limité.

Pointeurs et références.

Nous avons parlé des registres d'adresse du µP et de leur usage pour accéder à des zones mémoires, donc à des données. Une partie des adresses dont on peut avoir besoin est calculée par le compilateur et donc stockée comme constante dans le code objet du programme lui même. Mais il est également possible de stocker des adresses calculées ou déterminées pendant l'exécution dans une variable. L'accès à la donnée se fera donc en deux temps :

1-Récupérer son adresse dans une variable P et la placer dans un registre adresse.

La variable P qui contient une adresse est ce qu'on appelle un pointeur ou une référence. Du point de vue du µP et du code objet, il n'y a pas de différence. Ce sont les langages évolués qui introduisent cette différence qui est finalement assez minime. Comme les pointeurs n'existent pas en AS, nous n'en parlerons pas.

En AppleScript, on peut utiliser soit la donnée elle-même soit une référence à cette donnée, le résultat est le même :

```
set x to 1
set reff to a reference to x
display dialog x as text
display dialog reff as text
```

les deux dernières lignes donnent exactement le même résultat, même si le fonctionnement sous-jacent n'est pas exactement le même. Dans le premier cas, l'adresse de la variable x est une constante calculée par le compilateur, dans l'autre, on utilise une variable contenant cette adresse : il y a donc une instruction supplémentaire pour accéder à cette variable et y récupérer l'adresse.

1- Charger l'adresse de x (constante)	1- Charger l'adresse de reff
-------------	2- Charger le contenu de reff = adresse de x
3- Charger le contenu de x	3- Charger le contenu de x

Cette étape intermédiaire s'appelle déréférencer. Elle est automatiquement insérée par AppleScript lorsque c'est nécessaire (lorsque l'on utilise une référence et non une variable ordinaire), mais la plupart des langages ont une syntaxe spéciale pour distinguer les deux et vous obligent à faire ce travail vous-même[34] .

---

[34] C'est l'inconvénient des langages très évolués comme AS: ils facilitent les choses en masquant des fonctionnements comme celui des références, mais on peut être fort embistrouillé pour savoir exactement ce qui se passe quand on obtient pas le fonctionnement que l'on désire.

Variables AppleScript.

A quoi servent les références ?

Les références ne sont pas des subtilités théoriques. Elles servent pour résoudre des problèmes pratiques comme les variables en AppleScript. On a vu qu'en raison du typage faible, il est impossible au compilateur de créer une carte des variables puisqu'il ne sait pas ce que l'on va y ranger, donc la place qu'il faut réserver.

Or, il faut bien calculer des adresses utilisables par le processeur. Le compilateur crée donc des références pour chaque variable. Comme les adresses ont toujours la même taille, il n'y a aucun problème de calcul de taille et le tour est joué. Le compilateur réserve donc 4 octets pour chaque variable et y stocke la valeur NIL.

NIL est une constante prédéfinie qui signifie que la référence ne contient aucune adresse valide.

Ceci explique pourquoi à l'exécution on peut avoir une erreur si on cherche à manipuler le contenu d'une variable dans laquelle rien n'a été stocké. Cette variable n'existe pas réellement, car il n'y a encore aucune mémoire allouée pour sa valeur (et on ne peut rien allouer a priori puisqu'on ne sait pas encore ce qui va y être rangé). Il y a juste une référence vide que le programme repère, d'où l'erreur.

Quand on affecte une valeur à une variable, alors un bloc mémoire de la taille convenable est alloué dans le tas, et l'adresse de ce bloc stockée dans la référence.

Il s'ensuit une chose importante : à chaque affectation et modification de la variable, un nouveau bloc valeur peut être créé et remplacer l'ancien qui est libéré.

Pourquoi ne peut-on pas réutiliser l'ancien ? Parce que la valeur n'a pas forcément la même taille en octets que la précédente. Ce fonctionnement bien pratique pour le programmeur, est loin d'être efficace dans un certain nombre de cas. Par exemple, ajouter un caractère à la fin d'une chaîne :

```
set chr to "abcdefg"
set chr to chr & "h"
```

n'est pas exécuté comme un ajout à un bloc valeur existant (donc le stockage du caractère “h” à un emplacement approprié), mais comme la création d'une nouvelle chaîne dans laquelle on recopie l'ancienne avant d'y ajouter le caractère. On comprend alors pourquoi ce type d'opération peut devenir très lent sur des chaînes de grande longueur et qu'il vaut mieux les manipuler par petits bouts.

Nous allons étudier comment fonctionnent les références et les variables dans le détail.

Eclaircissons tout d'abord un point capital : dans une variable AppleScript[35] , on peut mettre n'importe quel type de donnée du langage et ceci à n'importe quel moment de l'exécution. C'est assez commode, car on ne rencontre jamais d'erreur,

[35] Ou une property.

mais il est aussi plus difficile de savoir exactement ce que l'on est en train de manipuler. L'objet de cette section est d'éclaircir tout cela.

Dans toute la suite, nous allons parler de variables de type liste ou référence. C'est un raccourci de langage car les variables ne sont pas typées : il faudrait parler à la place de variables contenant (provisoirement) une liste, un record, un script-objet et dont le contenu peut changer.

Ces trois types de valeurs ont en effet la spécificité de pouvoir être partagés. La variable contenant un (ou plusieurs) de ces types ne réagira pas de la même façon dans certains cas.
Le type référence étant une référence comme son nom l'indique.
La distinction que nous faisons entre variables simples et variables listes etc… est donc une commodité : elles sont en fait toutes pareilles, mais les règles qui s'appliquent ne sont pas les mêmes et dépendent de leur contenu.

Variables simples.

C'est le cas des variables contenant un nombre ou une chaîne ou tout autre classe de valeur que nous n'avons pas définie comme de type liste. Le comportement est sans mystère. Pour leur affecter une valeur, les instructions **set** et **copy** sont équivalentes et donnent le même résultat.

```
set var to "Ah !"
set reff to var
set var to "Oooooh !"
log var -->"Oooooh !"
log reff -->"Ah !"
```

Ce script montre que lorsque l'on crée la variable **reff** on lui donne la valeur contenue dans **var** au moment de la création. Si par la suite la valeur contenue dans **var** change, **reff** n'en est pas affectée. On peut représenter ceci dans le schéma suivant :

set var **to** "Ah !"	var "Ah !"	
set reff **to** var	var "Ah !"	reff "Ah !"
set var **to** "Oooooh !"	var "Oooooh !"	reff "Ah !"

```
log var --"Oooooh !"
log reff --"Ah !"
```

Chacune des variables possède son propre contenu (ou valeur) indépendant et séparé que nous représentons sous la forme d'un rectangle noir. Une nouvelle affectation remplace simplement le contenu de ces données par autre chose.

Rappelons simplement que ce rectangle qui représente une zone de mémoire,

est réservé soit dans la pile si la variable est une locale, soit dans le tas si c'est une globale ou une property. Ceci n'a aucune importance en général, mais pas quand on crée des références.

Voyons maintenant ce qui se produit lorsque l'on crée une référence. Au lieu de :

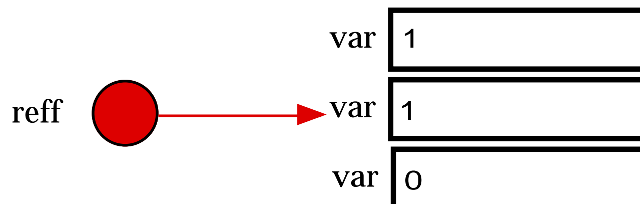
```
set var to 1
set reff to var
```

nous utilisons :

```
set var to 1
set reff to a reference to var
```

Dans ce cas, la variable **reff** ne contient pas une valeur, mais une adresse qui désigne la variable **var**. Comme le montre le schéma ci-dessous, il n'y a plus deux valeurs numériques distinctes mais une seule :

```
set var to 1
set reff to a reference to var
set var to 0
log reff -- 0
log var -- 0
```



Nous avons représenté reff par un rond rouge et une flèche pour bien montrer que l'organisation interne de cette variable a changé. Selon le cas elle contient le contenu de var ou une référence à var[36] . Voyons cela de plus près :

Utiliser **reff** dans une expression arithmétique ou dans une lecture (comme log) revient à utiliser le contenu de **var**. AppleScript effectue les opérations nécessaires de manière transparente. Par contre **set** (ou **copy**) donnent des résultats un peu surprenants :

```
set var to 1
set reff to a reference to var
set var2 to reff + 1 -- reff est remplacé par 1
set var to 0
log var2 --> 2
```

On voit sur cet exemple que var2 n'est pas une référence : c'est une nouvelle variable de type integer, par contre,

```
set var to 1
set reff to a reference to var
set reff2 to reff
set var to 0
log reff2 --> 0 –– reff2 ne contient pas 1 mais une référence à var
```

le résultat de la duplication de reff en reff2 n'est pas une nouvelle variable

---

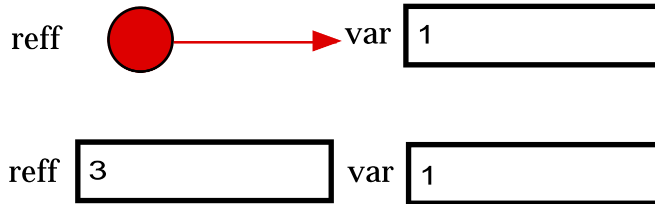
[36] On ne peut pas déterminer directement si une variable contient une référence ou une valeur: une lecture par **get** ou **get** contents **of** renvoie le même résultat dans les deux cas, log également. Seule l'instruction **return** reff permet de distinguer: elle retourne "var of «script» pour une référence, et une valeur pour une variable ordinaire.

numérique mais une nouvelle référence à **var** ! D'autre part :

```
set var to 1
set reff to a reference to var
set reff to 3 -- ceci supprime la référence !!!
log var --> 1 !! var est inchangée !
```

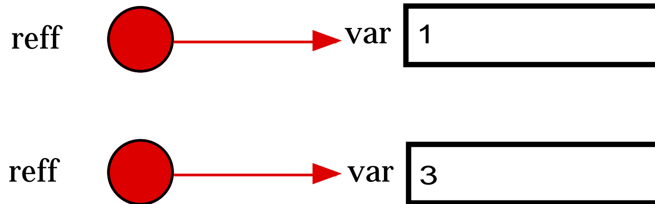
l'instruction **set** reff **to** 3 transforme **reff** en variable numérique contenant trois et supprime la référence à **var**. La valeur de **var** est inchangée.

```
set var to 1
set reff to a reference to var
set reff to 3
```



Pour modifier la valeur de **var** par l'intermédiaire de **reff**, il faut utiliser la syntaxe **contents of** :

```
set var to 1
set reff to a reference to var
set contents of reff to 3
log var -- 3
```



Passage des paramètres.

Que se passe-t-il lorsqu'on passe une variable simple comme paramètre à un handler ? En général, deux cas peuvent se présenter.

a/ le passage de paramètre par valeur : une copie du contenu de la variable est placée dans la pile. La variable d'origine ne peut être modifiée par le handler (sauf si elle est globale)

b/ le passage par référence : dans ce cas, c'est l'adresse de la variable qui est passée et le handler manipule la variable originale.

L'exemple suivant nous permet de déterminer ce qu'il en est :

```
set var to 1
my test(var)
log var --> 1

on test(x)
  log x --> 1
  set x to 2
  log x --> 2
end test
```

On voit donc que le passage de paramètres se fait par valeur : une variable locale x est créée par l'appel du handler. Elle est différente de la variable originale **var**.

Comment faire si l'on désire modifier **var** dans le handler **test**() ? La première solution est de déclarer **var** comme globale, mais il n'y a même plus besoin de passer

un paramètre :

```
global var
set var to 1
my test()
log var --> 2

on test()
  set var to 2
end test
```

Mais cette solution ne répond pas à tous les besoins. On peut également passer une référence à var comme paramètre :

```
set var to 1
my test(a reference to var)
log var --> 2

on test(x)
  set contents of x to 2
end test
```

Listes, records & scripts-objets

L'affaire se complique pour les listes, les records et les scripts-objets. Cette fois-ci, **copy** et **set** n'ont plus le même effet. La documentation AppleScript explique que **set** crée des données partagées alors que **copy** crée de nouvelles données.

Commençons donc par **copy**. Nous pouvons nous attendre à un comportement exactement identique à celui des variables simples et c'est effectivement le cas :

```
set liste1 to {1, 5, 15, 2}
copy liste1 to liste2
set item 1 of liste1 to 20
log liste1 --> {20, 5, 15, 2}
log liste2 --> {1, 5, 15, 2}
set liste1 to {}
log liste1 --> {}
log liste2 --> {1, 5, 15, 2}
```

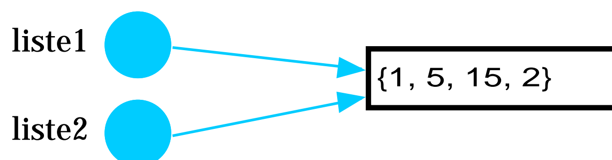
Vous pouvez constater que liste1 et liste2 sont deux variables dont le contenu est tout à fait indépendant, même si l'une a été créée à partir de l'autre. Par contre, la même suite d'instructions, sauf la duplication réalisée par **set** donne le résultat suivant :

```
set liste1 to {1, 5, 15, 2}
set liste2 to liste1
set item 1 of liste1 to 20
log liste1 --> {20, 5, 15, 2}
log liste2 --> {20, 5, 15, 2}
set liste1 to {}
log liste1 --> {}
log liste2 --> {20, 5, 15, 2}
```

On s'aperçoit que la modification d'un item de liste1 retentit sur liste2. Il n'existe donc qu'un seul jeu de valeurs. Par contre la modification de liste1 laisse intact le contenu de liste2.

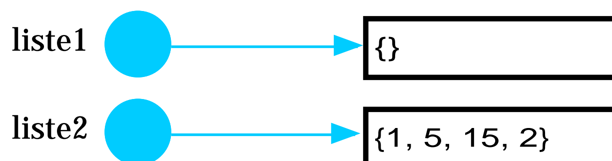
Comment ce partage fonctionne-t-il ? Techniquement parlant, liste1 et liste2 sont des références[37] , mais un peu différentes de ce que nous avons vu pour les variables simples. Elles ne pointent pas sur une autre variable, mais sur le contenu commun. Dans le croquis suivant, nous avons représenté ces références en bleu pour les distinguer des précédentes :

```
set liste1 to {1, 5, 15, 2}
set liste2 to liste1
set item 1 of liste1 to 20
```



Les données ne portent pas de nom et ne sont pas accessibles directement. On voit avec ce schéma pourquoi modifier l'item 1 de liste1 ou de liste2 revient au même. Modifier le contenu de la variable liste1 a l'effet suivant :

```
set liste1 to {}
```

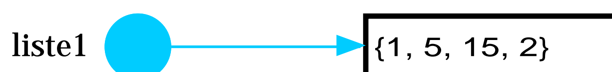


Une nouvelle zone de données est créée. Il n'y a pas, comme on pourrait l'imaginer de mise à zéro de l'ancienne zone de donnée attribuée à liste1 (et liste2).

<u>En résumé</u> :

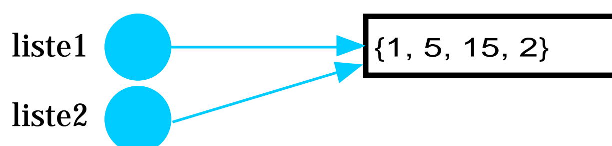
Le premier set (ou copy) crée les données et la première référence :

```
set liste1 to {1, 5, 15, 2}
```



Les set suivants, ajoutent des références sur les mêmes données :

```
set liste2 to liste1
```

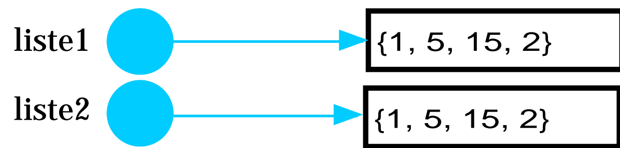


Par contre, copy crée de nouvelles données identiques :

---

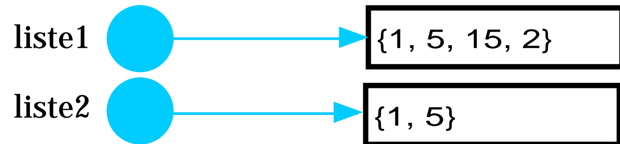
[37] C'est à dire que cette variable contient l'adresse ou l'identificateur de l'objet qu'elle désigne et non l'objet lui-même. Notez que la documentation AS emploie référence dans un sens plus restrictif: référence à une variable seulement (notées en rouge dans nos schémas). Elle traite de ce que nous décrivons sous une forme plus vague qui n'explicite pas le mécanisme du partage des données.

copy liste1 **to** liste2



Si l'on utilise des instructions qui modifient la taille de la liste originale[38] , il y a également création de nouvelles données :

set liste2 **to** items 1 thru 2 **of** liste1

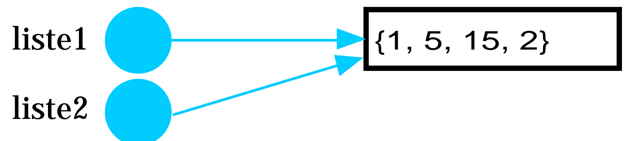


La seule exception à ce principe étant l'ajout d'élément en début ou fin de liste par **copy** ou **set**[39] :

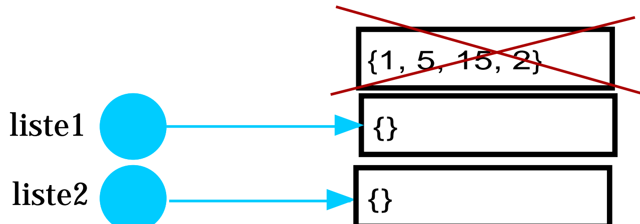
```
set liste1 to {1, 2}
set liste2 to liste1
copy 7 to the end of liste1 -- ou
-- set the end of liste1 to 7
log liste2 --> {1, 2, 7} -- liste2 pointe toujours sur les mêmes données que liste1
```

Enfin, lorsque nous supprimons toutes les références à une donnée, celle-ci devient inaccessible et disparaît, éliminée par le gestionnaire de mémoire[40] :

set liste2 **to** liste1



set liste1 to {}
set liste2 to {}



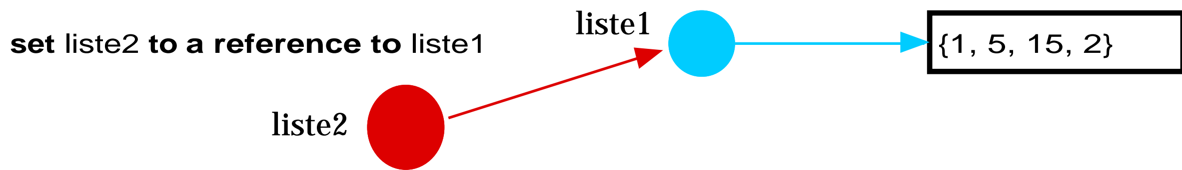
Qu'en est-il maintenant des références au sens de la documentation d'AppleScript ? Celles-ci pointent sur les variables, pas sur les données comme nous venons de le voir :

---

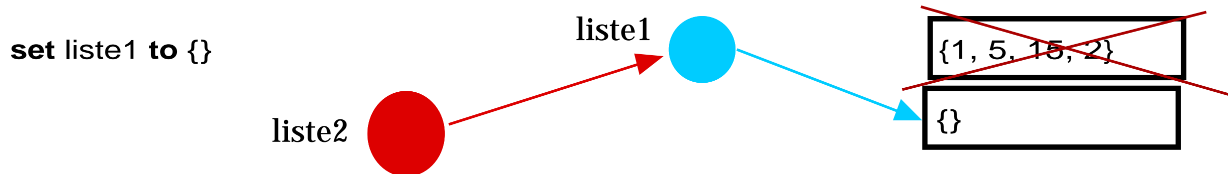
[38] En particulier la concaténation (&) et l'emploi de la propriété **rest** d'une liste.

[39] Ce qui explique vraisemblablement pourquoi ces instructions sont plus rapides que l'opérateur de concaténation &: **set** liste1 **to** liste1 **&** {7}.

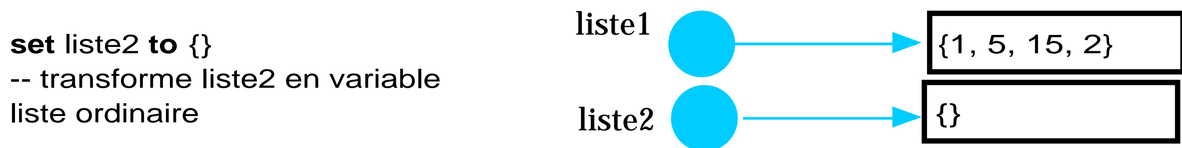
[40] Le fonctionnement est sans doute semblable à la gestion des références d'objets en Java ou celle des inodes UNIX: un compteur est associé à chaque donnée, il est incrémenté à chaque fois que l'on ajoute une référence et décrémenté quand on la change ou la supprime. Quand le compteur revient à zéro, cela signifie que toutes les références à cette donnée ont disparu: on peut donc la supprimer.



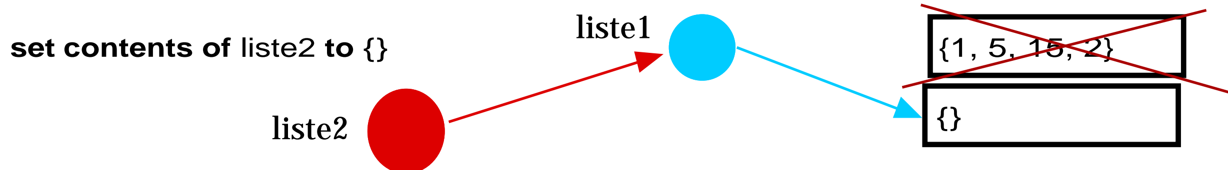
donc, une modification du contenu de liste1 n'aura pas le même effet que plus haut :



Cette fois, liste2 ne pointe plus vers la liste originale qui est perdue. D'autre part, il faut parfois utiliser la syntaxe **contents of** pour faire ce que l'on veut :

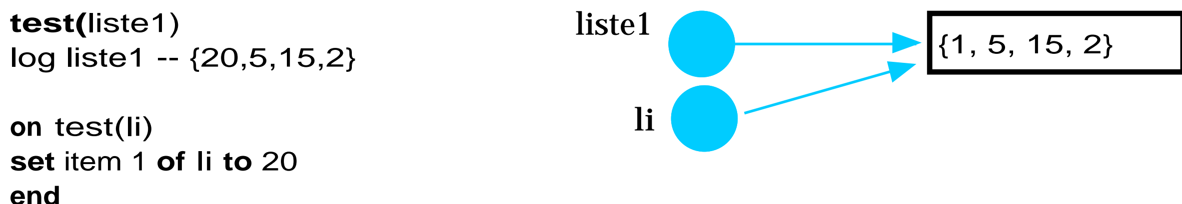


mais :



Passage des paramètres.

Cette fois, le passage d'un paramètre de type liste à un handler se fait par référence, mais une référence de type bleu, comme le montre l'exemple suivant :



L'appel crée une variable locale **li** qui pointe elle aussi vers les données. Donc, et c'est loin d'être sans importance, on manipule directement la liste originale en utilisant la variable locale **li**, pas une copie. Il est donc inutile et nuisible d'utiliser des références (rouges) pour l'appel à un handler. Au contraire, il faut prendre soin de dupliquer la liste originale si on souhaite la conserver intacte.

Cas complexes.

On peut rencontrer assez facilement des cas complexes où le fonctionnement devient assez difficile à débrouiller. Considérons l'exemple suivant :

```
set liste1 to {1, 5, 15, 2}
set liste2 to {}

repeat with elem in liste1
  if elem < 10 then
        copy elem to the end of liste2
  end if
end repeat
```

Je copie dans la liste2 tous les éléments de la liste1 plus petits que 10. J'obtiens :

```
log liste2 --{1,5,2}
```

ce qui paraît tout à fait normal. Si maintenant, je modifie le premier élément de la première liste :

```
set item 1 of liste1 to 20
log liste2 --{20,5,2}
```

Le résultat paraît déjà plus étrange. La raison en est que l'instruction **repeat with elem in liste1** renvoie des références (rouges), comme le précise la documentation, donc, ce n'est pas la valeur 1 que j'ai copié dans ma liste2 mais une référence pointant sur le premier item de la liste1[41] .

Si ensuite je modifie les items de ma liste2 en ajoutant 50 à tous ceux qui sont plus petits que 5 :

```
repeat with i from 1 to count of liste2
  if item i of liste2 < 5 then
        set item i of liste2 to (item i of liste2) + 50
  end if
end repeat

log liste2 -- {20, 5, 52}
set item 4 of liste1 to 3
log liste2 --{20, 5, 52}
```

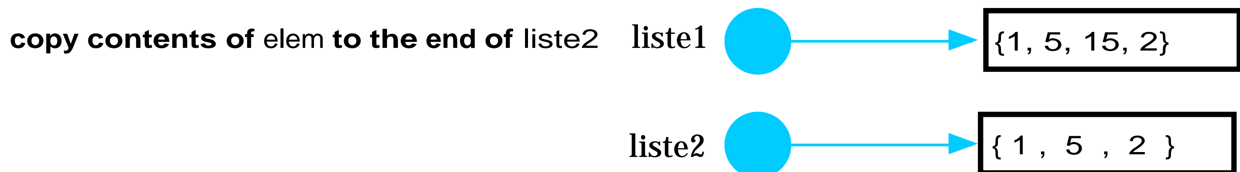
Le résultat paraît correct, mais une nouvelle étrangeté apparaît : cette fois, la modification du dernier élément de la liste1 n'apparaît pas dans la liste2. Pourquoi ? Parceque nous avons remplacé la référence à l'item 4 de liste1 par la valeur 52. La liste2 contient maintenant un mélange de références et de valeurs, ce qui peut conduire à des interrogations inextricables !

---

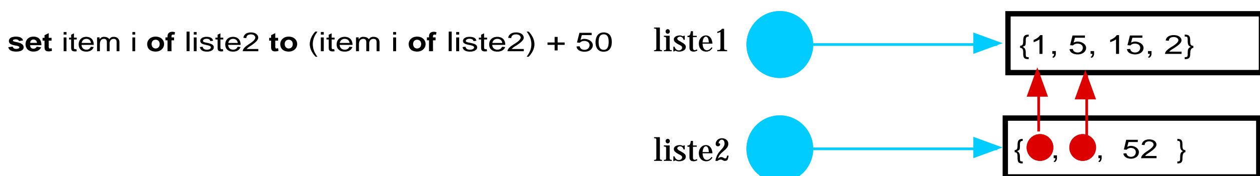
[41] On peut le vérifier directement en utilisant l'expression:
return liste2 --> *{item 1 of {1, 5, 15, 2}, item 2 of {1, 5, 15, 2}, item 4 of {1, 5, 15, 2}}*

Qu’avons nous fait exactement ? Nos schémas permettront de le comprendre :

On voit que toute modification dans liste1 retentit sur liste2. Si on voulait rendre les listes indépendantes, il aurait fallu écrire :



Ensuite, la modification directe des items de liste2 conduit à la situation suivante :



Où l’on voit que l’item 3 a perdu le lien avec l’item 4 de la liste1 alors que ce n’est pas le cas des deux autres. On pourra remarquer qu’en toute rigueur, il faudrait écrire :

```
contents of item 1 of liste2
```

au lieu de :

```
item 1 of liste2
```

puisqu’il s’agit d’une référence (rouge). AppleScript vous permet cette simplification qui n’est pas sans danger comme on vient de le voir. Mais que se passe-t-il si je supprime les données pointées par liste1 ?

```
set liste1 to {}
log liste2 -- {20, 5, 52}
```

Ces références devraient être “en l’air” et ne plus rien désigner, puisque liste1 ne contient plus aucun élément ! Mais dans ce cas, AppleScript remplace les références qui risqueraient d’être erronées par les valeurs correspondantes avant de les effacer :

```
set liste1 to {}
```

```
log liste2 -- {20, 5, 52}
```

Il en est de même si liste1 est raccourcie :

```
set liste1 to items 1 thru 2 of liste1
set item 1 of liste1 to 1
log liste2 -- {20, 5, 52}
```

On voit qu'ici tous les liens sont perdus, même si liste1 n'a pas perdu les items 1 et 2 seuls pointés dans liste2 !

Pour être complet, signalons que ce comportement (récupération des valeurs pointées par des références avant leur effacement), n'a pas lieu lorsque la liste contenant les éléments originaux fait partie d'un objet-script :

```
script sto
  property liste1 : {1, 5, 15, 2}
  property liste2 : {}
end script

repeat with elem in sto's liste1
  if elem < 10 then
        copy elem to the end of sto's liste2
  end if
end repeat

log sto's liste2 --{1,5,2}
set sto's liste1 to {}
log sto's liste2 --erreur !
```

Handles.

Un handle est une référence à deux étages. C'est une référence à une référence. Pour obtenir la valeur, il faut donc dé-référencer deux fois au lieu d'une. Cette notion n'est pas construite dans le langage AppleScript, mais le gestionnaire de mémoire du Macintosh l'utilise et on peut être amené à se servir de cette idée pour sa propre programmation.

Quel est l'intérêt de cette complexité supplémentaire ? Voyons cela en parlant du gestionnaire de mémoire du Macintosh.

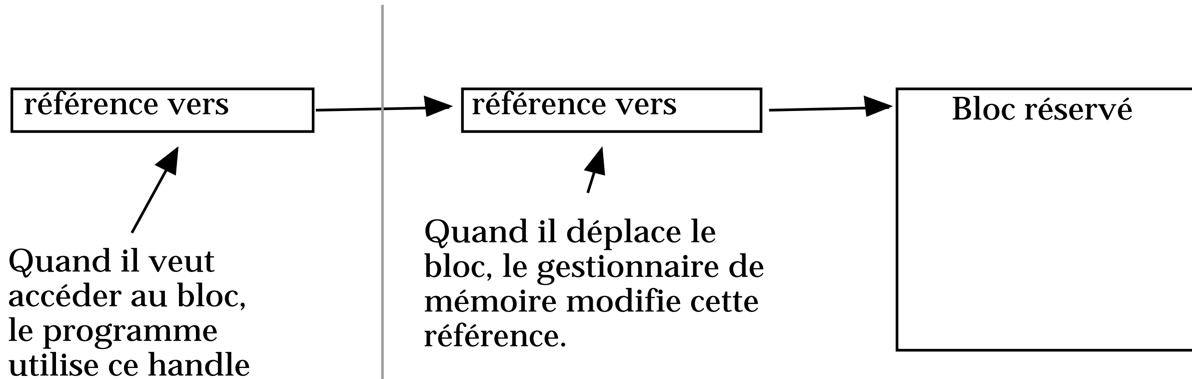
Lorsque les programme ont besoin d'une zone mémoire, ils appellent une routine du gestionnaire en précisant sa taille. Le gestionnaire leur réserve un bloc (quand c'est possible naturellement) et leur renvoie non pas une référence mais un handle pour dire à l'application quel est le bloc qu'elle peut utiliser.

Or le but du gestionnaire de mémoire n'est pas seulement de gérer la répartition des ressources disponibles mais aussi d'optimiser leur utilisation. C'est ainsi qu'il peut être amené à déplacer des blocs pour éviter la fragmentation ou les déplacer provisoirement sur le disque dur (mémoire virtuelle).

Or cela pose un problème : si le programme tente de manipuler les données du

bloc alors qu'il a été déplacé, il ne retrouvera pas ses données, mais quelque chose d'autre qui n'a rien à voir. Il faut donc mettre en place un mécanisme qui évite cela.

Il est naturellement impossible au gestionnaire de mémoire d'aller modifier une référence directement dans les variables du programme concerné, car il n'a aucune connaissance de la structure de ce programme et de l'organisation de ses variables. Par contre avec un handle, cela peut fonctionner : de la manière suivante :



Le programme ne modifie jamais la deuxième référence qu'il lit simplement chaque fois qu'il veut accéder au bloc. Le gestionnaire de mémoire lui ne modifie jamais la première dont il ne connaît pas l'emplacement.

Bien entendu, un signal permet au programme de verrouiller temporairement le bloc lorsqu'il a des accès répétés à y faire, de manière à ne pas pénaliser les performances.

AppleScript comme langage objet.

Introduction.

À cause de ses particularités techniques, AppleScript n'est pas un langage objet tout à fait comme les autres, mais il mérite bien ce nom, car il permet effectivement d'utiliser cette méthode de conception et de développement avec fruit.
C'est un des buts de l'objet-script.

Les scripts sont des objets. Pour ceux qui ne sont pas familiers de cette notion, nous commencerons par la présenter sommairement, puis nous exposerons son utilisation à travers un exemple assez développé qui exposera les avantages de la méthode par rapport aux autres techniques de développement.
Enfin, nous terminerons par une partie plus technique et générale sur AppleScript et le modèle objet : informations qui se révèlent fort utile pour comprendre le fonctionnement de certaines instructions comme **tell**.

Les scripts-objets : qu'est-ce ?

Définition.

Dans un programme AppleScript on peut définir n'importe où un bloc qui commence par script et finit par end script :

```
script monPremierObjet
  property a : 123
  property b : "Hello"

  on methode()
        display dialog b
  end methode

end script
```

Il est important de comprendre que ce bloc n'est pas exécuté en séquence comme une suite d'instructions ordinaires. Au contraire, il définit une structure complexe qui aura une existence séparée et autonome. C'est un script-objet.

Notons tout de suite un point important : il n'y a pas de différence entre un script-objet et un script, sinon le fait qu'un script occupe à lui tout seul un fichier, alors que les scripts-objets sont inclus dans un script principal. Les limites de bloc script---end script ne sont nécessaires qu'à partir du moment où plusieurs scripts-objets figurent dans le même fichier, mais c'est un détail de présentation. On peut

considérer que n'importe quel script est tout entier englobé dans un bloc script--end script implicite ou "sous-entendu".

Dans notre exemple précédent, nous pourrions parfaitement définir un script appelé "monPremierObjet" qui contient seulement le bloc :

```
property a : 123
property b : "Hello"

on methode()
  display dialog b
end methode
```

en le sauvant sur disque sous le nom "monPremierObjet", j'allais dire, comme d'habitude.

Ce script a une apparence tout à fait familière. Les limites de bloc n'introduisent donc aucun fonctionnement spécial ou inconnu. Elles permettent simplement de définir plusieurs scripts dans un même fichier et de définir une hiérarchie entre eux. Nous reviendrons en détail sur cette organisation à la fin.

Ce qui veut dire que, même si nous n'en tenons pas compte la plupart du temps, tous les scripts que nous créons sont en fait des scripts-objets.

Introduction.

La notion d'objet a émergé peu à peu en programmation quand on s'est rendu compte que les programmes avaient le plus souvent pour objectif de simuler le fonctionnement de choses ou de personnes réelles. Ces choses ou personnes (objets) ont des attributs qui les décrivent, ainsi que des règles de comportement.

Les attributs sont représentés par des données, et les règles de comportement sont des routines, donc des fragments de code.

Dans un programme classique, attributs et méthodes d'un même objet sont dispersés dans tout le programme, et l'on s'est rendu compte que les regrouper permettait une plus grande indépendance entre les objets et donc une plus grande facilité de maintenance et d'évolution.

En AppleScript, on regroupe donc ces éléments dans un bloc script. L'ensemble va constituer une description complète applicable à tous les objets de même espèce. C'est ce que l'on appelle une classe. Voyons maintenant plus en détail quels sont les éléments d'une classe, attributs et méthodes.

Les attributs.

Les attributs ou propriétés sont des valeurs qui définissent les propriétés particulières d'un objet. Ce sont aussi souvent des caractéristiques qui permettent de l'individualiser, de le distinguer par rapport aux autres. Dans l'exemple suivant, nous nous intéressons à des objets de la classe Personne et nous avons choisi de nous limiter à trois caractéristiques : la couleur de ses cheveux et de ses yeux et sa taille.

```
script Personne X
  property col_cheveux : "roux"
  property col_yeux : "bleu"
  property taille : 170
end script
```

Toutes les personnes possèdent l'attribut “taille”, et chaque personne possède une valeur individuelle qui est leur taille réelle. Dans le cas précis de cette personne, l'attribut taille a la valeur 170. Nous pouvons récupérer cette valeur par l'expression :

```
  get the taille of Personne
```

Finalement, quelle est la différence avec un record ? Le record permet aussi de regrouper des attributs et de les manipuler en bloc.

La différence est que l'on peut incorporer des méthodes (donc des routines) dans un script alors que le record est une structure de données pures.
On peut donc considérer en première approche un objet-script comme un record perfectionné.

Les méthodes.

Au script précédent, nous avons ajouté une méthode :

```
script Personne
  property col_cheveux : "roux"
  property col_yeux : "bleu"
  property taille : 170

  on affPers()
          display dialog "Cette personne a les yeux " & col_yeux
  end affPers
end script
```

Les méthodes sont des handlers au sens ordinaire du terme, incorporées dans le bloc script. Donc, comme d'habitude, on pourra les exécuter par une construction du genre :

```
tell Personne to affPers()
```

ou bien :

```
the affPers() of Personne
```

ou encore :

```
affPers() of Personne
```

Naturellement, rien ne nous empêcherait d'écrire à la place :

```
script Personne
  property col_cheveux : "roux"
```

```
  property col_yeux : "bleu"
  property taille : 170
end script

display dialog "Cette personne a les yeux " & col_yeux of Personne
```

construction qui paraît nettement plus simple. Il y a deux raisons à cette complexité apparente : nous avons déjà évoqué la première qui est le souci de regrouper en un même endroit (ou un même script) tout ce qui se rapporte à l'objet réel qu'il est censé représenter. La deuxième est encore plus intéressante, mais nous allons la voir sur un exemple développé.

L'objet par l'exemple.

Introduction.

Il reste encore bien des points à élucider après cette première description. Plutôt que de les traiter de manière théorique, nous allons les découvrir peu à peu sur un exemple de développement en quatre étapes. Nous y montrerons non seulement comment on définit des classes et des objets, puis comment on les utilise, mais de plus, nous donnerons pas à pas la comparaison avec un développement n'utilisant pas les techniques objet.

Naturellement, cet exemple a été choisi de manière à explorer un peu tous les aspects de la programmation objet en AppleScript et de mettre en valeur ses avantages par rapport aux techniques plus traditionnelles.

Première étape.

Spécifications.

Créer une application qui va nous permettre de gérer une liste de personnes avec les renseignements suivants :

nom (ou prénom)
âge

Quand on lance l'application, elle propose le choix entre :
- saisir une nouvelle personne
- afficher la liste des personnes avec leur nom et leur âge. (Par exemple, afficher la chaîne “nnnn a x ans”.)
- quitter

On devra naturellement retrouver les personnes saisies antérieurement en relançant l'application.

Construction des objets.

Comme cette application est censée gérer des personnes, nous allons définir un script-objet pour les représenter. D'après les exemples donnés dans le guide AppleScript, on pourrait tenter quelque chose comme :

```
script  personne
  property Nom : "Marie"
  property Age : 25
end script
```

mais ceci ne convient pas à notre propos. En effet, ce bloc crée effectivement un objet, mais il le range dans une globale appelée personne. Cela pourrait convenir si nous n'avions qu'une seule personne à gérer, mais ce n'est pas le cas.

Pour créer de nouvelles personnes et les conserver d'une exécution à l'autre, il nous faut donc :

1- créer une structure de données persistante pour y ranger les personnes. Ici, nous avons choisi d'utiliser une liste définie comme property :

```
property listeP : {}
```

2- écrire une routine qui va ajouter des personnes dans cette liste. Avec le script personne défini plus haut, il est parfaitement possible d'écrire :

```
display dialog ¬
  "Donner le nom de la personne" default answer ""
set nnom to (text returned of the result)
display dialog ¬
  "Donner l'âge de la personne" default answer ""
set aage to (text returned of the result) as integer

set the Nom of Personne to nnom
set the Age of Personne to aage

copy Personne to the end of listeP
```

donc, on demande à l'utilisateur un nom et un âge. On affecte ce nom et cet âge à l'objet personne, puis on en crée une copie que l'on ajoute à la fin de la liste. Ensuite, l'accès à ces différentes personnes pourra se faire en spécifiant l'item x de la liste. Par exemple :

```
get the Nom of item 1 of listeP
```

Comme on le voit, l'objet Personne lui-même ne sert à rien sinon à donner le modèle de l'objet ou si l'on préfère, la définition de sa classe. Mais la variable Personne ne contient rien de significatif. Elle contient simplement la dernière personne entrée, ou Marie, 25 ans au lancement de l'application.

C'est la raison pour laquelle il vaut mieux utiliser une autre syntaxe pour la création des objets : la fonction suivante est ce qu'on appelle un constructeur. Elle est entièrement équivalente à ce que nous avons écrit plus haut, mais présente les avantages suivants : elle ne crée pas de variable inutile et ne réclame pas de donner un nom au script :

```
on creaPersonne(sonNom, sonAge)
  script
        property Nom : sonNom
        property Age : sonAge
  end script
end creaPersonne
```

L'affectation des valeurs des attributs Nom et Age se fait automatiquement en

reprenant les valeurs passées en paramètre. Donc, avec cette syntaxe, nous pouvons écrire simplement[42] :

```
set the end of listeP to creaPersonne(nnom, aage)
```

Cette instruction appelle le constructeur qui crée un objet dont les attributs ont les valeurs passées en paramètre et renvoie cet objet que l'on stocke à la fin de la liste. Cette liste étant une property du script principal, elle est conservée d'une exécution à l'autre et on peut même y accéder depuis un autre script.

Dernière chose : nous allons avoir besoin d'afficher les infos sur une personne. On pourrait naturellement écrire la routine de construction de la chaîne dans la routine qui traite de l'affichage à la suite du choix de l'utilisateur.

La solution objet est d'incorporer cette routine dans la définition de la classe :

```
on creaPersonne(sonNom, sonAge) -- la classe fondamentale
  script
        property Nom : sonNom
        property Age : sonAge

        on getInfos()
              return ("Je m'appelle " & Nom & " et j'ai " & Age & " ans.")
        end getInfos
  end script
end creaPersonne
```

On appellera cette procédure par :

```
getInfos() of the item x of listeP
```

exactement comme si getInfos désignait une property. Dans notre script, on affichera cette chaîne dans un dialogue :

```
display dialog (getInfos() of the item x of listeP)
```

Voici maintenant le script complet de cette première étape. Le programme principal (run) affiche répétitivement le choix entre les trois options saisie, visualisation et quitter.

Les deux routines newPersonnes() et visuPersonnes() ne présentent pas de mystère. La seconde commence par afficher une liste complète des personnes dans laquelle on choisit, puis elle affiche les infos demandées.

```
(*    Script de démo des scripts-objets
Etape 1 avec la méthode objet
*)

-- Stockage des données : une liste d'scripts-objets.
-- Comme la property subsiste d'une fois sur l'autre je retrouverai ces données
```

[42] On pourrait imaginer d'écrire aussi

```
copy creaPersonne(nnom, aage) to the end of listeP
```

C'est une duplication inutile du script-objet, et, pour des raisons qui seront exposées plus loin, elle peut saturer la mémoire (cf. AppleScript est-il un langage objet ? - Héritage).

```
property listeP : {}

----------------- le programme principal -------------
on run
	repeat
		choose from list {"1- Saisir une personne", "2- Voir les personnes"} ¬
			cancel button name "Quitter"
		if result = false then -- = bouton Quitter
			exit repeat
		else
			set rep to first word of item 1 of result
			if rep = "1" then
				newPersonne()
			else if rep = "2" then
				visuPersonne("Liste des personnes :")
			end if
		end if
	end repeat
end run

--------------- Création des fiches personnes ---------------

on newPersonne()
	display dialog ¬
		"Donner le nom de la personne" default answer ""
	set nnom to (text returned of result)
	display dialog ¬
		"Donner l'âge de la personne" default answer ""
	set aage to (text returned of result) as integer

	set the end of listeP to creaPersonne(nnom, aage)

end newPersonne

on creaPersonne(sonNom, sonAge) -- la classe fondamentale
	script
		property nom : sonNom
		property age : sonAge

		on getInfos()
			return nom & " a " & age & " ans"
		end getInfos
	end script
end creaPersonne

-------------------- Visualisation --------------

on visuPersonne(aPrompt)
	-- 1- retour en cas de liste vide
	if listeP = {} then
		display dialog "Il n'y a aucune personne saisie !"
		return {}
	end if

	-- 2- constitution de la liste des informations des personnes
	set listeInfos to {}
	repeat with i from 1 to count listeP
		getInfos() of item i of listeP
```

```
        copy (i as string) & "- " & result to the end of listeInfos
    end repeat

    return choose from list listeInfos with prompt (aPrompt) ¬
        with multiple selections allowed and empty selection allowed

end visuPersonne
```

Une autre version plus classique.

Pour comparaison, voici l'autre version du même programme avec des méthodes classiques. On remarquera que toute l'organisation de la saisie est identique. Seule change la structure des données : on a utilisé des records au lieu de scripts-objets et il faut bien avouer qu'à ce stade, les différences sont minimes et paraissent sans importance : le code est identique, mais regroupé autrement. Pas de quoi changer ses habitudes.

```
(*      Script de démo des scripts-objets
Etape 1 avec la méthode traditionnelle
*)

-- Stockage des données : une liste de records.
-- Comme la property subsiste d'une fois sur l'autre je retrouverai ces données
property listeP : {}

---------------- le programme principal -------------
on run
  repeat
        choose from list {"1- Saisir une personne", "2- Voir les personnes"} ¬
            cancel button name "Quitter"
        if result = false then -- = bouton Quitter
            exit repeat
        else
            set rep to first word of item 1 of result
            if rep = "1" then
                newPersonne()
            else if rep = "2" then
                visuPersonne("Liste des personnes :")
            end if
        end if
  end repeat
end run

-------------- Création des fiches personnes ---------------

on newPersonne()
  display dialog ¬
        "Donner le nom de la personne" default answer ""
  set nnom to (text returned of result)
  display dialog ¬
        "Donner l'âge de la personne" default answer ""
  set aage to (text returned of result) as integer

  copy creaPersonne(nnom, aage) to the end of listeP
```

```
end newPersonne

on creaPersonne(sonNom, sonAge) -- création d'un record
  return {nom:sonNom, age:sonAge}
end creaPersonne

--------------- Les informations ---------------

on getInfos(pers)
  return nom of pers & " a " & age of pers & " ans"
end getInfos

------------------- Visualisation --------------

on visuPersonne(aPrompt)
  -- 1- retour en cas de liste vide
  if listeP = {} then
        display dialog "Il n'y a aucune personne saisie !"
        return {}
  end if

  -- 2- constitution de la liste des informations des personnes
  set listeInfos to {}
  repeat with i from 1 to count listeP
        getInfos(item i of listeP)
        copy (i as string) & "- " & result to the end of listeInfos
  end repeat

  return choose from list listeInfos with prompt (aPrompt) ¬
        with multiple selections allowed and empty selection allowed

end visuPersonne
```

Deuxième étape.

Spécifications.

Nous allons simuler les conditions d’un développement réel, où il est rare que l’on développe la version définitive d’une application du premier coup. Après quelques temps d’utilisation heureuse, nous nous avisons qu’il serait souhaitable de modifier quelque peu notre programme : pour des raisons de courtoisie, nous souhaitons ne pas afficher l’âge réel des femmes au dessus de trente ans, mais le résultat du calcul (âge réel - 20%).

Mais nous souhaitons cependant conserver l’âge réel de ces femmes dans l’attribut correspondant (on ne sait jamais !).

Comme on introduit la notion de sexe dans la base, on en profitera pour détailler les messages :
nnn est (un homme / un garçon / une jeune fille / une femme) de x ans.
La limite entre les catégories jeunes/adultes est fixée à dix huit ans pour les

filles, vingt et un pour les garçons.

Réalisation : notion d'héritage.

La solution évidente pour implémenter ces nouvelles spécifications consiste à introduire un attribut supplémentaire enregistrant le sexe dans les fiches personnes et à modifier notre procédure visuPersonnes en y incorporant des tests sur l'âge et le sexe de manière à construire le message adapté.

L'approche objet procède de manière différente.

Elle consiste à remarquer que nous avons maintenant deux types d'objets distincts, les femmes et les hommes, qui ont pourtant des caractéristiques en commun, leur nom et leur âge. Ce sont donc des catégories particulières d'une espèce plus générale.

On pourrait parfaitement écrire :

```
on creaHomme(sonNom, sonAge) -- les hommes
  script
        property Nom : sonNom
        property Age : sonAge

  end script
end creaHomme

on creaFemme(sonNom, sonAge) -- les femmes
  script
        property Nom : sonNom
        property Age : sonAge
  end script
end creaFemme
```

Mais on voit que finalement, il y a une partie commune entre les deux, tellement commune que la définition des scripts est identique (il n'y a pas de property sexe, mais ce n'est pas un oubli).

Cette situation où nous avons affaire à des catégories générales contenant des sous-catégories est suffisamment répandue pour que l'on aie mis en place un mécanisme spécial appelé héritage.

L'héritage est une relation entre classes. Quand une classe hérite d'une autre, elle possède ipso facto tous les attributs et toutes les méthodes de la classe parente. Ceci permet ainsi de ranger les attributs généraux dans la classe parente et les attributs particuliers dans la classe fille. Ici, nous conserverons donc la classe Personne que nous avons définie dans la première version et nous y ajouterons deux classes filles, homme et femme :

```
on creaPersonne(sonNom, sonAge) -- la classe fondamentale
  script
        property Nom : sonNom
        property Age : sonAge

        on getInfos()
              return nom & " a " & age & " ans"
        end getInfos
  end script
```

```
end creaPersonne

on creaHomme(sonNom, sonAge) -- les hommes
  copy creaPersonne(sonNom, sonAge) to zzz -- créer l'objet parent
  script
        property parent : zzz

  end script
end creaHomme

on creaFemme(sonNom, sonAge) -- les femmes
  copy creaPersonne(sonNom, sonAge) to zzz -- créer l'objet parent
  script
        property parent : zzz

  end script
end creaFemme
```

Comment est définie la relation d'héritage ? Par la property parent. Celle-ci doit spécifier un objet existant de la classe parente au moment de la création du nouvel objet. Cet objet parent est copié dans le nouvel objet dont il fait partie intégrante.

C'est la raison pour laquelle le constructeur des deux classes commence par créer un objet général Personne dans la variable temporaire zzz et y associe le nouvel objet Femme ou Homme[43] .

Ces deux classes d'objets peuvent paraître indiscernables à première vue, mais c'est faux : AppleScript ne confond pas les objets créés par l'un ou l'autre des constructeurs, même si leur structure est absolument identique.

C'est cette propriété qui nous permet de garder la trace du sexe, même si celui-ci n'apparaît pas dans les attributs.

Naturellement, nous pourrions placer dans les classes Femme et Homme de nouveaux attributs spécifiquement féminins ou masculins qui en feraient réellement des objets différents[44] .

Pour utiliser ces nouvelles classes, nous ajouterons en fin de newPersonne() la choix entre femmes et hommes, puis :

```
  choose from list {"Femme", "Homme"}
  if the first item of the result is "Femme" then
        set the end of listeP to creaFemme(nnom, aage)
```

[43] Contrairement aux autres langages objets où l'on spécifie l'héritage d'une classe par rapport à une autre, AppleScript définit l'héritage par rapport à une instance de classe (donc par rapport à un objet précis et non la définition générale de la classe). C'est pourquoi on est obligé de créer d'abord une instance de la super classe pour pouvoir la spécifier comme super classe. Cette instance ne sert à rien, mais elle est copiée dans l'instance de la sous-classe qui est créée.
Ceci a pour conséquence qu'il ne suffit pas de recompiler pour modifier les relations d'héritage existant dans les objets déjà créés: ceux-ci conservent en effet une copie de leur classe parente d'origine qui n'est pas modifiée par un changement du programme.

[44] Pour ne pas donner prise à l'accusation de sexisme, nous laisserons le lecteur imaginer lesquels.

```
  else
        set the end of listeP to creaHomme(nnom, aage)
  end if
```

nous créons l'objet approprié.

Qu'en est-il de la procédure getInfos() ? Si nous ne modifions pas davantage le script, elle continue de fonctionner comme avant, ce qui est un des aspects importants de l'héritage. La procédure getInfos() continue d'exister et de fonctionner aussi bien pour les hommes que pour les femmes. Vous trouverez en annexe le script complet que vous pouvez essayer :

```
(*      Script de démo des scripts-objets
Etape 1 avec la méthode objet
*)

-- Stockage des données : une liste de scripts-objets.
-- Comme la property subsiste d'une fois sur l'autre je retrouverai ces données
property listeP : {}

----------------- le programme principal -------------
on run
  repeat
        choose from list {"1- Saisir une personne", "2- Voir les personnes"} ¬
              cancel button name "Quitter"
        if result = false then -- = bouton Quitter
              exit repeat
        else
              set rep to first word of item 1 of result
              if rep = "1" then
                    newPersonne()
              else if rep = "2" then
                    visuPersonne("Liste des personnes :")
              end if
        end if
  end repeat
end run

--------------- Création des fiches personnes ---------------

on newPersonne()
  display dialog ¬
        "Donner le nom de la personne" default answer ""
  set nnom to (text returned of result)
  display dialog ¬
        "Donner l'âge de la personne" default answer ""
  set aage to (text returned of result) as integer

  choose from list {"Femme", "Homme"}
  set rep to first item of result
  if rep = "Femme" then
        set the end of listeP to creaFemme(nnom, aage)
  else
        set the end of listeP to creaHomme(nnom, aage)
  end if
end newPersonne
```

```
on creaPersonne(sonNom, sonAge) -- la classe fondamentale
  script
        property nom : sonNom
        property age : sonAge

        on getInfos()
              return nom & " a " & age & " ans"
        end getInfos
  end script
end creaPersonne

on creaHomme(sonNom, sonAge) -- les hommes
  copy creaPersonne(sonNom, sonAge) to zzz -- créer l'objet parent
  script
        property parent : zzz

  end script
end creaHomme

on creaFemme(sonNom, sonAge) -- les hommes
  copy creaPersonne(sonNom, sonAge) to zzz -- créer l'objet parent
  script
        property parent : zzz

  end script
end creaFemme


------------------- Visualisation --------------

on visuPersonne(aPrompt)
  -- 1- retour en cas de liste vide
  if listeP = {} then
        display dialog "Il n'y a aucune personne saisie !"
        return {}
  end if

  -- 2- constitution de la liste des informations des personnes
  set listeInfos to {}
  repeat with i from 1 to count listeP
        getInfos() of item i of listeP
        copy (i as string) & "- " & result to the end of listeInfos
  end repeat

  return choose from list listeInfos with prompt (aPrompt) ¬
        with multiple selections allowed and empty selection allowed

end visuPersonne
```

Remarquons déjà une chose importante : nous n'avons pas eu à modifier le code existant excepté la routine de création des objets. Nous avons simplement ajouté deux nouvelles classes. C'est très important, car nous sommes ainsi certains de ne pas introduire d'erreur dans ce qui existe et qui marche.

Maintenant il nous reste l'essentiel : réaliser un affichage distinct pour les hommes et les femmes y compris le calcul spécial sur l'âge dans ce cas.
De manière très naturelle, il suffit d'écrire deux handlers getInfos() différents,

un pour les hommes, un pour les femmes. On les associe directement aux classes intéressées en les plaçant dans leur bloc script. Voici donc les nouvelles définitions des classes Femme et Homme :

```
on creaHomme(sonNom, sonAge) -- les hommes
  copy creaPersonne(sonNom, sonAge) to zzz -- créer l'objet parent
  script
        property parent : zzz

        on getInfos()
              if my age > 21 then
                    set tmp to "un homme"
              else
                    set tmp to "un garçon"
              end if
              return my nom & ", " & tmp & ¬
                    " âgé de " & my age & " ans"
        end getInfos

  end script
end creaHomme

on creaFemme(sonNom, sonAge) -- les hommes
  copy creaPersonne(sonNom, sonAge) to zzz -- créer l'objet parent
  script
        property parent : zzz

        on getInfos() -- retourne une liste contenant l'âge et la nature de la personne
              if my age > 18 then
                    set tmp to "une femme"
              else
                    set tmp to "une jeune fille"
              end if

              if my age > 30 then -- le calcul de courtoisie
                    set x to (my age) div 5
              else
                    set x to 0
              end if

              return my nom & ", " & tmp & ¬
                    " âgée de " & ((my age) - x) & " ans"
        end getInfos

  end script
end creaFemme
```

Notons deux choses importantes : la première est qu'il est indispensable d'utiliser la syntaxe **my** Nom, **my** Age pour atteindre les properties de la classe parente. (Sinon, AppleScript cherche dans le script local : Homme ou Femme et sinon dans les variables et properties définies au premier niveau du script).

La deuxième est que le script fonctionne ainsi, sans aucune modification de la routine visuInfos(). Le même appel :

```
getInfos() of the item x of listeP
```

va donner un résultat différent selon que l'objet visé appartient à la classe Homme ou Femme. Le handler associé à cette classe sera automatiquement exécuté. Il est donc inutile de prévoir dans visuInfos() le test permettant de savoir si la fiche considérée représente une femme ou un homme, car AppleScript l'effectue automatiquement.

Il y a plus : nous avons placé un handler getInfos() dans chacune des classes Femme et Homme, mais vous pouvez si vous voulez supprimer purement et simplement celui des hommes. Cela ne sera pas conforme aux nouvelles spécifications, mais il n'y aura pas d'erreur d'exécution : AppleScript, ne trouvant pas de handler getInfos() dans la classe Homme, ira chercher dans la classe dont il hérite un handler de même nom, et puisqu'il existe, va l'utiliser.

Ce choix automatique du handler approprié pendant l'exécution est ce qu'on appelle le polymorphisme. On voit maintenant le principe : on écrit un handler général dans la classe la plus générale, et s'il y a lieu, on le surcharge par d'autres versions (portant le même nom et les mêmes paramètres) dans les classes spécialisées, pour tenir compte de leurs particularités. Le langage se charge de choisir le handler approprié : sa présence dans le script définissant une classe suffit.

Il s'ensuit que l'on peut préciser un fonctionnement simplement en créant des sous-classes et leurs handlers, sans modifier le programme principal.

Vous trouverez en annexe le script complet.

<u>Exemple parallèle</u>

Et par comparaison, voici le traitement classique. Des différences plus nettes commencent à apparaître.

D'une part, on a ajouté un champ désignant le sexe dans le record.

D'autre part, on a réparti l'élaboration des infos sur la personne en deux routines séparées getGenre() et makiAge() qui toutes les deux comportent une série de tests destinés à déterminer à quel type de personne nous avons affaire afin de lui appliquer le traitement approprié.

Cette série de tests est absente en programmation objet, car elle est prise en charge de manière cachée par le langage. C'est un allégement de travail qui va devenir de plus en plus important.

```
(*	Script de démo des scripts-objets
Etape 2 avec la méthode traditionnelle
*)

-- Stockage des données : une liste de records.
-- Comme la property subsiste d'une fois sur l'autre je retrouverai ces données
property listeP : {}

---------------- le programme principal -------------
...
--------------- Création des fiches personnes ---------------

on newPersonne()
  display dialog ¬
```

```
        "Donner le nom de la personne" default answer ""
  set nnom to (text returned of result)
  display dialog ¬
        "Donner l'âge de la personne" default answer ""
  set aage to (text returned of result) as integer

  choose from list {"Femme", "Homme"}
  set rep to first item of result
  if rep = "Femme" then
        set asexe to "F"
  else if rep = "Homme" then
        set asexe to "H"
  end if

  copy creaPersonne(nnom, aage, asexe) to the end of listeP

end newPersonne

on creaPersonne(sonNom, sonAge, sonSexe) -- création d'un record
  return {nom:sonNom, age:sonAge, sexe:sonSexe}
end creaPersonne

--------------- Les informations ---------------

on getInfos(pers)
  return ¬
        nom of pers & ", " & getGenre(sexe of pers, age of pers) & ¬
        " âgé de " & makiAge(sexe of pers, age of pers) & " ans"
end getInfos

on getGenre(sonSexe, sonAge) --le genre de la personne
  if sonSexe is "H" then
        if sonAge > 21 then
              return "un homme"
        else
              return "un garçon"
        end if
  else if sonSexe is "F" then
        if sonAge > 18 then
              return "une femme"
        else
              return "une jeune fille"
        end if
  end if
end getGenre

on makiAge(sonSexe, sonAge) -- le calcul de courtoisie
  if sonSexe is "F" and sonAge > 30 then
        return sonAge - sonAge div 5
  else
        return sonAge
  end if
end makiAge
```

Le script complet se trouve en annexe.

Troisième étape.

Spécifications

Nous voudrions introduire la gestion des couples. Donc ajouter une option qui permette d'associer deux personnes de la base en tant que conjoints et une autre qui nous permette d'afficher les couples les uns après les autres.
Il faut naturellement vérifier que les personnes sont de sexe différent, ont l'âge requis (15 et 18 ans) et ne sont pas déjà mariées.

Organisation des données :

Il n'y a pas de nouvelle notion à introduire à ce stade. Nous allons ajouter quelques handlers aux classes existantes, ce qui nous permettra d'explorer la méthode plus en détail.
Pour repérer le conjoint, nous aurons besoin d'une nouvelle propriété. Pour des raisons de simplicité, nous avons choisi d'utiliser le rang de la fiche du conjoint dans la liste. Autrement dit, si Marie occupe la cinquième place dans la liste, la propriété conj dans la fiche de son conjoint Albert aura la valeur 5[45] .

Comme cette propriété est commune aux deux sexes, il est plus normal de la faire figurer dans la classe Personne :

```
property conj : 0
```

L'initialisation de la property à 0 (qui n'est pas un rang possible dans une liste), permet de signaler qu'une personne est célibataire.

Maintenant, pour enregistrer le mariage d'une personne, nous allons :
1- constituer une liste des personnes qu'il peut épouser.
2- associer les conjoints choisis.

Pour la constitution de la liste, nous devons sélectionner les personnes :
1- du sexe opposé,
2- qui ont l'âge requis
3- qui ne sont pas mariées.

Certains de ces tests dépendent du sexe, et d'autres non. Nous allons écrire un handler Convoler() qui se répartira entre les différentes classes de la manière suivante.
Pour les femmes, constituer une liste d'hommes d'âge supérieur à 18 ans
Pour les hommes, constituer une liste de femmes d'âge supérieur à 15 ans
Pour les deux : ils ne doivent pas être mariés.

Le test complet sera implémenté par un handler Mariable(sexe_souhaité) que

---

[45] Notons que cette disposition empêche les tris ultérieurs de la liste, et c'est pourquoi elle n'est pas satisfaisante. Dans un développement réel, on définirait une clef unique pour chaque fiche.

voici pour les femmes :

```
on Mariable(sex)
    if sex ≠ "F" then
        return false
    end if

    if my age > 15 then
        return continue Mariable()
    else
        return false
    end if
end Mariable
```

et pour les hommes :

```
on Mariable(sex)
    if sex ≠ "H" then
        return false
    end if

    if my age > 18 then
        return continue Mariable()
    else
        return false
    end if
end Mariable
```

et la partie commune aux deux :

```
on Mariable()
  return (my conj = 0) -- évite la bigamie !!!
end Mariable
```

Le point à noter est l'instruction :

```
continue Mariable()
```

Celle-ci a pour effet de transférer l'exécution au handler Mariable() situé dans la classe parent. Donc la partie commune aux deux handlers pourra ainsi être exécutée.

Pour exécuter ce handler nous aurions pu aussi nous servir du terme d'appartenance :

```
Mariable() of parent
```

ou

```
tell parent to Mariable()
```

Ceci permet de tester le célibat de manière unique à la fois pour les hommes et pour les femmes, autrement dit, l'appel de Mariable() a une première partie spécifique au sexe de la personne qui se poursuit par une partie commune.

Ce handler Mariable() permet également de s'assurer que la personne choisie pour convoler est également en âge de le faire et n'est pas elle-même mariée.

Voici donc le handler Convoler() pour une femme :

```
on Convoler(moi)
  -- vérifier l'âge du postulant
  if my Mariable("H") then
        -- créer une liste de conjoints possibles
        set liste to {}
        repeat with i from 1 to count listeP
              set pers to item i of listeP
              if (Mariable("F") of pers) then
                    copy ((i as string) & "-" & nom of pers) to the end of liste
              end if
        end repeat
        continue Convoler(liste, moi)
  else -- la personne est trop jeune ou déjà marié
        display dialog (my nom & "  est trop jeune ou déjà marié.")
  end if
end Convoler
```

Le premier appel Mariable("H") permet simplement de vérifier que la femme est suffisamment âgée. Ensuite on parcourt la liste des personnes pour vérifier s'il s'agit d'un conjoint possible : un homme de plus de dix huit ans.

Une fois cette liste constituée, commence une partie commune aux hommes et aux femmes : choisir un conjoint dans la liste et réaliser les mises à jour nécessaires de leurs fiches. Ce travail est effectué dans le handler Convoler(liste,moi) de la classe Personne :

```
on Convoler(liste, moi)
  if liste ≠ {} then
        choose from list liste with prompt "Choisissez un conjoint."
        set x to (first word of item 1 of result) as integer
        set my conj to x
        set conj of (item x of listeP) to moi
  end if
end Convoler
```

Le travail est pratiquement terminé : il reste à implémenter l'interface des deux options et la routine d'affichage des mariages partant du même principe que getInfos(). Comme elle ne dépend pas du sexe, on la place dans la classe Personne qui devient :

```
on creaPersonne(sonNom, sonAge) -- la classe fondamentale
  script
        property nom : sonNom
        property age : sonAge
        property conj : 0

        on getInfos()
              return nom & " a " & age & " ans"
        end getInfos

        on getConjoint() -- retourne l'info mariage
              if conj = 0 then -- pas marié
                    return " est célibataire."
              else
```

```
                return " a épousé " & (nom of item (my conj) of listeP) & "."
            end if
        end getConjoint

        on Mariable()
            return (my conj = 0) -- évite la bigamie !!!
        end Mariable

        on Convoler(liste, moi)
            if liste ≠ {} then
                choose from list liste with prompt "Choisissez un conjoint."
                set x to (first word of item 1 of result) as integer
                set my conj to x
                set conj of (item x of listeP) to moi
            end if
        end Convoler
    end script
end creaPersonne
```

Voir le script complet en annexe. Il commence à devenir relativement complexe.

Exemple parallèle.

Voyons maintenant notre exemple parallèle.

Son évolution consiste essentiellement en l'ajout de trois procédures à la fin. On peut faire la même observation que plus haut : dans Mariable() qui a la même fonction que dans la version objet, il devient nécessaire de mettre en place une batterie de tests imposante et complexe pour tenir compte de tous les cas de figure.

```
on Mariable(pers1, pers2)
  set {sonAge1, sonSexe1, sonConjoint1} to {age, sexe, conj} of pers1
  set {sonAge2, sonSexe2, sonConjoint2} to {age, sexe, conj} of pers2
  --conditions
  set b to true
  set b to b and (pers1 ≠ pers2) -- les 2 records sont différents
  set b to b and (sonConjoint1 = 0) --pers1  est célibataire
  set b to b and (sonConjoint2 = 0) --pers2  est célibataire
  set b to b and (sonSexe1 ≠ sonSexe2) -- les sexes sont différents
  set b to b and isAgeMari(sonSexe1, sonAge1) --pers1 est en âge de se marier
  set b to b and isAgeMari(sonSexe2, sonAge2) --pers2 est en âge de se marier
  return b
end Mariable

on isAgeMari(sonSexe, sonAge)
  if sonSexe is "F" then
        return sonAge ≥ 18
  else if sonSexe is "H" then
        return sonAge ≥ 15
  end if
end isAgeMari
```

Voir en annexe le script complet.

Quatrième étape.

Spécifications.

Les martiens arrivent[46] . Ils ont un sexe de plus que les nôtres (appelé neutre). De plus l'âge qu'ils déclarent (donc que l'on saisit) est compté en années martiennes et on veut l'affichage en années terrestres (une année martienne = 3 années terrestres). La règle de courtoisie ne s'applique pas aux femmes martiennes.

Naturellement, les mariages sont des triades constituées de trois personnes de sexes différents (pas de limite d'âge). Les mariages interraciaux sont interdits.

Réalisation.

Nous n'avons pas de nouvelle fonction à ajouter : seulement une nouvelle catégorie de personnes imprévues. Donc normalement, nous ne devrions avoir qu'à définir de nouvelles classes pour les martiens et modifier la routine de saisie des personnes. Celle-ci devient :

```
--------------- Création des fiches personnes ---------------

on newPersonne()
  display dialog ¬
          "Donner le nom de la personne" default answer ""
  set nnom to (text returned of result)
  display dialog ¬
          "Donner l'âge de la personne" default answer ""
  set aage to (text returned of result) as integer

  choose from list {"Femme", "Homme"} & {"Martien", "Martienne", "Martien neutre"}
  set rep to first item of result
  if rep = "Femme" then
          set the end of listeP to creaFemme(nnom, aage)
  else if rep = "Homme" then
          set the end of listeP to creaHomme(nnom, aage)
  else if rep = "Martien" then
          set the end of listeP to creaMartien(nnom, aage)
  else if rep = "Martienne" then
          set the end of listeP to creaMartienne(nnom, aage)
  else if rep = "Martien neutre" then
          set the end of listeP to creaMartienNeutre(nnom, aage)
  end if
end newPersonne
```

Ajoutons les classes pour les martiens. Ces classes seront bâties sur le même modèle que les autres, puisqu'elles devront répondre aux mêmes appels.

La classe martienne générale hérite de la classe Personne. On lui ajoute une

---

[46] Il s'agit de simuler une situation commune: celle d'un événement absolument imprévisible, et donc imprévu au début du développement. On va voir que la méthode objet se tire de ce genre de mauvais pas avec élégance.

propriété supplémentaire qui est l'index du deuxième conjoint. Par ailleurs les trois handlers de base sont modifiés de manière à tenir compte du troisième conjoint.

```
on creaMars(sonNom, sonAge) -- les personnes martiennes en général
  copy creaPersonne(sonNom, sonAge) to zzz -- créer l'objet parent
  script
        property parent : zzz
        property conj2 : 0 -- nous avons besoin d'un second conjoint: le neutre

        on getConjoint() -- retourne l'info mariage
              if my conj2 = 0 then -- pas de deuxième conjoint
                    continue getConjoint() -- on voit s'il en a un comme pour les humains
              else
                    if my conj ≠ 0 then
                          return " a épousé " & (nom of item (my conj) of listeP) & ¬
                                " et " & (nom of item (my conj2) of listeP)
                    else
                          return " est célibataire."
                    end if
              end if
        end getConjoint

        on Mariable()
              return (my conj = 0) and (my conj2 = 0) -- évite la bigamie !!!
        end Mariable

        on Convoler(liste1, liste2, moi)
              if (liste1 ≠ {}) and (liste2 ≠ {}) then
                    choose from list liste1 with prompt "Choisissez un premier conjoint."
                    set x to (first word of item 1 of result) as integer
                    set my conj2 to x
                    set conj2 of (item x of listeP) to moi

                    choose from list liste2 with prompt "Choisissez un deuxième conjoint."
                    set y to (first word of item 1 of result) as integer
                    set my conj to y
                    set conj of (item y of listeP) to moi

                    set conj of (item x of listeP) to y
                    set conj2 of (item y of listeP) to x
              end if
        end Convoler

  end script
end creaMars
```

Trois nouveaux sexes sont définis pour pouvoir utiliser le handler Mariable(). De la sorte, on peut mélanger martiens et humains dans la base sans risquer de créer des mariages intergalactiques.

Le handler Convoler() tient naturellement compte du fait qu'il faut deux conjoints au lieu d'un seul.

```
on creaMartien(sonNom, sonAge)
  copy creaMars(sonNom, sonAge) to zzz -- créer l'objet parent
  script
        property parent : zzz
```

```
        on getInfos() -- retourne une liste contenant l'âge et la nature de la personne
                return my nom & ", " & "un martien " & ¬
                        " âgé de " & ((my age) * 3) & " ans" & my getConjoint()
        end getInfos

        on Mariable(sex)
                if sex ≠ "mm" then
                        return false
                end if
                return continue Mariable()
        end Mariable

        on Convoler(moi)
                -- créer une liste de conjoints possibles
                set liste1 to {}
                repeat with i from 1 to count listeP
                        set pers to item i of listeP
                        if (Mariable("mn") of pers) then
                                copy (i as string) & "-" & nom of pers to the end of liste1
                        end if
                end repeat

                set liste2 to {}
                repeat with i from 1 to count listeP
                        set pers to item i of listeP
                        if (Mariable("mf") of pers) then
                                copy (i as string) & "-" & nom of pers to the end of liste2
                        end if
                end repeat

                continue Convoler(liste1, liste2, moi)
        end Convoler

  end script
end creaMartien
```

Les deux autres classes peuvent être facilement construites par copier/coller de cette dernière. Elles diffèrent simplement par le message renvoyé par getInfos() et les sexes passés en paramètres de la fonction Mariable().

Voir le script final en annexe.

En dehors de la création de ces classes, nous n'avons eu à modifier aucune partie du programme. Voyons maintenant comment une approche classique se tire de cette affaire.

Exemple parallèle.

La remarque essentielle est que l'arrivée des Martiens a conduit à modifier presque tout le programme : partout il a fallu prévoir des tests supplémentaires pour en tenir compte. Le risque d'introduire des erreurs dans ce qui fonctionnait déjà n'est donc pas négligeable.

On peut également constater que ces batteries de tests commencent à devenir touffues et difficiles à comprendre, mais surtout qu'elles se répètent presque à l'identique dans toutes les fonctions. Elles sont finalement bâties sur le même squelette répétitif qui consiste à classer les personnes dans les diverses catégories. L'utilisation des scripts-objets permet de simplifier tout cela.

```
on Convoler(x)
  set pers1 to item x of listeP
  set liste to {}
  repeat with i from 1 to count listeP
        set pers to item i of listeP
        if Mariable(pers1, pers) then
              copy (i as string) & "-" & nom of pers to the end of liste
        end if
  end repeat

  if liste = {} then return

  if sexe of pers1 is in {"H", "F"} then
        choose from list liste with prompt "Choisissez un conjoint."
        set y to (first word of item 1 of result) as integer
        set pers2 to item y of listeP
        set conj of pers1 to y
        set conj of pers2 to x
  else if sexe of pers1 is in {"mm", "mf", "mn"} then
        repeat
              set rep to choose from list liste with prompt ¬
                    "Choisissez 2 conjoints de sexes différents." with multiple selections allowed
              if (count rep) = 2 then
                    set y to (first word of item 1 of rep) as integer
                    set z to (first word of item 2 of rep) as integer
                    set pers2 to item y of listeP
                    set pers3 to item z of listeP
                    if (sexe of pers2) ≠ (sexe of pers3) then
                          -- conjoints de pers1
                          set conj of pers1 to y
                          set conj2 of pers1 to z
                          -- conjoints de pers2
                          set conj of pers2 to x
                          set conj2 of pers2 to z
                          -- conjoints de pers3
                          set conj of pers3 to x
                          set conj2 of pers3 to y
                          exit repeat
                    end if
                    beep
              end if
        end repeat
  end if
end Convoler

on Mariable(pers1, pers2)
  set {sonAge1, sonSexe1, sonConjoint1} to {age, sexe, conj} of pers1
  set {sonAge2, sonSexe2, sonConjoint2} to {age, sexe, conj} of pers2
  --conditions
  set b to true
  set b to b and (pers1 ≠ pers2) -- les 2 records sont différents
```

```
  set b to b and (sonConjoint1 = 0) --pers1  est célibataire
  set b to b and (sonConjoint2 = 0) --pers2  est célibataire
  set b to b and (sonSexe1 ≠ sonSexe2) -- les sexes sont différents
  set b to b and isAgeMari(sonSexe1, sonAge1) --pers1 est en âge de se marier
  set b to b and isAgeMari(sonSexe2, sonAge2) --pers2 est en âge de se marier
  if sonSexe1 is in {"H", "F"} then  -- pers1 et pers2 sont de la même planète
        set b to b and sonSexe2 is in {"H", "F"}
  else if sonSexe1 is in {"mm", "mf", "mn"} then
        set b to b and sonSexe2 is in {"mm", "mf", "mn"}
  end if
  return b
end Mariable
```

Notes.

L'exemple que nous venons de traiter appelle quelques remarques.

Remarque -1

En premier lieu, la hiérarchie proposée des différentes classes peut se discuter : en particulier, on peut se demander s'il est opportun de faire dériver les classes de martiens de la classe Personne. C'est vrai et cela n'a guère d'importance : on peut avoir autant de classes fondamentales que l'on veut.

Plus fondamentalement, on peut se demander s'il n'aurait pas mieux valu définir des sous-classes “Femmes de plus de trente ans”, “Femmes de moins de trente ans” et “Jeune fille” pour traiter le problème des âges. C'est effectivement une approche tout à fait valable. Naturellement un grand nombre de classes rend aussi les programmes plus complexes, si bien qu'on ne peut pas donner de règle absolue.

D'une façon plus générale, nous avons cherché à montrer comment fonctionne l'héritage de la manière la plus claire et plus complète possible. On aurait naturellement pu éviter de faire appel aux handlers de la classe Personne avec l'instruction **continue**, ce qui aurait plutôt simplifié les choses.

Mais on voit que la répartition des différentes fonctions et attributs entre les classes est faite en fonction des caractéristiques réelles des objets et non en fonction de raisons d'opportunité informatique. C'est une règle très importante à suivre si l'on veut profiter d'un des avantages les plus importants de la méthode objet, rendre les logiciels plus faciles à faire évoluer.

Remarque - 2

Nous avons soigneusement évité de mettre dans les classes filles des propriétés portant le même nom qu'une propriété des classes parentes. La chose est cependant possible et le résultat est la création de deux propriétés différentes portant le même nom, toutes les deux accessibles dans les classes filles. Le sujet étant largement traité dans le guide AppleScript, nous ne l'avons pas traité du tout.

Dans notre idée, c'est une pratique à proscrire autant que possible car elle pose de gros problèmes (à quoi accède-t-on ? on n'en est jamais sûr) et qu'elle n'apporte que bien rarement des avantages intéressants.

Remarque - 3

Dans les exemples de polymorphismes donnés ci-dessus, en particulier pour la méthode getInfos(), on pourra remarquer que la méthode originale, placée dans la classe Personne, est parfois inutile puisqu'elle est masquée dans toutes les classes filles par d'autres méthodes spécifiques. On peut effectivement la supprimer dans ces cas là, ce que nous n'avons pas fait, d'abord pour ne pas alourdir l'exposé, ensuite parce qu'il est souvent utile d'avoir une méthode par défaut pour le cas où l'on dériverait de nouvelles classes qui pourraient l'utiliser.

Remarque - 4

Dans la réalisation des méthodes Convoler(), on aura peut-être observé que cette méthode n'est pas identique dans la classe Personne (elle prend deux paramètres), dans la classe martien (trois paramètres) et dans les classes dérivées (elle n'en prend pas du tout). En toute rigueur, malgré l'identité des noms, il ne s'agit pas de la même méthode. Si l'on définit une nouvelle classe dérivée de Personne qui ne comporte pas de handler Convoler(), et qu'on appelle cette méthode sans paramètre, il y aura naturellement erreur, pas parce que la méthode manque, mais parce qu'elle n'a pas le bon nombre de paramètres.

Il serait peut-être de meilleure pratique dans ces cas-là de distinguer par des noms différents les trois procédures Convoler() selon le nombre de paramètres qu'elle prennent.

Compléments.

Accès externe.

Puisque la liste des scripts-objets définie ci-dessus est une propriété d'une application, on peut se demander s'il est possible d'y accéder depuis un autre script.

La chose est en effet possible sous diverses conditions. Si le script d'exemple a été sauvé en tant qu'application nommée “BaseNoms”, on peut en effet écrire ceci :

```
tell application "BaseNoms"
  launch
  set laListe to the listeP of application "BaseNoms"
end tell

display dialog getInfos() of the first item of laListe
```

ou encore pour accéder directement à l'item 1.

```
tell application "BaseNoms"
  launch
  set pers to the first item of listeP of application "BaseNoms"
end tell

display dialog getInfos() of pers
```

Deux remarques sont à faire :

- d'abord que l'objet-script est importé avec toute sa structure et en particulier ses méthodes. Même en dehors du bloc tell application “BaseNoms”, la méthode getInfos() est utilisable, car elle fait partie de l'objet-script et non de l'application. Si elle était écrite en tant que handler de l'application elle-même, cela serait l'inverse.

Il s'ensuit que l'objet-script récupéré par cet accès est entièrement autonome et autosuffisant : si vous le gardez dans une globale ou que vous le sauvez sur le disque, il continuera à fonctionner, même si l'application BaseNoms a disparu. Il n'en dépend absolument pas.

- qu'on ne peut accéder directement à listeP en supprimant le bloc tell et le launch. La raison en est que si on oublie cette précaution, l'application BaseNoms est lancée, autrement dit, son handler run s'exécute et elle ne répond rien. Il faut donc la charger sans lancer d'exécution ce qui est l'objectif du launch.

Load/Store script.

Nous avons vu fonctionner l'accès dans une application qui contient les objets visés. Mais il existe une autre possibilité. Nous pouvons sauver un objet dans un fichier par la commande store script (compléments standard).

```
tell application "BaseNoms"
  launch
  set pers to the first item of listeP of application "BaseNoms"
end tell

display dialog getInfos() of pers

  set tmp to ((path to desktop) as string) & "test"
  store script pers in file tmp
```

L'objet contenu dans la variable pers est sauvé dans un fichier appelé test. Ce fichier est en fait un script compilé, et vous pouvez l'ouvrir avec l'éditeur de script. Ce qui donne le résultat suivant :

```
property parent : zzz

on getInfos() -- retourne une liste contenant l'âge et la nature de la personne
  if my age > 18 then
        set tmp to "une femme"
  else
        set tmp to "une jeune fille"
  end if
```

```
	if my age > 30 then -- le calcul de courtoisie
		set x to (my age) div 5
	else
		set x to 0
	end if

	return my nom & ", " & tmp & ¬
		" âgée de " & ((my age) - x) & " ans" & my getConjoint()
end getInfos

on Mariable(sex)
	if sex ≠ "F" then
		return false
	end if

	if my age > 15 then
		return continue Mariable()
	else
		return false
	end if
end Mariable

on Convoler(moi)
	-- vérifier l'âge du postulant
	if my Mariable("F") then
		-- créer une liste de conjoints possibles
		set liste to {}
		repeat with i from 1 to count listeP
			set pers to item i of listeP
			if (Mariable("H") of pers) then
				copy (i as string) & "-" & nom of pers to the end of liste
			end if
		end repeat
		continue Convoler(liste, moi)
	else -- la personne est trop jeune ou déjà mariée
		display dialog (my nom & "  est trop jeune ou déjà mariée.")
	end if
end Convoler
```

on reconnaît sans peine la définition de la classe Femme. La classe parente (Personne) n'est pas visible, mais elle est présente ainsi que les propriétés, comme on peut s'en assurer en effectuant l'opération à l'envers :

```
set tmp to ((path to desktop) as string) & "test"
	set pers to load script file tmp

	display dialog getInfos() of pers
```

Qui donne bien le résultat attendu indépendamment de l'application BaseNoms.

On peut ainsi envisager par exemple de sauver les données de notre application BaseNoms en autant de fichiers scripts que de personnes avec le script suivant :

```
	tell application "BaseNoms"
```

```
  launch
  set laListe to the listeP of application "BaseNoms"
end tell

repeat with i in laListe
  set tmp to the Nom of i & ".scr"
  store script i in file tmp
end repeat
```

Ceci permet de développer des applications à base d'objets distribués un peu comme les Java beans.

Spécialiser des objets existants.

Une autre possibilité intéressante consiste à créer des scripts-objets qui héritent des objets définis dans les dictionnaires des applications. L'intérêt est de pouvoir personnaliser des objets existants sans perdre les propriétés qu'ils possèdent déjà.

Supposons par exemple que nous souhaitions adjoindre à notre base de Personnes des fichiers images contenant leurs photos. Ceci peut se faire en déclarant que la classe Personne hérite de la classe "file" du Finder.

```
on creaPersonne(sonNom, sonAge, thePath)

  script
        property parent : file thePath of application "Finder"
        property nom : sonNom
        property age : sonAge
        property conj : 0

        on voirImage()
              open my parent
        end voirImage

  end script

end creaPersonne
```

(Nous n'avons pas reporté les autres méthodes de la classe Personne).

Si nous créons un tel objet dans la variable "test", l'instruction :

```
get the name of test
```

nous donnera le nom du fichier image au sens du Finder (le tell n'est pas nécessaire ici), car les objets de classe Personne sont des objets de classe file par héritage.

Ce dispositif comporte certaines limitations, mais il est applicable à toute classe d'objet définie dans un dictionnaire.

AS est-il un langage orienté objet ?

Introduction :

Cette partie assez technique ne prétend pas donner une réponse définitive ou instaurer une polémique stérile, mais exposer et clarifier un certain nombre de points du fonctionnement d'AppleScript. En particulier, il ne s'agit pas de faire le "procès" d'AppleScript et de ses concepteurs. Il est évident que ce langage devant rester accessible à des utilisateurs non spécialistes, il fallait éviter que ces utilisateurs novices ne soient confrontés à des notions qui déroutent même certains professionnels.

D'autre part, certains mécanismes d'AppleScript qui sont en dehors du champ du modèle objet ont apporté leur contraintes propres.

Les concepteurs ont donc fait le maximum pour réaliser un langage ayant un comportement aussi proche que possible d'un langage orienté objet au sens strict du terme, mais il reste un certain nombre d'étrangetés et de divergences qu'il était sans doute impossible d'éliminer.

Il n'est certainement pas indispensable de lire ceci pour appliquer les techniques exposées au chapitre précédent, mais le scripteur pourra y trouver une clarification des notions qu'il utilise, même en dehors de toute programmation objet, comme l'instruction **tell** et les mots clefs "my" et "it".

Les lecteurs allergiques aux considérations générales pourront sans doute sauter la première section pour passer directement au chaînage de recherche.

Conformité au modèle objet.

Introduction.

La première difficulté est de s'entendre sur ce qu'est le modèle objet. Ce n'est pas la comparaison avec des langages orientés objet comme C++ ou Java qui va nous guider, car le modèle objet est indépendant des langages de programmation.

Nous avons choisi de suivre la description que James Rumbaugh en donne au début de son livre fondamental sur la méthode OMT[47] ancêtre directe d'UML qui est à ce jour pratiquement un standard en conception orientée objet. L'auteur y explique qu'un consensus général s'est réalisé autour des points suivant :

Un objet est une entité discrète (c'est à dire aux limites parfaitement claires) constitués d'une structure de données ou état, et d'un comportement. Il s'agit en

[47] J.RUMBAUGH & alt: OMT, modélisation et conception orientées objet, Masson-Prentice Hall, ISBN: 2-225-84684-7

effet de construire une représentation simplifiée des objets réels en ne retenant que les traits pertinents pour l'application.

La structure de données dont il s'agit est l'ensemble des variables et groupements de variables destinées à recevoir les valeurs qui décrivent l'objet ou son état. Par exemple, on peut définir la couleur d'une voiture qui pourra prendre la valeur "rouge". On parle alors d'attributs.

Pour préciser ce que l'on entend par structure de données discrète, donnons l'exemple suivant. Si nous voulons représenter informatiquement les voitures à vendre dans un garage, nous pourrions utiliser diverses listes comme suit :

```
property modeles : {"Xantia", "C5", "Laguna"}
property couleurs : {"verte", "gris métallisé", "rouge"}
```

Plus éventuellement d'autres listes d'attributs comme le numéro minéralogique. Cette représentation des voitures, sans être incorrecte, n'est pas conforme au modèle objet, car les différents attributs sont dispersés dans des structures de données différentes. L'arrangement suivant par contre l'est :

```
property voitures : {¬
{modele:"Xantia",couleur :"verte"},¬
{modele:"C5",couleur :"gris métallisé"},¬
{modele:"Laguna",couleur :"rouge"}¬
}
```

Dans cette liste, tous les attributs de chaque voiture sont regroupés dans un record.

Le comportement est la réponse d'un objet à diverses opérations. que l'on représente souvent par un verbe : "Démarrer", "Accélérer" sont des opérations pour une voiture.

Ceci posé, voyons les propriétés requises pour les objets et leurs éléments, les attributs et les opérations.

Identité.

L'objet possède une identité propre qui n'est pas simplement caractérisée par ses attributs et son comportement. En d'autres termes, deux objets ayant exactement le même comportement et les mêmes attributs sont des clones que l'on peut distinguer. L'une des conséquences est que l'égalité (ou plus précisément identité) entre objets est seulement vérifiée lorsque l'on compare l'objet avec lui même.

En AppleScript, cette propriété n'est pas vérifiée pour tous les objets. Par exemple, le script suivant :

```
copy {1, 2} to test1
copy {1, 2} to test2
test1 = test2
--> true
```

Or les deux listes contenues dans test1 et test2 sont réellement distinctes, même si leurs contenus sont identiques. Par contre le script suivant montre que le principe d'identité est respecté pour les scripts-objets :

```
on creaPersonne(sonNom)
  script
          property nom : sonNom
  end script
end creaPersonne

set test1 to creaPersonne("Michel")

set Choix to 1  -- choisir la définition de test2

if Choix = 1 then
  set test2 to creaPersonne("Michel")  -- créé comme test1
  test1 = test2  --> false
else if Choix = 2 then
  copy test1 to test2  -- clonage par copy
  test1 = test2  --> false
else if Choix = 3 then
  set test2 to test1  -- référence par set
  test1 = test2  --> true
end if
```

En mettant la variable Choix à 1,2 ou 3 on obtient les résultats signalés en commentaires. Le choix 1 crée deux objets identiques en appelant deux fois le constructeur avec le même paramètre. Le choix 2 montre que le comportement est cohérent avec un clonage pur et simple par copy. Le troisième illustre le fonctionnement du set : test1 et test2 ne contiennent plus des copies mais des références au même objet, l'égalité ou plutôt l'identité est alors vérifiée.

Classification.

Les objets ayant les mêmes comportements et les mêmes attributs (mais des valeurs d'attributs qui peuvent être différentes) sont regroupés en collections ou ensembles que l'on appelle des classes. En quelque sorte, la classe est une description générale des objets de même espèce (au sens informatique du terme). Chaque objet appartenant à une classe est dit instance de cette classe.

En pratique, cela signifie que l'on trouve dans le code de l'application une description des objets appartenant à la classe, un peu comme un modèle ou un moule. Ce modèle ne fait pas partie des données : au début du programme, il n'existe aucun objet dans les données. C'est l'un des constructeurs de la classe qui va créer des objets réellement présents en mémoire et utilisables par l'application. Cette opération s'appelle instanciation. En principe, il faut toujours créer une instance de la classe pour pouvoir utiliser ses méthodes.

Dernière chose : tous les objets contiennent une référence implicite à leur classe. Ils "savent" de quelle classe ils sont.

En AppleScript, il est difficile de dire si le concept de classe existe vraiment. Le

terme est utilisé dans tous les dictionnaires et il existe une instruction "class" qui permet de récupérer la classe d'un objet. Mais observons le script suivant :

```
on creaPersonne(sonNom)
  script
        property nom : sonNom
  end script
end creaPersonne

set test2 to {1, 2}
log the class of test2
set test1 to creaPersonne("Michel")
log the class of test1

1--> list
2--> script
```

Le point à remarquer est que class renvoie toujours "script" quelque soit l'objet-script considéré. Pour les autres objets (construits dans le langage comme la liste ou provenant d'un dictionnaire), on obtient bien un nom de classe spécifique.

Ce qui veut dire que l'utilisateur AppleScript ne peut pas définir de classe et que les scripts-objets contreviennent au principe de classification : en effet, ces scripts-objets n'ont pas tous les mêmes attributs ni les mêmes comportements.

Pourtant, cela ne veut pas dire qu'AppleScript ignore absolument tout ce qui dérive de la notion de classe. Il simule un comportement analogue et on peut probablement dire qu'une construction comme :

```
on creaPersonne(sonNom)
  script
        property nom : sonNom
  end script
end creaPersonne
```

définit une classe anonyme. En tous cas, les blocs script sont très proches des descriptions de classe que l'on trouve en C++ ou en Java et remplissent les mêmes objectifs : fournir un modèle pour la création des objets.

Polymorphisme.

Le polymorphisme est la propriété qu'ont les objets de pouvoir répondre par une méthode appropriée à une opération. Une opération comme "Déplacer" n'aura pas le même effet sur une fenêtre d'ordinateur, une pièce d'échec ou un fonctionnaire dans une administration.

Le polymorphisme est donc un mécanisme qui fait que chaque objet est à même de répondre à une opération générale (comme "Déplacer") en mettant en œuvre une méthode appropriée à sa nature, définie dans l'objet lui-même (et non dans un jeu de procédures externes).

En AppleScript, le polymorphisme est pratiquement conforme au modèle objet théorique. Par exemple, ce qu'on appelle la **required suite** (présente dans tous les

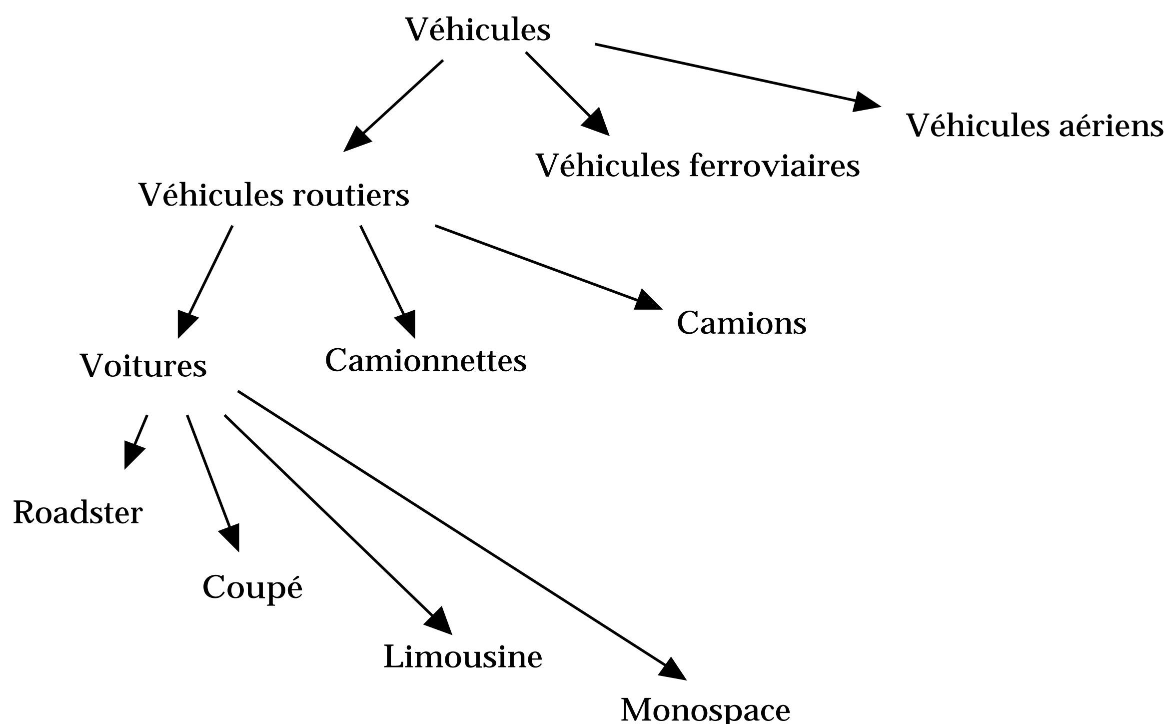
dictionnaires d'applications) est une liste d'opérations : chaque application la comprend et la met en œuvre selon sa méthode particulière qui se trouve dans l'application et non dans le langage ou dans les scripts.

L'exemple que nous avons développé au chapitre précédent montre en détail comment ce mécanisme fonctionne sur les scripts-objets. Nous avons par exemple implémenté une opération "Convoler" dont la méthode était différente pour les humains (qui se marient à deux) et les martiens (qui se marient à trois).

La réserve que l'on peut faire est en relation directe avec l'héritage, et nous la traiterons en même temps.

Héritage.

L'héritage est la propriété des classes de s'organiser hiérarchiquement. Si nous considérons par exemple une voiture de tourisme, nous pouvons dire qu'elle fait partie de la catégorie plus large des véhicules routiers qui elle même est contenue dans la catégorie véhicules. D'autre part, on peut distinguer plusieurs catégories de voitures de tourisme comme les limousines, les monospaces, les 4x4, etc…



C'est le principe bien connu de la classification en général[48] . Nous avons représenté la relation d'héritage sous forme de flèches. On dit que "Véhicules routiers" et "Véhicules" sont les super-classes de la classe "Voiture" dont "Roadster" et "Coupé" sont des sous-classes.

La relation d'héritage peut s'exprimer comme "est un(e)" [intitulé d'une des super-classes] : une voiture est un véhicule routier. Cette relation est transitive, autrement dit : si une voiture est un véhicule routier, alors une voiture est un véhicule car les véhicules routiers sont des véhicules.

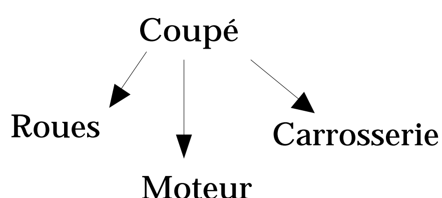
[48] Signalons en passant que ces classifications que l'on adopte sont arbitraires (le modèle objet ne propose pas de "bonne" méthode pour classifier). Cependant, je recommande fortement de leur donner les propriétés d'une taxinomie: exhaustivité et univocité. Autrement dit, tout objet doit trouver une place (exhaustivité) et une seule (univocité) dans la classification.

Comme nous l'avons vu, l'héritage a pour but d'organiser les attributs et les méthodes des objets. Tous les véhicules ont une vitesse maximum et fonctionnent avec un carburant : ces informations peuvent donc être placés comme attributs de la classe véhicule. Le mécanisme d'héritage permet d'implémenter pratiquement et simplement la présence dans tous les objets de toutes les sous-classes de "Véhicules" de ces attributs communs. Du moment que x est un véhicule, il aura une vitesse maximale et un carburant.

Il en est de même pour les opérations : "Démarrer" peut être associé à tout véhicule, mais on préférera peut-être réserver "Décoller" à la classe des "Véhicules aériens".

Donc on ne répète pas dans les sous-classes les attributs et méthodes définies dans les super classes sauf si une modification ou une précision est nécessaire. On ajoute en général dans les sous-classes des attributs et opérations supplémentaires.

On voit pourquoi l'extension de la hiérarchie des classes présentée ci-dessous est une erreur :



Une carrosserie n'est pas un véhicule et n'a ni carburant ni vitesse maximale. Il faut donc soigneusement éviter de confondre la hiérarchie des classes avec celle des composants d'un objet.

En pratique l'héritage fonctionne de la manière suivante : lorsqu'une opération est appliquée à un objet (par exemple "Démarrer"), le run time cherche une méthode applicable à l'opération, d'abord dans la classe de l'objet lui-même, puis en remontant la hiérarchie dans chacune des super-classes.

De la même manière, s'il a besoin d'un attribut, il cherchera dans les attributs définis dans la classe de l'objet, puis remontera la hiérarchie des super-classes jusqu'à trouver le premier qui fonctionne.

On voit comment ce mécanisme permet la surcharge ou le masquage des attributs et méthodes dans les sous-classes et nous l'avons utilisé dans notre exemple de démonstration. Mais fonctionne-t-il réellement ainsi en AppleScript ? Superficiellement ou sous certaines conditions oui, mais réellement non. Nous allons voir les différences.

Ce que l'on appelle héritage en AppleScript se déclare en incorporant aux objets une **property parent**. On peut déjà soupçonner tout de suite qu'il ne s'agit pas d'héritage au sens propre du terme : le **parent** est plutôt un composant de l'objet, pas l'indication d'une super classe. Autrement dit la hiérarchie des classes est mélangée avec celle des composants. Un certain nombre d'astuces permettent d'en minimiser les conséquences, mais cela n'arrange pas tout et il faut prendre un certain nombre de précautions si l'on veut "coller" au modèle objet.

Plus grave : le parent étant un composant, les attributs définis dans les super-classes de l'objet ne font pas partie de l'objet lui-même, mais des parents, ce qui contrevient au principe de l'individualité de l'objet (il ne peut y avoir d'attributs partagés entre plusieurs objets, chacun d'eux doit avoir tous les siens même si leur valeur est identique). On pourrait même imaginer de modifier la property parent une fois l'objet créé autrement dit bouleverser totalement la classification en cours d'exécution, mais heureusement, le langage ne permet pas de le faire.

Voyons un exemple basé sur notre développement précédent. Pour créer des objets Femme ou Homme, qui héritent d'objets Personne, nous avons programmé à peu près comme suit :

```
on creaPersonne(sonNom, sonAge)
  script
        property nom : sonNom
        property age : sonAge
  end script
end creaPersonne

on creaHomme(sonNom, sonAge)
  set zzz to creaPersonne(sonNom, sonAge)

  script
        property parent : zzz

        on getInfos()
              display dialog my nom & " est un homme de " & my age & " ans."
        end getInfos
  end script
end creaHomme

set test to creaHomme("Michel", 36)
```

Mais on pourrait parfaitement imaginer, au lieu de créer systématiquement un parent dans la variable locale zzz, d'utiliser toujours le même script Personne de la manière suivante :

```
script Personne
  property nom : "Jojo"

end script

on creaHomme(sonNom, sonAge)
  set the nom of Personne to sonNom
  script
        property parent : Personne
        property age : sonAge

        on getInfos()
              display dialog my nom & " est un homme de " & age & " ans."
        end getInfos
  end script
end creaHomme

on creaFemme(sonNom, sonAge)
```

```
  set the nom of Personne to sonNom
  script
      property parent : Personne
      property age : sonAge

      on getInfos()
          display dialog my nom & " est une femme de " & age & " ans."
      end getInfos
  end script
end creaFemme

set PList to {}
set tmp to creaHomme("Michel", 26)
--copy tmp to the end of PList
set PList to PList & tmp
set tmp to creaFemme("Corinne", 30)
--copy tmp to the end of PList
set PList to PList & tmp

repeat with pp in PList
  getInfos() of pp
end repeat

set the nom of Personne to "Jojo"

repeat with pp in PList
  getInfos() of pp
end repeat
```

Si vous lancez ce script, vous allez constater un résultat inattendu : les deux instances créées sont bien différentes (il y a un homme et une femme), mais il partagent le même parent, donc ils ont le même nom. (Pas le même âge, car ce n'est pas enregistré dans le parent),

Si je modifie le nom de Personne, cette modification est aussitôt perceptible dans les deux objets à la fois.

En fait, les personnes ainsi créées n'ont pas de nom propre. Elles partagent le même car la property parent contient une référence qui pointe sur le même objet Personne.

Les deux lignes **copy** en commentaire peuvent remplacer les **set** qu'ils précèdent. Dans ce cas on obtient un fonctionnement plus conforme à ce que l'on désirait probablement : les deux objets Michel et Corinne sont vraiment distincts et séparés de la variable Personne, parce que la copie force une duplication de l'objet Personne et qu'au lieu d'un, nous en avons maintenant trois. Deux associés à Michel et Corinne (que l'on ne peut d'ailleurs plus atteindre directement) et celui qui se trouve encore dans la variable Personne.

Qu'en est-il de notre variable **zzz** utilisée dans la procédure originale ? Comme il s'agit d'une variable locale, elle devrait disparaître dès que l'exécution du constructeur est terminée, donc, en toute rigueur, on devrait se retrouver avec un objet dont le parent n'existe plus !!

Mais il existe une "protection" cachée dans le langage : comme il est impossible de modifier la property parent après coup, AppleScript s'assure que le parent ne disparaît pas à l'occasion de la suppression ou de l'effacement de la variable qui le

contient. Il l'incorpore à l'objet fils. (voir la section sur l'instruction load/store script).

On peut le vérifier également sur le script suivant : l'effacement de la variable "papa" déclenche une sorte de sauvegarde automatique.

```
script papa
  -- script minimaliste ;-)
  property prenom : "Jojo"
  property nom : "Wozniak"

  on affNom()
          return my prenom & " " & my nom
  end affNom
end script

script toto
  property parent : papa
  property prenom : "toto"
end script

log affNom() of toto --> "toto Wozniak"
log affNom() of papa --> "Jojo Wozniak"
copy "" to papa --> effacement de la variable papa

log parent of toto --> «script papa» papa survit en moi !!!
log affNom() of toto --> "toto Wozniak"
log affNom() of papa --> erreur !!!! je n'ai plus de papa !!!
```

Ceci peut également créer des problèmes lorsque l'on clone des scripts-objets par copy. S'ils n'ont pas de parent défini, le super-script étant leur parent par défaut, il est incorporé à l'objet-script, ce qui ne tarde pas à provoquer des erreurs graves (la saturation de la mémoire en tous cas). C'est la raison pour laquelle il vaut mieux éviter de cloner un script défini au premier niveau et utiliser un constructeur. Le script suivant plante en général dès la première exécution :

```
script Personne
  property nom : "Jojo"
end script

property liste : {}

repeat 10 times
  copy Personne to the end of liste
end repeat
```

Si l'on tient à faire du clonage, on peut spécifier un parent vide de la manière suivante :

```
script Personne
  property parent : "" -- = pas de parent
  property nom : "Jojo"
end script

property liste : {}
```

```
repeat 10 times
  copy Personne to the end of liste
end repeat
```

Cependant, cette disposition ne résoud pas tous les problèmes : en particulier, elle rompt les chaînes de recherche expliquées ci-dessous, ce qui n’est pas sans inconvénient.

Maintenant, ce n'est pas la seule particularité de l'héritage selon AppleScript. Nous avons vu que le modèle objet précise un ordre de recherche dans les classes pour les méthodes et les attributs, ordre basé sur leur hiérarchie. AppleScript ne respecte pas exactement cette prescription : il a même deux ordres différents, un pour les méthodes ou handlers, l'autre pour les attributs (properties).

Les chaînages de recherche.

Nous allons essayer de préciser la manière dont AppleScript recherche un handler ou une variable/property. Ceci nous permettra de préciser le fonctionnement de l'instruction **tell** et des variables prédéfinies : **it**, **me**, **current application**.

On peut considérer en effet qu'AppleScript recherche un handler ou une variable dans un script selon une sorte de chaîne d'objets qui sont reliés les uns aux autres. Comme dans l'héritage, ce qui n'est pas trouvé dans le premier objet de la chaîne est cherché dans le suivant et ainsi de suite, jusqu'au succès ou à l'échec en fin de chaîne. La différence avec l'héritage classique est qu'AppleScript rajoute implicitement des éléments dans la chaîne et qu'il la parcourt de façon différente selon qu'il s'agit d'un handler ou d'une variable.

Nous allons démontrer ce qui suit sur des exemples que vous pouvez copier/coller dans une fenêtre de votre éditeur de script. Nous appellerons le script ainsi créé le super-script (dans le dictionnaire AS, il est désigné sous le nom de "top-level script"). Il n'est pas fondamentalement différent des scripts-objets qu'il peut contenir, mais il a un rôle à part néanmoins.
Ces exemples ont été vérifiés soigneusement, cependant, le résultat tous les tests qui cherchent à mettre en lumière les maillons de la chaîne au delà du super-script dépend étroitement de votre configuration personnelle. Si vous avez installé Akua ou ce genre d'Osax, vous pouvez constater des résultats différents, car ces composants logiciels prennent aussi leur place dans la chaîne.

Notion d'application.

Prenons le cas le plus simple : celui d'un script ordinaire sans aucun objet-script incorporé. A priori, il n'est pas question d'héritage ni de parent ici. Cependant, si nous écrivons :

```
get parent --> «script AppleScript»
get parent of parent --> current application
```

nous voyons qu'il n'en est rien. Il existe deux parents au super-script (au delà, on rencontre une erreur).

Il n'est pas très facile de savoir ce qu'est exactement *script AppleScript* . Il existe effectivement une propriété de la classe app définie dans l'extension AS :

```
Class app : specifies global properties of AppleScript
Plural form:
  applications
Properties:
  result anything [r/o] -- the last result of evaluation
  space character [r/o] -- a space character
  return character [r/o] -- a return character
  tab character [r/o] -- a tab character
  minutes integer [r/o] -- the number of seconds in a minute
  hours integer [r/o] -- the number of seconds in an hour
  days integer [r/o] -- the number of seconds in a day
  weeks integer [r/o] -- the number of seconds in a week
  pi real [r/o] -- the constant pi
  print length integer [r/o] -- the maximum length to print
  print depth integer [r/o] -- the maximum depth to print
  text item delimiters list [r/o] -- the text item delimiters of a string
  AppleScript script [r/o] -- the top-level script object
```

Et lorsque l'on sauve le script suivant en tant qu'application :

```
set x to AppleScript of current application
store script x in file "test"
```

on obtient effectivement un fichier script, mais enregistré sans source. Donc on ne peut guère deviner ce qu'il y a dedans. Comme il est impossible de réaliser la même manœuvre sous un éditeur de scripts, on peut conjecturer qu'il s'agit d'une fraction du run-time qui est enregistrée de cette façon dans les scripts applications et incorporée d'une autre manière dans les éditeurs de scripts[49] . Mais c'est sans doute sans grande importance, d'ailleurs la documentation AS n'en souffle mot.

Par contre elle précise que l'application courante est le parent par défaut du super-script. Pourquoi ? AppleScript rajoute une relation d'appartenance de tout objet à une application, car c'est dans cette application que se trouvent la description et les méthodes de l'objet considéré.

Cette relation d'appartenance (qui n'est pas un héritage, car les objets ne "sont" pas des applications) est cependant exploitée par la recherche des handlers et des variables.

Si AppleScript ne trouve pas la variable ou le handler cherchés dans l'objet, il les cherche dans l'application dont il dépend (mais pas systématiquement dans l'éditeur de script ou le run-time AppleScript comme nous allons le voir bientôt). Nous appellerons cette application, l'application de référence de l'objet.

---

[49] La position de cet élément dans le chemin d'héritage suggère qu'il pourrait s'agir de la définition de la classe "script": l'objet parent de tous les scripts.

Quelle est cette application de référence ?

Pour un objet défini dans un dictionnaire d'application, c'est l'application qui fournit le dictionnaire et l'on ne peut rien y changer.
Pour un script, c'est l'éditeur de script si vous exécutez à partir de la fenêtre de l'éditeur. Si le script lui-même a été sauvé comme application, c'est le run-time AppleScript. Mais on peut également modifier le parent d 'un script pour le faire pointer sur une application.

```
property parent : application "Finder"
get parent --> application "Finder"
get parent of parent --> pas de parent
```

Encore une fois, ce n'est plus de l'héritage, mais une manière de relier un script à une application.

Il est important de ne pas confondre cette application de référence avec la variable prédéfinie "current application". Celle-ci désigne en effet l'application qui fournit au script les ressources d'exécution : c'est donc, soit l'éditeur de script, soit le script enregistré comme application autonome, soit une application capable de lancer des scripts. Il se trouve que current application est aussi par défaut l'application de référence des scripts-objets, mais pas celle de tous les objets !

Si je modifie le script ci dessus en :

```
property parent : application "Finder"
get current application --> current application
get name of current application --> "Smile"  N.B, cette ligne ne fonctionne qu'avec Smile
get name of parent --> "Finder"
```

Ici, current application est l'éditeur de script. Mais l'application de référence est le Finder.

Enchaînement dans les scripts-objets.

Compliquons maintenant les choses en définissant des scripts-objets dans le super script. Il est important de voir que par défaut le parent de tout objet-script est le super script.

```
property name : "super-script"

script Personne
  property nom : "Jojo"
end script

set x to the name of parent of Personne
display dialog x
```

renvoie "super-script" bien que nous n'ayons pas défini de parent pour Personne. La chaîne dont nous avons parlé se constitue maintenant des maillons suivants :

Personne -> super-script -> «script AppleScript» -> current application

Si le parent est défini explicitement dans Personne, ce parent prend la place du super-script. Si cet objet dépend d'une autre application de référence que current application, celle-ci est également remplacée.

C'est ce qui se produit lorsqu'on spécifie un objet parent qui n'est pas un script. Par exemple :

```
script Personne
  property parent : item "Corbeille" of application "Finder"
  property nom : "Jojo"
end script

get parent of Personne --> trash of application "Finder"
get the kind of Personne --> "dossier"
```

La chaîne de recherche pour la propriété **kind** de ce script Personne est :

Personne -> item Finder -> application "Finder".

Si plusieurs scripts-objets sont reliés par un héritage (au sens AS), la chaîne s'allonge simplement en ajoutant des termes, par exemple :

Femme -> Personne -> super-script -> «script AppleScript» -> current application.

Tell me of it !

AppleScript implémente correctement le modèle objet en ce sens que toutes les instructions du langage sont des opérations qui s'adressent à des objets. Il y a donc toujours un objet cible à qui s'adressent les opérations ou dont on essaie d'obtenir les propriétés. La syntaxe complète utilise l'instruction tell :

```
tell me to display dialog x
est équivalent à:
display dialog x
```

car, pour des raisons de simplicité d'écriture, lorsqu'on ne spécifie pas de cible, la cible par défaut est l'objet dans lequel se trouve l'instruction en exécution[50] .

Cet objet est désigné par la variable prédéfinie **me**. Cet objet est placé au début de la chaîne de recherche, et le reste est construit comme nous l'avons vu. On désigne donc ses composants par la syntaxe **of me** ou **my**. Ces deux formes :

```
get parent of me
get my parent
```

sont équivalentes.

Lorsque le super-script ne contient pas d'objet-script, **me** le désigne du début jusqu'à la fin. Lorsqu'il contient des scripts-objets, **me** désigne l'objet dans lequel se

[50] C'est le cas dans tous les langages orientés objet.

trouve le code en exécution, par exemple :

```
-- me est le super-script
property nom : "super-script"

script Personne  -- me = Personne
	property nom : "Jojo"

	on uneMethode()
		return my nom  -- nom of me, nom of Personne, Personne's nom
	end uneMethode

end script  -- me = Personne jusqu'ici

display dialog my nom -- nom of me (me = super-script), me's nom (îîîîrkkkk)
display dialog uneMethode() of Personne --> Jojo
```

Cette chaîne de recherche commençant à **me** est toujours disponible, mais ce n'est pas toujours la chaîne par défaut. En particulier les blocs et instructions **tell** ont justement pour objectif de changer la cible par défaut des opérations demandées[51] . Dans le script suivant, on a mis en place un bloc tell et des appels à des handlers appartenant soit au script Personne soit au super-script :

```
-- me est le super-script, it aussi
property nom : "super-script"

script Personne
  property nom : "Jojo"

  on uneMethode() -- me = Personne, it aussi
	return my nom
  end uneMethode -- jusqu'ici

end script

on uneMethode()
  return my nom -- me est le super-script
end uneMethode

display dialog uneMethode() --> super-script

tell Personne -- it = Personne
  display dialog uneMethode() --> Jojo
  display dialog my uneMethode() --> super-script, me reste le super-script
end tell  -- jusqu'ici
```

On remarquera que la même opération display dialog uneMethode() donne des résultats différents selon qu'on est dans le bloc tell ou non, simplement parce que l'objet cible par défaut change. On remarquera aussi que l'on peut toujours accéder à l'objet en exécution (ici le super-script) en utilisant **my** ou **me**.

Enfin, dans un bloc tell, la cible par défaut est stockée dans la variable

[51] De notre point de vue il est secondaire de considérer si la cible du tell est une application ou un objet, script ou pas. Le fonctionnement reste le même.

prédéfinie **it**. Ceci permet de référencer l'objet cible par défaut[52] .

Enfin, le mot clef **of** permet de désigner les properties ou les méthodes de l'objet-script. On peut parfaitement écrire :

```
parent of parent of parent of monObjet
```

pour aller chercher l'objet de la super classe à trois étages au dessus. Il faut naturellement que les properties parent existent. L'expression ne crée pas de liens “au vol”.

Méthodes ou handlers.

Maintenant que nous avons vu comment sont faites ces chaînes de recherche, à partir des relations définies par la property parent et les différents compléments implicites, voyons comment s'effectue la recherche des handlers. Elle s'effectue dans la chaîne de recherche de l'objet cible de l'opération. Le script suivant permet d'illustrer le fonctionnement :

```
property parent : item "Corbeille" of application "Finder"

on activate {} -- 3
  display dialog "super-script"
end activate

script ancetre
  --property parent : item "Corbeille" of application "Finder" -- 3

  on activate {} -- 2
        display dialog "Ancêtre"
  end activate

end script

script-objet
  property parent : ancetre

  on activate {} -- 1
        display dialog "Objet"
  end activate

end script

tell Objet to activate {}
```

En mettant en commentaire les handlers marqués de 1 à 3 (dans cet ordre), on peut tester les réactions de l’objet au même appel **tell** Objet **to** activate. Si la méthode existe dans Objet, elle est exécutée, sinon, elle est recherchée dans le parent (ligne 2), et si celle-ci manque également, soit dans le parent par défaut (super-script) ou le

[52] Mais me dira-t-on, pourquoi est-il utile de référencer l'objet par défaut puisqu'il est déjà le défaut ? Il suffit de ne rien mettre du tout ! Oui, et ben voilà, c'est malheureusement nécessaire dans certains cas.

parent qui n’est pas toujours un objet-script, ici la Corbeille.

Lorsque l'on se trouve dans un bloc tell qui spécifie une autre cible que **me**, c'est cette cible qui est recherchée pour les méthodes. Ce qui explique qu'il faille systématiquement utiliser la syntaxe **my** pour appeler un handler du script courant lorsqu'on se trouve à l'intérieur d'un bloc **tell**. En particulier, le super-script ne fait en général pas partie de la chaîne associée à la cible.

Variables et properties.

Là commencent les vraies difficultés. On pourrait espérer que la chaîne de recherche est la même que pour les méthodes, mais c'est plus complexe, même si le principe est le même. En fait, la question des variables vient perturber tout cela, sans compter quelques menues étrangetés. On ne saurait trop conseiller d'être attentif.

Propriété définie partout dans la chaîne de recherche.

C'est le cas le plus simple : considérons le script suivant et supprimons successivement les lignes définissant la property name dans l'ordre où elles sont numérotées.

```
property parent : item "Corbeille" of application "Finder"
property name : "super-script" -- 3

script ancetre
  property name : "ancêtre" -- 2

end script

script-objet
  property parent : ancetre
  property name : "mon Objet" -- 1

  on aff()
        return name
  end aff
end script

display dialog aff() of Objet
```

On obtient successivement "mon Objet", “ancêtre”, et "super-script". Jusqu'ici, rien de très inattendu. Par contre en supprimant la ligne 3, on n'obtient pas du tout "Corbeille" comme on pourrait l'espérer mais "Objet", autrement dit <u>le nom de la variable qui contient le script du même nom</u>.

C'est d'autant plus déroutant que l'accès direct comme dans l'exemple suivant donne bien comme résultat "Corbeille" : la property name n'étant pas définie dans le super-script, celle de son parent (la corbeille) devrait être renvoyée, qu’on la demande à l’objet ou au super-script. Or il n'en est rien :

```
property parent : item "Corbeille" of application "Finder"
--property name : "super-script" -- 3

script ancetre
  --property parent : item "Corbeille" of application "Finder"
  -- faire hériter directement ancetre de la Corbeille n'y change rien
  --property name : "Ancêtre" -- 2

end script

script-objet
  property parent : ancetre
  --property name : "mon Objet" -- 1

  on aff()
        return name -- my name n'y change rien
  end aff
end script

display dialog aff() of Objet --> Objet
set x to name
display dialog x --> Corbeille
```

Par contre, dans le script suivant où l'objet cible est anonyme (c'est pour ne pas lui donner de nom que nous l'avons placé dans une liste), le résultat est bien "Corbeille".

```
property parent : item "Corbeille" of application "Finder"
--property name : "super-script" -- 3

script ancetre
  --property parent : item "Corbeille" of application "Finder"
  -- faire hériter directement ancetre de la Corbeille n'y change rien
  --property name : "Ancêtre" -- 2

end script

script-objet
  property parent : ancetre
  --property name : "mon Objet" -- 1

  on aff()
        return name -- my name n'y change rien
  end aff
end script

set liste to {}
copy Objet to the end of liste
copy Objet to the end of liste
display dialog aff() of (item 1 of liste) --> "Corbeille"
```

Propriétés définies seulement dans les scripts.

On se demande peut-être pourquoi je m'obstine à utiliser la property name pour explorer les chemins d'héritage. Après tout, il est rare que l'on spécialise des classes d'objets existants. Il y a deux raisons à cela :

- La première est que seule une propriété définie pour les objets que je ne peux modifier permet de remonter le chemin de recherche jusqu'au bout.
- La deuxième est de mettre en évidence que l'on peut référencer sans s'en apercevoir une propriété existante (comme name) et obtenir des comportements "inexplicables" si on a oublié de la définir et de l'initialiser. Il n'y aura pas d'erreur mais un contenu venant d'on ne sait où.
- Enfin (on me pardonnera d'en rajouter une troisième), les propriétés définies par l'utilisateur ne fonctionnent pas exactement de la même manière.

Voyons le script suivant : il ne se distingue pas des exemples précédents sauf par le nom de la property **nom** plutôt que **name** : **nom** n’est évidemment pas une propriété des parents cachés.

```
property parent : item "Corbeille" of application "Finder"
property nom : "super-script" -- 3

script ancetre
  property nom : "Ancêtre" -- 2

end script

script-objet
  property parent : ancetre
  property nom : "mon Objet" -- 1

  on aff()
        return nom
  end aff
end script

display dialog aff() of Objet
```

Une fois de plus, nous mettons en commentaire les lignes numérotées les unes après les autres. Le premier essai renvoie "mon Objet" : c’est logique.

Le deuxième sort des rails : on attendait "Ancêtre" et c'est Blücher qui arrive sous la forme d'un "super-script". Comment se fait-il que le chemin explore le super-script avant l'objet "ancetre" ?

La documentation précise qu'il faut utiliser la syntaxe :

```
return my nom
```

pour forcer la recherche dans le chemin d'héritage. Ensuite tout rentre dans l'ordre et fonctionne comme prévu. Mais quelle est la raison de ce passage étrange par le super-script ? Ce sont les variables, que nous avions "oublié" dans notre étude.

Les variables.

En effet, AppleScript permet de définir n'importe où des variables globales[53] et locales et la syntaxe ne permet pas de les différencier des properties. Autrement dit, elles font toutes partie du même espace de noms. Il s'ensuit que des variables peuvent masquer une property. Le script suivant éclaire l'ordre de recherche :

```
global nom
set nom to "globale"

script ancetre
  property nom : "Ancêtre"

end script

script-objet
  property parent : ancetre
  property nom : "mon Objet" -- 2

  on aff()
        local nom -- 1
        set nom to "locale" -- 1
        return nom
  end aff
end script

display dialog aff() of Objet
```

Le premier appel renvoie "locale" : c'est donc la variable locale qui a priorité sur tout le reste, par contre si j'utilise la syntaxe **my** nom, c'est "mon Objet" qui arrive. Même résultat si j'élimine les lignes traitant la variable locale. Si j'élimine enfin la ligne 2, c'est la variable globale qui est récupérée. La syntaxe **my** nom, permet d'atteindre la property dans le script ancetre comme nous l'avons vu.

En résumé,

- si l'on spécifie **my** nom, la recherche suivra le chemin d'héritage en ignorant les variables.
- si l'on ne dit rien du tout, AS cherchera dans l'ordre : une variable locale, une property du script (pas les propriétés dans les parents), une globale, et enfin une propriété du super-script. Le chemin d'héritage est ignoré.

Accès direct. Usage de it.

La même situation se présente lorsque l'on utilise un bloc **tell** dans le super-script pour accéder à la property **nom** :

```
property nom : "super"

script ancetre
  property nom : "Ancêtre"
```

[53] Au contraire des autres langages orientés objet qui les ignorent complètement. Si on a besoin de globales, on les regroupe dans un objet général qui ne sert qu'à ça.

```
end script

script-objet
  property parent : ancetre
  property nom : "mon Objet"

  on aff()
        return nom
  end aff
end script

tell Objet
  display dialog nom --of it
  display dialog aff()
end tell
```

Ce script renvoie "super" même si la cible est clairement Objet. Autrement dit, les variables du super-script sont examinées et elles seulement. En effet, la suppression de la propriété nom du super-script conduit à un message disant que la variable n'est pas définie. Pour atteindre la propriété **nom** du script-objet (ou sa propriété héritée de Ancetre), il faut utiliser la syntaxe `nom of it` ou `it's nom`.

C'est d'autant plus déroutant que l'instruction `display dialog aff()` qui appelle une méthode de la cible, n'a pas besoin de cette précision.

Conclusion :

Le tableau que nous avons dressé peut donner l'impression effrayante d'un maquis épais dans lequel on ne peut manquer de se perdre dès que l'on programme des choses un peu compliquées. Pourtant, nous pensons qu'en appliquant quelques règles simples, on peut éviter facilement tous les problèmes que nous avons rencontrés. L'objectif de ces règles est de faire coïncider systématiquement le chemin de recherche avec le chemin d'héritage que l'on peut suivre à travers la propriété parent.

Si donc vous commencez à rencontrer des problèmes d'accès à vos handlers et variables, serrez les boulons de la manière suivante :

Utilisez my et it.

Utilisez **my** systématiquement pour accéder à des property ou des handlers définis dans vos scripts. Bien sûr, ce n'est pas toujours nécessaire en dehors d'un bloc **tell**, mais ça ne coûte pas cher et cela clarifie vos scripts car on sait du premier coup d'œil où doit se trouver la définition des handlers et properties.

Si le super-script ne se trouve pas dans votre chemin d'héritage, utilisez à la place, **of** current application.

A l'intérieur des blocs **tell**, vous pouvez aussi spécifier systématiquement **my** ou **it** selon que la cible voulue est le script ou un l'objet spécifié par le **tell**.

Evitez les variables globales.

Les variables globales ont tendance à obscurcir le chemin d'héritage. Remplacez

les par des properties qui ont un comportement plus conforme au modèle objet.

Evitez de surcharger sans raison des handlers ou properties.

N'oubliez pas que vous pouvez réemployer par hasard un nom de propriété existant dans les parties "invisibles" de la chaîne de recherche. AppleScript est très coulant sur le réemploi des noms, ce qui limite les erreurs de compilation toujours agaçantes, mais peut provoquer des comportements aberrants difficiles à débrouiller[54] .

Pour vous assurer qu'un nom de propriété ou de handler n'existe pas dans les profondeurs de votre configuration, supprimez sa définition dans votre script et testez : vous devez obtenir un message "la variable n'est pas définie" ou " xxx ne comprend pas le message zzz". Sinon, il est défini : changez son nom[55] .

Utilisez des constructeurs et évitez copy.

Pour les scripts-objets, évitez la construction automatique par :

```
script nomDeVariable
```

sauf dans le cas où vous n’avez besoin que d’un seul objet de ce type (on trouvera un exemple dans la section consacrée aux automates à états finis).

Utilisez un constructeur comme dans notre exemple où le résultat renvoyé est stocké dans la variable destiné à le recevoir. Evitez de cloner des scripts-objets par **copy** même si cela évite bien des problèmes liés à l'héritage d'un script à l'autre (parent commun, chemin de recherche des properties non conforme au chemin d'héritage, etc …).

---

[54] Nous avons rencontré un problème de ce type en rédigeant ces exemples, car le script "ancetre" avait été nommé "base". Quelque chose de ce nom existe dans l’OSAX Akua. Ce qui donnait des fonctionnements complètement différents selon qu'Akua était présent ou non.

[55] Pour ma part, j'utilise des noms tirés du français ce qui liquide le problème une fois pour toutes.

Quelques constructions avancées du langage AppleScript.

Trucs et astuces, ou pourquoi faire simple alors que l'on peut faire compliqué.

Ce chapitre n'a pas d'autre but que de signaler quelques possibilités d'écritures qui pourront vous servir, soit pour décrypter un code soit pour l'utiliser vous-même[56] . C'est souvent dans l'optique d'une meilleure lisibilité du code (ou moins bonne, selon les points de vues...), mais parfois cela peut servir à se sortir d'une situation embarrassante.

Nous profiterons de ces exemples pour nous servir de commandes peu utilisées et pour mettre en valeur certaines "curiosités"[57] d'AppleScript.

Dompter le signe de continuation ¬ en fin de ligne

Ce signe permet à AppleScript de considérer que plusieurs lignes sont en fait une seule ligne. Rien de bien nouveau, si ce n'est qu'à la compilation ce signe peut se retrouver dans certains cas à un endroit non désiré.

Par exemple vous aimeriez garder cette disposition :

```
display dialog "un texte"¬
default answer ""¬
buttons {"Annuler ", "Oui"} ¬
default button 2 ¬
with icon 1 ¬
giving up after 2
```

Malheureusement après compilation/vérification vous vous retrouverez avec ça :

```
display dialog ¬
  "un texte" default answer ¬
  "" buttons {"Annuler ", "Oui"} ¬
  default button 2 ¬
  with icon 1 ¬
  giving up after 2
```

L'astuce courante est d'employer des parenthèses autour des "textes" ce qui donnera :

---

[56] Pour tout renseignements complémentaires au sujet des différentes notions abordées, se référer au Guide AppleScript en français. Disponible à <http://trad.applescript.free.fr/Accueil.html>.
[57] Qui a dit bug ???

```
display dialog ("un texte") ¬
  default answer ("") ¬
  buttons {"Annuler ", "Oui"} ¬
  default button 2 ¬
  with icon 1 ¬
  giving up after 2
```

```
("C'est valable pour ") ¬
  & ¬
  "les concaténations et autres..."
```

Pour les **listes** pas de problème :

```
{"A", ¬
  1, ¬
  "B", ¬
  2}
```

Le résultat sera le même que le texte saisi.

Par contre pour les **records**... essayez donc de compiler ça :

```
{label1:"A",¬
label2:1,¬
label3:"B",¬
label4:2}
```

Vous obtiendrez :

```
¬
  ¬
          {label1:"A", label2:1, label3:"B", label4:2} ¬
```

Pas vraiment le résultat recherché (et ça énerve toujours...).
L'astuce est de placer le caractère de continuation et le saut de ligne avant la virgule :

```
{label1:"A"¬
, label2:1¬
, label3:"B"¬
, label4:2}
```

Ce qui donne en rajoutant quelques parenthèses autour des **strings** :

```
{label1:("A") ¬
  , label2:1 ¬
  , label3:("B") ¬
  , label4:2}
```

Ces quelques astuces pourront être utiles pour la mise en forme de scripts dans des mails ou autres pdf.

L'emploi de Result.

> AppleScript stocke le résultat d'une commande dans la variable prédéfinie **result**. La valeur stockée y reste jusqu'à ce qu'une prochaine commande soit exécutée. Si une commande ne retourne pas de résultat, la valeur de **result** est indéfinie.
> (extrait du Guide AppleScript version française)

Voilà de plus quelques définitions supplémentaires de cette même "Bible" qui peuvent être utiles pour comprendre la brutale solitude d'une valeur sur une ligne... :

> **Get**
> La commande **Get** peut fonctionner comme une commande AppleScript ou comme une commande d'application. La commande AppleScript retourne la valeur d'une expression. La commande d'application retourne la valeur d'un objet. Dans les deux cas, la commande assigne la valeur retournée à la variable prédéfinie **result**, cette variable est décrite dans "Utiliser la variable prédéfinie result" (T2 - p.21).
>
> Le terme **get** de la commande **Get** est optionnel, car AppleScript obtient automatiquement la valeur des expressions et des références quand elles apparaissent dans les scripts.
> (toujours extrait du GAS...)

Pour donner quelques repères, voilà tout de suite un exemple suivi de ses commentaires.

Avec un peu de chance, cet exemple retournera une erreur de **result** non défini :

```
set n to some item of {700, 701}
if n = 701 then
  "" & n & " a été aléatoirement choisi. "
end if
return result
```

L'instruction **if composée** sous-entend ceci :

```
if n = 701 then
  "" & n & " a été aléatoirement choisi. "
else
  -- RIEN DU TOUT
end if
```

Donc, en fonction du tirage aléatoire (par la référence **some**) le résultat sera, soit défini si c'est `701` qui sort, soit non défini si c'est `700`.

En ce qui concerne la ligne : `"" & n & " a été aléatoirement choisi "`

- C'est donc l'équivalent de : `get "" & n & " a été aléatoirement choisi "`
- La chaîne vide "" est employée en début de concaténation pour forcer le résultat en **string**.

Return est employé ici comme fin de la routine implicite **Run** de ce script.

Nous aurions pu nous en passer, la dernière ligne de **commande** d'une routine étant retournée automatiquement par AppleScript comme résultat de cette routine. On peut dire que la dernière ligne de **commande** d'une routine commence par un

return sous-entendu.

Si **Return** est précisé, mais sans rien à la suite, le résultat retourné est non défini.

Pour résumer ces subtilités, le simple script :

```
701
```

peut s'écrire de manière détaillée :

```
on run
  return get 701
end run
```

Voici un autre exemple renvoyant un résultat non défini :

```
maRoutine()
return result -- erreur : result n'est pas défini.

on maRoutine()
  701
  return -- ne renvoie RIEN
end maRoutine
```

On peut employer **result** pour éviter de définir une variable utilisateur supplémentaire. Voilà un exemple qui fera sourire (dans le meilleur des cas…), mais qui montre surtout que cette variable est utilisable de façon inhabituelle :

```
----------------------------------------
display dialog ("Veuillez saisir un entier de 1 à 3") ¬
  default answer ""

try
  text returned of result as integer
  if result ≤ 3 and result ≥ 1 then
        result
  else
        701
  end if
on error
  700
end try

if result = 1 then
  "un"
else if result = 2 then
  "deux"
else if result = 3 then
  "trois"
else if result = 700 then
  beep
  "quelque chose qui n'est pas un entier"
else if result = 701 then
  beep
  " un entier non compris entre 1 et 3"
end if
```

```
"Vous avez saisi : " & result
display dialog result
--------------------------------------
```

Result peut donc être utile pour tronçonner une commande complexe, pour simplifier la lecture ou l'écriture d'un script, ou pour en limiter la taille .

Mais plus prosaïquement cela peut servir quand les choses ne se passent pas comme prévu. Un résultat intermédiaire est parfois nécessaire pour passer outre une "curiosité". Par exemple :

```
set leRecord to {label1:("A") ¬
  , label2:1 ¬
  , label3:("B") ¬
  , label4:2}
return {label1 of leRecord, label3 of leRecord} as string
--> resultat : ""
```

C'est une "curiosité" AppleSscript... Le résultat est bien une chaîne vide. Ce n'est pas ce que nous souhaitions évidemment.

Une solution (parmi d'autres...) :

```
set leRecord to {label1:("A") ¬
  , label2:1 ¬
  , label3:("B") ¬
  , label4:2}
get {label1 of leRecord, label3 of leRecord}
return result as string
--> Résultat : "AB"
```

Un autre exemple :

```
tell application "Finder"
  set unFichier to choose file with prompt "Nom pas trop long SVP ;-)"
  make alias file ¬
          at (folder of item unFichier) ¬
          to item unFichier ¬
          with properties {name:"alias de " & (item unFichier's name)}
end tell
--> Resultat : erreur
```

Le Finder (ce n'est pas limité à lui...) a parfois du mal à exécuter une commande trop complexe en une seule ligne. Il retourne une erreur pas toujours compréhensible, et c'est ici le cas.

Une solution, simple et de bon goût, bien entendu, consiste à tronçonner délicatement la commande avec **result** :

```
tell application "Finder"
  set unFichier to choose file with prompt "Un nom pas trop long SVP..."
  folder of item unFichier -- = get folder of item unFichier
  make alias file ¬
          at result ¬
          to item unFichier ¬
          with properties {name:"alias de " & (item unFichier's name)}
end tell
```

--> *Resultat : l'alias est bien créé*

L'instruction Tell

> Les instructions **Tell** spécifient la cible par défaut, l'objet à qui sont envoyées les commandes si elles ne comportent pas de paramètre direct.
> [...]
> Quand AppleScript rencontre une référence partielle (une référence qui ne spécifie pas chaque container d'un objet), il utilise la cible par défaut pour la compléter.
> (extrait du Guide AppleScript version française).

L'instruction **Tell** peut donc servir avec tous les objets AppleScript et pas uniquement avec les applications. L'intérêt est d'éviter la répétition de l'objet cible, ce qui allège le code et sa compréhension.

Un exemple pour démarrer :

```
---------------------------------------
property listeP : {}

on run
  set laListe to {"Femme", "Homme"} & ¬
        {"Martien", "Martienne", "Martien neutre"}

  choose from list laListe
  set rep to first item of result

  tell laListe
        if rep = item 1 then
              copy my creaPersonne("1- " & item 1) to the end of listeP
        else if rep = item 2 then
              copy my creaPersonne("2- " & item 2) to the end of listeP
        else if rep = item 3 then
              copy my creaPersonne("3- " & item 3) to the end of listeP
        else if rep = item 4 then
              copy my creaPersonne("4- " & item 4) to the end of listeP
        else if rep = item 5 then
              copy my creaPersonne("5- " & item 5) to the end of listeP
        end if
  end tell
end run

on creaPersonne(nnn)
  return {ttt:nnn}
end creaPersonne
---------------------------------------
```

Ici item 1, item 2 etc... se réfèrent à la cible par défaut laListe que nous avons définie grâce à **Tell.**

my (ou **of me**) est nécessaire pour l'appel des routines n'appartenant pas à l'objet cible. **me** est une référence au script courant.

Dans certains cas il faut repréciser la cible par défaut avec **it.** C'est en particulier nécessaire quand la cible par défaut est un **record** ou un **script-objet.** AppleScript se voulant un langage proche du langage commun Anglais, l'appartenance **of it** peut-être remplacé indifféremment par **its** ou **it's**.

L'exemple suivant retourne une erreur :

```
tell {label1:("A") ¬
  , label2:1 ¬
  , label3:("B") ¬
  , label4:2}
  return "" & label2 & label1 & label3 & label4
end tell
--> Resultat : erreur
```

Avec les variantes **of it/its/it's,** ça fonctionne :

```
tell {label1:("A")
  , label2:1 ¬
  , label3:("B") ¬
  , label4:2}
  return "" & label2 of it & its label1 & it's label3 & its label4 -- variantes éducatives...
end tell
--> Résultat : "1AB2"
```

Attention néanmoins, **Tell** n'est pas une panacée : certains inconvénients peuvent apparaître !!! Entre autres avec certaines commandes qui nécessitent alors un **Tell me** ou **Tell application X** (appel au script courant ou à une application) pour fonctionner.

Reprenons l'exemple de démarrage, que nous allégeons grâce à **result** :

```
property listeP : {}

on run
  set laListe to {"Femme", "Homme"} & ¬
        {"Martien", "Martienne", "Martien neutre"}

  choose from list laListe
  set rep to first item of result

  tell laListe
        if rep = item 1 then
              "1- " & item 1
        else if rep = item 2 then
              "2- " & item 2
        else if rep = item 3 then
              "3- " & item 3
        else if rep = item 4 then
              "4- " & item 4
        else if rep = item 5 then
              "5- " & item 5
        end if
        copy my creaPersonne(result) to the end of listeP
  end tell
end run
```

```
on creaPersonne(nnn)
  return {ttt:nnn}
end creaPersonne
```

Nous n'avons pas envie de remplacer rep par **result**, pour l'instant. Mais par contre, nous repérons que la variable laListe est déjà utilisée un peu plus haut dans le script, et dans un élan irrépressible nous mettons la liste en référence par défaut :

```
----------------------------------------
property listeP : {}

on run
  tell {"Femme", "Homme"} & ¬
        {"Martien", "Martienne", "Martien neutre"}

        choose from list it
        set rep to first item of result

        if rep = item 1 then
              "1- " & item 1
        else if rep = item 2 then
              "2- " & item 2
        else if rep = item 3 then
              "3- " & item 3
        else if rep = item 4 then
              "4- " & item 4
        else if rep = item 5 then
              "5- " & item 5
        end if
        copy my creaPersonne(result) to the end of listeP
  end tell
end run

on creaPersonne(nnn)
  return {ttt:nnn}
end creaPersonne
----------------------------------------
```

Las... la commande choose from list fait des siennes dans ce **Tell.** Il nous faut donc la replacer dans le contexte du script courant :

```
tell {"Femme", "Homme"} & ¬
  {"Martien", "Martienne", "Martien neutre"}
  tell me to choose from list it -- it désigne ici le script courant
end tell
--> Resultat : erreur
```

Mais là, **it** désigne la cible courante de cette ligne. C.-à-d. le **me** de **tell me** : le script courant donc. Appelons à la rescousse super **result** pour résoudre ce cas épineux :

```
tell {"Femme", "Homme"} & ¬
  {"Martien", "Martienne", "Martien neutre"}
  get it
  tell me to choose from list result
end tell
```

Ouf ça marche... merci superResult. Ce qui nous permet d'écrire (mais est-ce vraiment raisonnable ?...) :

```
----------------------------------------
property listeP : {}

on run
  tell {"Femme", "Homme"} & ¬
         {"Martien", "Martienne", "Martien neutre"}
         it
         tell me to choose from list result
         set result to first item of result -- pourquoi pas...

         if result = item 1 then
                 "1- " & item 1
         else if result = item 2 then
                 "2- " & item 2
         else if result = item 3 then
                 "3- " & item 3
         else if result = item 4 then
                 "4- " & item 4
         else if result = item 5 then
                 "5- " & item 5
         end if
         copy my creaPersonne(result) to the end of listeP
  end tell
end run

on creaPersonne(nnn)
  return {ttt:nnn}
end creaPersonne
----------------------------------------
```

Gestion d'erreur.

Voici la syntaxe complète d'une instruction **Try** :

```
try
[ statement ]...
on error [ errorMessageVariable ] ¬
[ number errorNumberVariable ] ¬
[ from offendingObjectVariable ] ¬
[ partial result resultListVariable ] ¬
[ to expectedTypeVariable ]
[ global variable [, variable ]...]
[ local variable [, variable]...]
[ statement ]...
end [ error | try ]
```

À noter qu'à partir du système 9.0 (AS 1.4.0), la partie gestionnaire de l'instruction Try est devenue optionnelle. Si on ne gère pas les erreurs, on pourra donc écrire :

```
try
[ statement ]...
end [ try ]
```

La plupart du temps nous nous servons d'une forme simplifiée qui suffit largement à nos besoins :

```
try
  display dialog ("Veuillez saisir un entier de 1 à 3") ¬
         default answer ""
  set leNombre to text returned of result as integer
  if leNombre < 1 or leNombre > 3 then error number 701 -- erreur définie par le scripteur
on error errorMessageVariable number errorNumberVariable
  beep
  if errorNumberVariable = -128 then -- erreur système
         "Vous avez annulé le dialogue."
  else if errorNumberVariable = -1700 then -- erreur AppleEvent
         "Le texte saisi ne peut pas être convertit en nombre entier."
  else if errorNumberVariable = 701 then -- erreur définie par le scripteur
         "Le nombre saisi n'est pas un entier compris entre 1 et 3."
  end if
  display dialog result buttons {"OK"} default button 1 with icon 0
end try
```

Il existe une possibilité de ne gérer qu'un seul type d'erreur : en donnant une valeur précise à un ou plusieurs paramètres du gestionnaire **on error**. Par exemple si vous voulez uniquement gérer l'erreur -1700 ("Impossible de faire de machinBidule un integer") :

```
try
  display dialog "Veuillez saisir un entier" default answer ""
  set leNombre to text returned of result as integer
on error number -1700
  "Le texte saisi ne peut pas être converti en nombre entier."
  display dialog result buttons {"OK"} default button 1 with icon 0
end try
```

ou bien avec un autre paramètre, et une erreur "exotique"[58] :

```
try
  display dialog "Veuillez saisir un entier de 1 à 3" default answer ""
  set leNombre to text returned of result as integer
  if leNombre < 1 or leNombre > 3 then error "Coucou" -- erreur définie par le scripteur
on error "Coucou"
  "Le nombre saisi n'est pas un entier compris entre 1 et 3."
  display dialog result buttons {"OK"} default button 1 with icon 0
end try
```

avec deux paramètres, pour essayer de passer outre un bug du "Finder" de la coercition `as alias list`. Une erreur intervenant si le dossier ne contient qu'un élément

---

[58] Il est à noter qu'Apple recommande de choisir un numéro d'erreur positif de 500 à 10 000 pour éviter des conflit avec d'autres numéros d'erreurs existants.
Si vous n'indiquez pas de paramètre number, la valeur -2700 (erreur inconnue) sera envoyée au gestionnaire d'erreur.

et/ou aucun (en fonction des versions apparemment) :

```
set leDossier to choose folder

tell application "Finder"
  try
        items of folder leDossier as alias list
  on error number -1700 from offendingObjectVariable
        offendingObjectVariable
  end try
  set laListe to result as list
end tell
```

Surcharger une commande.

Surcharger une commande c'est simplement modifier une commande préexistante. Nous parlerons ici de commandes prédéfinies par AppleScript, un Osax ou même une application, qui seront redéfinies par le scripteur à l'aide d'un handler de même nom.

```
beep
--> Résultat : le dialogue de la routine

on beep
  display dialog "beep"
end beep
```

Chaque fois qu'il y aura la commande **beep** dans ce script c'est la routine qui se lancera : le dialogue au lieu du son.

La cible par défaut (le début de la recherche) est le script. AppleScript cherche donc d'abord dans le script, trouve la routine et ne va pas plus loin. Pour obliger AppleScript à commencer la recherche aux niveaux d'héritages supérieurs, et donc exécuter la commande par une autre application, on peut rajouter continue :

```
continue beep -- envoie la commande à l'éditeur de script
--> Résultat :  le son beep

on beep
  display dialog "beep"
end beep
```

Cette routine est un gestionnaire presque "normal"[59] !!!! Apple appelle ça un gestionnaire de commande. On aurait pu écrire ce script avec les 2 formes classiques des gestionnaires :

---

[59] Différences entre routine et gestionnaire de commande.
Pour un gestionnaire de commande :
- Pas de forme positionnée (). Mieux vaut éviter.
- Pour la forme étiquetée, pas d'étiquettes"exotiques" (into, from, around, etc...). Donc seulement given pour les paramètres secondaires.
Possibilité de mettre un paramètre unique sans rien : c'est le paramètre principal.

1) Avec paramètres positionnés.

Forme : (on | to) commandName ([paramVariable [, paramVariable]...])

Non référencée par Apple pour les gestionnaires de commande. À éviter donc, c'est juste pour dire que ça peut exister.

```
-- positionné avec 1 paramètre
beep ("beep") -- forme non référencée par Apple

on beep (zzz)
  display dialog zzz
end beep

-- positionné avec n paramètres
beep {"beep", "rebeep", "rerebeep"} -- forme non référencée par Apple

on beep {z, zz, zzz}
  display dialog z & zz & zzz
end beep
```

2) La même chose avec paramètres étiquetés :

Forme : (on | to) commandName [¬
[of] directParameterVariable] ¬
[given label:paramVariable [, label:paramVariable]...]

```
-- étiqueté avec 1 paramètre
beep "beep" -- 1 paramètre direct

on beep zzz
  display dialog zzz
end beep


-- étiqueté avec n paramètres
-- 1 paramètre direct + 2 paramètres secondaires
beep "beep" given bbb:"rebeep", ccc:"rerebeep"

on beep z given bbb:zz, ccc:zzz
  display dialog z & zz & zzz
end beep
```

Run script

Run Script est une commande d'OSAX des Compléments standard d'Apple.

La commande **Run Script** exécute un script qui est, soit un script dans le même script, soit un fichier script extérieur.

Syntaxe

`run script` *referenceOrString* ¬
`[ with parameters` *listOfParameters* `]` ¬
`[ in` *scriptingComponent* `]`

Paramètres

referenceOrString Une référence telle que `file` *nameString* ou `alias` *nameString* qui spécifie un fichier script, ou une chaîne de caractères composée d'un script valide.
Classe : Reference ou String

listOfParameters Une liste de paramètres devant être transmis au gestionnaire Run de la cible.
Classe : List

scriptingComponent Le nom du scripting component à utiliser lors de l'exécution du script.
Classe : String

Résultats

La valeur retournée par l'exécution du script.
(extrait de GAS_complements_standard.pdf [60])

Le résultat de cette commande n'est que ce que renvoie le handler run du script exécuté. Rien de plus.

Cette commande permet d'exécuter un script déjà compilé, mais surtout de compiler un texte puis d'exécuter son handler run. C'est ce dernier point qui nous intéresse évidemment ici, en se rappelant toutefois que la compilation ne fait rien pour accélérer un script...

```
set x to 2
display dialog "Saisissez la formule :" & return & "Avec x = " & x default answer "20 * x"
set script_texte to text returned of result
set script_texte to "property X :" & x & return & script_texte
set resultat to run script script_texte
display dialog "Le résultat est : " & resultat
```

On peut créer avec **Run Script** des objets de script :

Des records :

```
set etiquettes to {"nom", "prenom", "age"}
set valeurs to {"\"totor\"", "\"albert\"", 12}
```

[60] Disponible à <http://trad.applescript.free.fr/Accueil.html>.

```
set script_texte to {}

repeat with i from 1 to count etiquettes
  set the end of script_texte to "" & item i of etiquettes & ¬
          ":" & item i of valeurs
end repeat

set AppleScript's text item delimiters to {", "}
set script_texte to script_texte as string
set AppleScript's text item delimiters to {""}

run script "{" & script_texte & "}" -- création d'un record
-->Résultat : {nom:"totor", prenom:"albert", age:12}
```

Des scripts-objets.

Ne pas oublier de les créer dans un run **explicite**, ou d'appeler une routine dédiée.

```
set script_texte to ¬
  "on run
  script test
          property valeur : 200

          on run
                    my dlg()
          end run

          on dlg()
                    display dialog valeur as string
          end dlg
  end script
end run"

set unScript to run script script_texte -- création du script-objet
tell unScript
  display dialog its name
  set its valeur to 300
  run -- appelle la routine run de unScript
  run script -- la commande exécute aussi les scripts-objets...
end tell
store script unScript in file ("" & (path to desktop) & "test")
```

et même des handler...

Pour plus de détails sur les manipulations de handlers par variables, voir le chapitre "**Pointeurs de fonctions**.".

```
set script_texte to ¬
  "return my dlg

on dlg(unTexte)
```

```
  display dialog unTexte
end dlg"

set monHandler to run script script_texte  -- création du Handler
monHandler("Salut monde !")
```

Pour terminer cet aperçu des possibilités de la commande **Run Script**, on notera, avec tout l'intérêt requis, qu'elle permet aussi d'exécuter des scripts qui sont rédigés dans d'autres langages qu'AppleScript, comme Quickeys et Tcl/Tk. Il faut alors spécifier le paramètre "scripting component" et, naturellement, que Quickeys, Tcl/Tk, ou autres langages supportant l'OSA (Open Scripting Architecture), soient installés sur votre système.

Pointeurs de fonctions.

Ce que nous décrivons ici n'est pas documenté par Apple. Un peu par hasard, nous nous sommes rendus compte que les handlers se comportaient comme des variables d'un type spécial «handler» . Ainsi, quand on écrit :

```
on routine(test)
  return "Bonjour " & test & " !"
end routine
```

on définit une variable appelée "routine" de type handler dont le contenu est le corps de la fonction, autrement dit le code compris entre **on** -- **end**. On peut en effet en lire le contenu comme une variable ordinaire :

```
display dialog routine("Daniel") --> Bonjour Daniel ! sans surprise, mais :
-- si la coercition en string est rendue possible (avec OSAX ou application. Jon's Commands, Smile etc...) :
display dialog routine --> return "Bonjour " & test & " !" -- on récupère le corps de la fonction, sinon :
get routine --> Résultat : «handler routine»
```

L'usage du nom de handler sans aucun paramètre référence donc le corps de la fonction. On peut aussi écrire dans une variable de type handler [61] . Ceci ne serait qu'une curiosité dangereuse s'il n'était également possible d'affecter le corps de la fonction à une nouvelle variable ou property et de se servir de celle-ci comme handler de la manière suivante :

```
property routine : "" -- un pointeur de fonction : on peut l'initialiser à n'importe quoi

on bonjour(test)
  return "Bonjour " & test & " !"
end bonjour
```

---

[61] Ce que nous ne saurions trop déconseiller: le résultat est un programme qui ne correspond plus à sa version source, autrement dit un programme vraiment incompréhensible.
Signalons aussi qu'il est impossible de remplacer directement le corps du handler "routine" par une série d'instructions au format texte:
set routine **to** "return \"Adieu \" & test & \" !\" "
échoue. Mais comme nous l'avons montré ailleurs, c'est possible avec un run script.

```
on adieu(test)
  return "Adieu " & test & " !"
end adieu

set routine to bonjour -- pour que routine exécute en fait la fonction bonjour
display dialog routine("Daniel") --> Bonjour Daniel !
set routine to[62]  adieu -- pour que routine exécute en fait la fonction adieu

display dialog routine("Daniel") --> Adieu Daniel !
```

Dans cet exemple "routine" a beau être une property, elle peut contenir un handler et fonctionner comme un handler. Cette property est ce qu'on appelle un pointeur de fonction car elle sert à désigner la fonction qui sera réellement exécutée.

La documentation Apple ne souffle mot de cette fonctionnalité. Nos tests montrent cependant qu'elle est présente dans toutes les versions et utilisable sans accrocs sous la forme que nous venons de présenter. Mais bien sûr, comme il ne s'agit pas d'une fonction documentée, la prudence s'impose.

Donnons quelques exemples d'utilisation des pointeurs de fonctions[63] . Exactement comme le polymorphisme auquel ils sont étroitement liés, les pointeurs de fonctions permettent d'éviter la répétition de tests compliqués ou de routines semblables. Ils fonctionnent comme un aiguillage de chemin de fer : que l'on peut positionner vers des directions différentes et qui garde sa position tant qu'on n'y touche pas[64] .

Premier usage : traitements multiples, accès unique.

Soit une liste de nombres : je veux réaliser des fonctions qui ne traitent que les éléments pairs et leur appliquent diverses opérations arithmétiques : additionner 2, retrancher 2, les remplacer par le reste de la division entière par 3.

Il tombe sous le sens que ces trois fonctions auront une similitude prononcée :

```
on faire(laListe)
  set lister to {}
  repeat with i in laListe
        if (i mod 2) = 0 then
              -- l'opération arithmétique prend place ici
        end if
        copy i to the end of lister
  end repeat
  return lister
```

[62] En utilisant cette syntaxe, vous constaterez peut-être qu'elle ne fonctionne qu'à condition que les handlers "bonjour" et "adieu" soient définis dans le script avant l'affectation "**set** routine **to** bonjour". Pour que l'affectation fonctionne même si elle se trouve avant la définition du handler, utiliser la syntaxe "**set** routine **to my** bonjour".

[63] Qui existent dans beaucoup d'autres langages (C, C++, Java, etc…). Signalons d'autre part, que les µP possèdent des instructions spéciales très efficaces pour implémenter les pointeurs de fonctions.

[64] Les handlers sont des variables persistantes d'une exécution à l'autre comme les properties et les variables globales. L'aiguillage reste donc positionné d'une exécution à l'autre si le pointeur de fonction est une property ou que l'on modifie un handler ordinaire.

```
end faire
```

Seule la ligne en commentaire est différente dans chaque fonction. On peut donc la remplacer par un pointeur de fonction qui va désigner l'opération voulue que l'on passe aussi en paramètre :

```
property traiter : "" -- le pointeur de fonction

on faire(laListe, fptr)
  set traiter to fptr
  set lister to {}
  repeat with i in laListe
        if (i mod 2) = 0 then
              set i to traiter(i) -- l'appel via le pointeur de fonction
        end if
        copy i to the end of lister
  end repeat
  return lister
end faire

set liste to {1, 2, 3, 4, 5, 6}
log faire(liste, my add) -- exemples d'appel
log faire(liste, my subst)
log faire(liste, my modulo)

------------------- les fonctions arithmétiques ------------------
on add(k)
  k + 2
end add

on subst(k)
  k - 2
end subst

on modulo(k)
  k mod 3
end modulo
```

Il est évident que plus la partie commune est complexe, plus la solution a d'intérêt. Dans l'exemple suivant, l'algorithme de tri utilisé (interclassement ou fusion) est très efficace, mais il est assez long. Il contient naturellement des comparaisons entre les éléments. En utilisant un pointeur de fonction pour ces comparaisons, on peut utiliser le même handler pour traiter non seulement des tris ascendants et descendants de nombres ou de chaînes , mais aussi des listes de records triés par des champs différents. Nous avons mis en rouge les utilisations du pointeur de fonctions :

```
(* Tri par interclassement
--> renvoie une copie triée de la liste originale.
Algorithme : L. Sebilleau & D. Varlet
*)

on run
  ------------ exemple d'utilisations -----------
  set maListe to {1, 5, 15, 2, 7, 3, 23, 125, 4, 15, 8, 7, 6, 5, 27, 35, 12, 22, 16, 47}
```

```
  repeat 7 times
          set maListe to maListe & maListe -- l'étendre à 2 560 éléments
  end repeat

  set t1 to current date -- évaluation du temps d'exécution

  set maListe to intersort(maListe, my cmpasc) -- tri ascendant, notez le pointeur en paramètre

  set t2 to current date
  set laps to t2 - t1
  log "Durée: " & laps & " secondes" -- 4 secondes chez moi (G3 233Mhz)

  set maListe to intersort(maListe, my cmpdesc) -- tri descendant

  -- tri de listes de records
  set maListe to {¬
          {id:1, nom:"totor", Age:20, sexe:"M", list_IDconjoint:{}, IDPlanete:1}, ¬
          {id:2, nom:"Zézette", Age:100, sexe:"F", list_IDconjoint:{}, IDPlanete:1}, ¬
          {id:3, nom:"Lulu", Age:256, sexe:"M", list_IDconjoint:{}, IDPlanete:2}, ¬
          {id:4, nom:"Minouchet", Age:201, sexe:"F", list_IDconjoint:{}, IDPlanete:2}, ¬
          {id:5, nom:"Grmblwz", Age:12, sexe:"N", list_IDconjoint:{}, IDPlanete:2} ¬
                  }
  set maListe to intersort(maListe, my recmp) -- tri alphabétique sur les noms
  set maListe to intersort(maListe, my agecmp) -- tri par âges

end run

------------ les fonctions de comparaison ------------
--> les deux items de la liste à comparer
-- <-- renvoient vrai ou faux selon le résultat de la comparaison

on cmpasc(n1, n2) -- pour le tri ascendant des nombres et des chaînes
  return n1 < n2
end cmpasc

on cmpdesc(n1, n2) -- tri descendant
  return n1 > n2
end cmpdesc

on recmp(n1, n2) -- tri par noms des records
  return (nom of n1 < nom of n2)
end recmp

on agecmp(n1, n2) -- tri par âges des records
  return (Age of n1 < Age of n2)
end agecmp

------------ la routine principale

on intersort(laListe, fcmp)
  -- le deuxième paramètre est un pointeur sur la fonction de comparaison à employer

  script lstock -- l'intérêt de ce script sera exposé plus loin
          property listeorg : laListe
          property liste1 : {}
          property liste2 : {}
          property listea : {}
          property listeb : {}
```

```
		property lister : {}
		property Compare : fcmp -- le pointeur de fonction est stocké là
	end script

	-- répartition des éléments à trier en sous-listes de longueur deux

	tell lstock
		set ct to (its listeorg)'s length
		set flag to false
		if (ct mod 2) = 1 then
			set ct to ct - 1
			set flag to true
		end if
		repeat with i from 1 to ct by 2
			set x to item i of its listeorg
			set y to item (i + 1) of its listeorg
			if Compare(x, y) then -- la fonction de comparaison est employée ici
				{x, y}
			else
				{y, x}
			end if
			set the end of its liste1 to result
		end repeat
		if flag then set the end of its liste1 to {item -1 of its listeorg}

		-- interclassement des listes   contenues dans liste1
		repeat
			set ct to (its liste1)'s length
			if ct = 1 then exit repeat -- tant qu'il me reste plusieurs listes, c'est pas fini
			set flag to false
			if (ct mod 2) = 1 then
				set ct to ct - 1
				set flag to true
			end if
			set its liste2 to {}
			repeat with i from 1 to ct by 2 -- on prend les listes 2 par 2
				set its listea to item i of its liste1
				set its listeb to item (i + 1) of its liste1
				set its lister to {} -- pour n'en faire qu'une seule dans celle-là
				set cta to (its listea)'s length
				set ctb to (its listeb)'s length
				set ya to 1
				set yb to 1
				repeat

					-- la fonction de comparaison est aussi employée ici :
					if Compare(item yb of its listeb, item ya of its listea) then --
comparaison des premiers éléments
						set the end of its lister to (item yb of its listeb) --
"extraction" du premier

						set yb to yb + 1
						if yb > ctb then -- si cette liste est maintenant vide, on arrête là.
							if ya ≤ cta then -- il peut rester des éléments dans l'une des
listes
								set its lister to (its lister) & (items ya thru -
1 of its listea)

							end if
							exit repeat
```

```
end if
else -- symétrique dans le cas inverse
    set the end of its lister to (item ya of its listea)
    set ya to ya + 1
    if ya > cta then
        if yb ≤ ctb then -- il peut rester des éléments dans l'une des listes
            set its lister to (its lister) & (items yb thru -1 of its listeb)
        end if
        exit repeat
    end if
end if
end repeat
-- la nouvelle liste produite par la fusion est ajoutée en tant que liste au résultat
set the end of its liste2 to its lister
end repeat -- et on recommence tant qu'il reste des paires de listes
-- sans oublier l'orpheline si le nombre de listes est impair : on l'ajoute simplement à la fin
if flag then set the end of its liste2 to {} & item -1 of its liste1
set its liste1 to its liste2 -- et c'est reparti pour un tour
end repeat
return first item of its liste1 -- parce que c'est une liste de listes, même si elle n'en contient plus qu'une seule
end tell
end intersort
```

Pour utiliser la procédure, il suffit donc d'écrire une procédure de comparaison qui tient généralement en peu de lignes au lieu de copier/coller le handler tout entier pour modifier seulement les lignes où se font les comparaisons (comme ici il y en a deux, on peut facilement en oublier une).

Deuxième usage : éviter des tests multiples.

Nous reprenons ici un des exemples de la documentation Apple, le script John qui répond Hello ! un certain nombre de fois seulement. Dans la version originale, on peut observer que le test devient inutile à partir du moment où l'on a dépassé le compte. Si l'on veut économiser ce test, on peut utiliser un pointeur de fonction :

```
script John
  property cnt : 1
  property sayHello : ""

  set sayHello to my Hello1 -- run implicite
  set cnt to 1

  on Hello1(nn)
      display dialog "Hello " & nn & " !"
      if cnt < 2 then
          set cnt to cnt + 1
      else
          set sayHello to my Hello2 -- on aiguille définitivement vers la deuxième réponse
      end if
  end Hello1

  on Hello2(nn)
```

```
            display dialog "I'm tired to say Hello !"
    end Hello2

end script

tell John to run -- pour initialiser le pointeur : property sayHello :my Hello1 -- ne marche pas
tell John to sayHello("Béatrice")
tell John to sayHello("Michèle")
tell John to sayHello("John")
```

Dès que le compte est dépassé, le pointeur de fonction dirige l'exécution directement sur Hello2, ce qui permet de faire l'économie de la gestion ultérieure du compteur et du test quand ils ne servent plus à rien.

L'intérêt est plus net lorsque le test est composite : dans ce script, John est capable de dire bonjour indéfiniment, tant qu'il s'agit d'interlocuteurs différents. Sinon, ses réponses deviennent de plus en plus distantes.

```
script John
    property mem : "" -- enregistre le nom de l'interlocuteur
    property Hello : "" -- pointeur de fonction

    on sayHello(nn)
            if mem ≠ nn then -- si l'interlocuteur est différent, soyons civil !
                    Hello1(nn)
            else
                    Hello(nn) -- sinon, manifestons un ennui croissant
            end if
    end sayHello

    on Hello1(nn) -- la réponse standard
            display dialog "Hello " & nn & " !"
            set mem to nn
            set Hello to my Hello2
    end Hello1

    on Hello2(nn) -- deuxième fois
            display dialog "Well, Hello " & nn & " !"
            set Hello to my Hello3
    end Hello2

    on Hello3(nn)
            display dialog "Please, don't try to fool me " & nn & " !"
            set Hello to my Hello4
    end Hello3

    on Hello4(nn)
            display dialog "The hell with you, " & nn & " !"
            set Hello to my Hello5
    end Hello4

    on Hello5(nn)
            display dialog "Shit !"
    end Hello5

end script

tell John to sayHello("Béatrice")
```

```
tell John to sayHello("Michèle")
tell John to sayHello("Béatrice")
tell John to sayHello("Béatrice")
tell John to sayHello("Béatrice")
tell John to sayHello("Béatrice")
tell John to sayHello("Béatrice")
```

La simplification et la souplesse apportée par cette disposition n'est pas forcément évidente sur des exemples aussi simples. On peut obtenir le même résultat avec le script suivant qui utilise des techniques plus classiques sans être plus compliqué :

```
script John
  property mem : "" -- enregistre le nom de l'interlocuteur
  property cnt : 1

  on sayHello(nn)
        if mem ≠ nn then -- si l'interlocuteur est différent, soyons civil !
              set cnt to 1
        else -- sinon, manifestons un ennui croissant
              set cnt to cnt + 1
              if cnt > 5 then set cnt to 5
        end if
        display dialog item cnt of {¬
              "Hello " & nn & " !", ¬
              "Well, Hello " & nn & " !", ¬
              "Please, don't try to fool me " & nn & " !", ¬
              "The hell with you, " & nn & " !", ¬
              "Shit !"}
        set mem to nn
  end sayHello
end script

tell John to sayHello("Béatrice")
tell John to sayHello("Michèle")
tell John to sayHello("Béatrice")
tell John to sayHello("Béatrice")
tell John to sayHello("Béatrice")
tell John to sayHello("Béatrice")
tell John to sayHello("Béatrice")
```

Pour illustrer les avantages de cette approche, voici un exemple plus compliqué.

Automate à états finis.

Ceci représente mon comportement ordinaire lorsque je travaille à la maison, en particulier, la manière dont je réponds aux appels téléphoniques (T) ou à la sonnette de la porte d'entrée (S). [65]

---

[65] J'espère que le lecteur sera sensible à la confiance que je lui fais en lui révélant ma vie privée et qu'il s'abstiendra de sarcasmes trop faciles sur mes habitudes sexuelles (Je sais, en armure, c'est grotesque, mais si elles aiment ça…).

Les ronds représentent des états dans lesquels je peux me trouver et qui conditionnent ma manière de répondre. Il est par exemple évident que ne peux pas prendre mes maîtresses au téléphone pendant que je suis occupé avec l'une d'elles, car le bruit de l'armure pourrait nous trahir.

Les flèches représentent des changements d'états qui surviennent à la suite d'événements comme les appels téléphoniques ou l'échéance d'un délai (je ne reste pas dans mon bain jusqu'à ce que Zézette sonne, il faut que j'enfile mon armure). Ces flèches représentent donc tous les enchaînements d'événements possibles (vous remarquerez par exemple que je ne reçois pas mes maîtresses sans avoir pris un

bain)[66] .

Le but du script est de fournir des handlers qui simulent les réponses aux événements extérieurs :

```
-- 1 - Appel téléphonique
on appelTel(qui)
  tel(qui)
end appelTel
-- 2 - On sonne à la porte d'entrée
on sonnePorte(qui)
  sonne(qui)
end sonnePorte
-- 3 - Un certain délai est écoulé
on dringDring()
  dring()
end dringDring
-- 4 - Le jour fait place à la nuit ou le contraire.
on aubeAuxDoigtsdeRose()
  set jour to not jour
  dodo()
end aubeAuxDoigtsdeRose
```

S'y ajoutent deux événements qui peuvent être déclenchés par une horloge externe : le passage du jour à la nuit. Pour ce faire, on met en place un certain nombre de pointeurs de fonction :

```
property tel : "" -- pointeur de fonction répondre au téléphone
property sonne : ""
property dring : ""
property dodo : ""
```

dans lesquels on placera une référence à la procédure adéquate. Un changement d'état va donc correspondre à une mise en place de nouveaux aiguillages pour diriger le programme vers un comportement approprié. L'entrée dans l'état 1 est le handler "etat1" qui met ces aiguillages en place :

```
------------------ état 1 : disponible mais sale
on tel1(qui)
  if qui is in mescopines then
        etat2() -- je file prendre mon bain
        set laCopine to qui
        set sortie to my etat4 -- et après en armure !
        return "D'ici une heure, d'accord ?"
  else
        return "Allô ?"
  end if
end tel1

on dring1() -- je décide de me laver quand même
  etat2()
```

[66] Les spécialistes auront reconnu dans cette représentation un automate à états finis. Les non-spécialistes savent aussi maintenant que c'est une manière pratique de formaliser un grand nombre de systèmes comme des automates de connexion (implémentant un protocole comme TCP/IP) ou d'analyse syntaxique (dans les compilateurs).

```
  set sortie to my etat3 -- ensuite je ferais mon courrier
  set tel to my tel21 -- et je suis disponible pour répondre à mes copines
end dring1

on etat1()
  set laCopine to ""
  set tel to tel1
  set sonne to defSonne -- pas de réponse à la porte
  set dring to dring1
  set dodo to my etat7 -- on va au lit immédiatement
  set estat to "1-Je glande"
end etat1
```

On remarquera qu'il n'y a pas de handler spécifique à l'état 1 pour la réponse à la sonnette de la porte d'entrée : on utilise un handler général qui est utilisé dans plusieurs états. Par ailleurs, le pointeur de fonction "sortie" permet d'enregistrer facilement ce que je dois faire après avoir pris mon bain : mettre mon armure si j'y vais suite à l'appel d'une de mes maîtresses ou spontanément. Les appels d' "etat2" et "etat7" représentent les deux issues possibles de cet état.

Les autres états fonctionnent de même. Ils possèdent des handlers spécifiques si c'est nécessaire, sinon, ils font appel à des handlers communs à plusieurs états.

```
(* Automate implémenté par pointeurs de fonction *)

----------- les pointeurs de fonction
property tel : "" -- pointeur de fonction répondre au téléphone
property sonne : "" -- coup de sonnette à la porte
property dring : "" -- le réveil ou l'écoulement d'un délai
property dodo : "" -- l'arrivée du soir
property sortie : "" -- que dois-je faire en sortant du bain ?

property jour : true -- le flag jour/nuit

property mescopines : {"Zézette", "Marjorie", "Chantal", "Cunégonde"}
property laCopine : "" -- celle que j'attends

property estat : "" -- pour la démo

(* Ce script est conçu comme une appli d'arrière-plan qui reçoit des messages par les points d'entrées
suivants :
Ces points d'entrée utilisent immédiatement un pointeur de fonction qui leur fait exécuter l'opération appropriée
*)

-- 1 - Appel téléphonique
on appelTel(qui)
  tel(qui)
end appelTel
-- 2 - On sonne à la porte d'entrée
on sonnePorte(qui)
  sonne(qui)
end sonnePorte
-- 3 - Un certain délai est écoulé
on dringDring()
  dring()
end dringDring
-- 4 - Le jour fait place à la nuit ou le contraire.
on aubeAuxDoigtsdeRose()
```

```
  if jour then
          dodo()
  else
          dring()
  end if
  set jour to not jour
end aubeAuxDoigtsdeRose

----------- les procédures par défaut  ---------
on defTel(qui)
  return "Par suite de l'encombrement des lignes, votre appel ne peut aboutir."
end defTel

on defSonne(qui)
  return "Il n'y a visiblement personne."
end defSonne

on defDring()
  return
end defDring

on defDodo()
  return
end defDodo

------------------ le run réalise l'initialisation dans l'état 1
set jour to true
etat1()

------------------ état 1 : disponible mais sale
on tel1(qui)
  if qui is in mescopines then
          etat2() -- je file prendre mon bain
          set laCopine to qui
          set sortie to my etat4 -- et après en armure !
          return "D'ici une heure, d'accord ?"
  else
          return "Allô ?"
  end if
end tel1

on dring1() -- je décide de me laver quand même
  etat2()
  set sortie to my etat3 -- ensuite je ferais mon courrier
  set tel to my tel21 -- et je suis disponible pour répondre à mes copines
end dring1

on etat1()
  set laCopine to ""
  set tel to tel1
  set sonne to defSonne -- pas de réponse à la porte
  set dring to dring1
  set dodo to my etat7 -- on va au lit immédiatement
  set estat to "1-Je glande"
end etat1

------------------ état 2 : dans mon bain
on dring2()
```

```
  sortie() -- quand j'en sors, sortie() pointe ce qu'il convient de faire
end dring2

on tel2(qui) -- si j'attends déjà quelqu'un
  if qui is in mescopines then
		if qui ≠ laCopine then
			return "Un autre jour, d'accord ?"
		else
			return "Mais oui ! Je t'attends !"
		end if
  else
		return "Allô ?"
  end if
end tel2

on tel21(qui) -- si je n'attends personne
  if qui is in mescopines then
		set sortie to my etat4 -- préparons nous !
		set tel to tel2
		set laCopine to qui
		return "D'ici une heure, d'accord ?"
  else
		return "Allô ?"
  end if
end tel21

on dodo2() -- je vais au lit, mais seulement à la fin de mon bain
  if sortie ≠ my etat4 then -- et seulement si je n'attends pas Zézette
		set sortie to my etat7
  end if
end dodo2

on etat2()
  set timer to 60 -- un timer de 60 minutes
  set tel to tel2
  set sonne to defSonne -- pas de réponse à la porte
  set dring to dring2 -- envoie vers l'état 3 ou 4 ou 7 si la nuit est tombée
  set dodo to dodo2
  set estat to "2-Je fais des bulles."
end etat2

------------------ état 3 : dispo et propre
on tel3(qui)
  if qui is in mescopines then
		set laCopine to qui
		etat4() -- je vais me mettre en armure
		return "Je t'attends !"
  else
		return "Allô ?"
  end if
end tel3

on Sonne3(qui)
  if qui is in mescopines then
		etat5() --opérons sur le champ
		set laCopine to qui
		return "Les estampes japonaises ? Par ici !"
  else
```

```
        return "Entrez donc !"
  end if
end Sonne3

on etat3()
  if jour then
        set tel to tel3
        set sonne to Sonne3 -- Je réponds à la porte
        set dring to defDring
        set dodo to my etat7 -- on va dormir immédiatement
        set estat to "3-Je travaille (enfin…)."
  else
        etat7() -- ou je vais me coucher
  end if
end etat3

------------------ état 4 : enfilons notre armure
on dodo4()
  return
end dodo4

on Sonne4(qui)
  if qui = laCopine then
        etat5()
        return "Jouvencelle ! entrez dans mon humble castel !"
  else
        defSonne(qui)
  end if
end Sonne4

on etat4()
  set tel to defTel -- je ne peux plus téléphoner à cause du heaume
  set sonne to Sonne4 -- Je réponds à celle que j'attends
  set timer to 30 -- je lui laisse 30 minutes
  set dring to etat3 -- sinon je me suis fait poser un lapin
  set dodo to defDodo
  set estat to "4-Je mets mon armure et j'attends " & laCopine & "."
end etat4

------------------ état 5 : un moment agréable avec Zézette !
on tel5(qui)
  if qui is in mescopines then
        if qui = laCopine then
              return "Tu n'as pas vraiment besoin du téléphone pour me parler !"
        else
              return defTel(qui) -- soyons galant !
        end if
  else
        return "Allô ?"
  end if
end tel5

on etat5()
  set tel to tel5
  set sonne to defSonne -- Je ne réponds pas à la porte
  set timer to 120 -- ne gâchons pas le boulot !
  set dring to my etat6 -- Quand nos sens sont enfin apaisés…
  set dodo to defDodo
```

```
  set estat to "5-Censuré."
end etat5

------------------ état 6 : dispo mais fatigué
on Sonne6(qui)
  if qui is in mescopines then
        return "Un autre jour, s'il te plaît !"
  else
        return "Entrez donc !"
  end if
end Sonne6

on etat6()
  if jour then
        set laCopine to ""
        set tel to tel2 -- assez de galipettes !
        set sonne to Sonne6 -- Je réponds à la porte
        set dring to defDring
        set dodo to my etat7
        set estat to "6-Après l'effort, le réconfort."
  else
        etat7() -- ou je vais me coucher
  end if
end etat6

------------------ état 7 : un repos bien gagné !
on etat7()
  set timer to 7 * 60 -- sept heures de sommeil, c'est ma dose
  set dring to etat1 --et c'est reparti !
  set sonne to defSonne -- Je ne réponds pas
  set tel to defTel -- Je ne réponds pas
  set dodo to defDodo
  set estat to "7-Zzzzzzzz ..."
end etat7

(* ------------- Pour les tests, on fournit l'interface directe suivante :
 *)
set personnes to mescopines & {"Daniel Varlet", "Steve Jobs", "Quiconque"}

repeat
  set rep to choose from list {"1-Téléphone", "2-Sonnette", "3-Réveil", "4-Jour/nuit", "5-
Terminer"} ¬
        with prompt estat
  try
        set rep to the first item of rep
  on error
        exit repeat
  end try
  if rep begins with "1" then
        choose from list personnes with prompt "Qui est au bout du fil ?"
        display dialog appelTel(first item of result)
  else if rep begins with "2" then
        choose from list personnes with prompt "Qui frappe à la porte ?"
        display dialog sonnePorte(first item of result)
  else if rep begins with "3" then
        dringDring()
  else if rep begins with "4" then
        aubeAuxDoigtsdeRose()
```

```
  else if rep begins with "5" then
          exit repeat
  end if

end repeat
```

Implémentation par scripts-objets.

Il est également possible d'utiliser des scripts-objets pour réaliser ce travail : le polymorphisme remplace alors les pointeurs de fonction sauf pour la sortie de l'état 2. Chaque état est représenté par un objet-script différent. Ils héritent tous du script **deflt** qui permet de recueillir les procédures par défaut.

Une référence à l'état courant est stockée dans la property **ptr**, ce qui permet d'appeler la procédure adéquate par :

```
tel(qui) of ptr  -- appelle la procédure tel() de l'état courant.
```

On passe d'un état à l'autre en appelant le handler **start()** de l'état voulu. Comme la plupart du temps, il suffit de mettre le nouvel état courant dans **ptr**, c'est le handler **start()** de l'objet parent deflt qui est utilisé. Voici le script :

```
(* Automate implémenté par scripts-objets *)

property jour : true -- le flag jour/nuit
property mescopines : {"Zézette", "Marjorie", "Chantal", "Cunégonde"}
property laCopine : "" -- celle que j'attends

property ptr : "" -- l'état en cours

(* Ce script est conçu comme une appli d'arrière-plan qui reçoit des messages par les points d'entrées
suivants :
Ces points d'entrée utilisent immédiatement un pointeur de fonction qui leur fait exécuter l'opération appropriée
*)

-- 1 - Appel téléphonique
on appelTel(qui)
  tel(qui) of ptr
end appelTel
-- 2 - On sonne à la porte d'entrée
on sonnePorte(qui)
  sonne(qui) of ptr
end sonnePorte
-- 3 - Un certain délai est écoulé
on dringDring()
  dring() of ptr
end dringDring
-- 4 - Le jour fait place à la nuit ou le contraire.
on aubeAuxDoigtsdeRose()
  if jour then
          dodo() of ptr
  else
          dring() of ptr
  end if
```

```
  set jour to not jour
end aubeAuxDoigtsdeRose

------------ les procédures par défaut sont dans ce script ---------
script deflt
  on start()
        set my ptr to me
  end start

  on tel(qui)
        return "Par suite de l'encombrement des lignes, votre appel ne peut aboutir."
  end tel

  on sonne(qui)
        return "Il n'y a visiblement personne."
  end sonne

  on dring()
  end dring

  on dodo()
  end dodo

end script

------------------- état 1 : disponible mais sale
script etat1
  property parent : deflt
  property estat : "1-Je glande"

  on start()
        set laCopine to ""
        continue start()
  end start

  on tel(qui)
        if qui is in mescopines then
              tell my etat2 to start() -- je file prendre mon bain
              set laCopine to qui
              return "D'ici une heure, d'accord ?"
        else
              return "Allô ?"
        end if
  end tel

  on dring() -- je décide de me laver quand même
        tell my etat21 to start()
  end dring

  on dodo()
        tell my etat7 to start()
  end dodo

end script

------------------- état 2 : dans mon bain, j'attends quelqu'un
script etat2
  property parent : deflt
```

```
  property estat : "2-Je fais des bulles."

  on start()
        set timer to 60 -- un timer de 60 minutes
        continue start()
  end start

  on dring() -- c'est l'heure de sortir !
        tell my etat4 to start() -- maintenant en armure !
  end dring

  on tel(qui)
        if qui is in mescopines then
              if qui ≠ laCopine then
                    return "Un autre jour, d'accord ?"
              else
                    return "Mais oui ! Je t'attends !"
              end if
        else
              return "Allô ?"
        end if
  end tel

end script

------------------ état 2 : dans mon bain, je n'attends personne
script etat21
  property parent : deflt
  property estat : "2-Je fais des bulles."
  property sortie : ""

  on start()
        set my sortie to my etat3
        set timer to 60 -- un timer de 60 minutes
        continue start()
  end start

  on dring() -- c'est l'heure de sortir !
        tell sortie to start() -- Je vais faire mon courrier ou au lit
  end dring

  on tel(qui)
        if qui is in mescopines then
              set ptr to my etat2 -- préparons nous !
              set laCopine to qui
              return "D'ici une heure, d'accord ?"
        else
              return "Allô ?"
        end if
  end tel

  on dodo() -- je vais au lit, mais seulement à la fin de mon bain
        set sortie to my etat7
  end dodo

end script

------------------ état 3 : dispo et propre
```

```
script etat3
  property parent : deflt
  property estat : "3-Je travaille (enfin…)."

  on start()
          if not my jour then
                  tell my etat7 to start() -- ou je vais me coucher
          else
                  continue start()
          end if
  end start

  on tel(qui)
          if qui is in mescopines then
                  tell my etat4 to start() -- je vais me mettre en armure
                  set laCopine to qui
                  return "Je t'attends !"
          else
                  return "Allô ?"
          end if
  end tel

  on sonne(qui)
          if qui is in mescopines then
                  tell my etat5 to start() --opérons sur le champ
                  set laCopine to qui
                  return "Les estampes japonaises ? Par ici !"
          else
                  return "Entrez donc !"
          end if
  end sonne

  on dodo()
          tell my etat7 to start()
  end dodo

end script

------------------ état 4 : en armure
script etat4
  property parent : deflt
  property estat : "4-Je mets mon armure et j'attends " & laCopine & "."

  on start()
          set timer to 30 -- je lui laisse 30 minutes
          continue start()
  end start

  on dring()
          tell my etat3 to start() -- je me suis fait poser un lapin
  end dring

  on sonne(qui)
          if qui = laCopine then
                  tell my etat5 to start()
                  return "Jouvencelle ! entrez dans mon humble castel !"
          else
                  continue sonne(qui)
```

```
		end if
	end sonne

end script

---------------- état 5 : un moment agréable avec Zézette !
script etat5
	property parent : deflt
	property estat : "5-Censuré."

	on start()
		set timer to 120 -- ne gâchons pas le boulot !
		continue start()
	end start

	on dring()
		tell my etat6 to start() -- Quand nos sens sont enfin apaisés…
	end dring

	on tel(qui)
		if qui is in mescopines then
			if qui = laCopine then
				return "Tu n'as pas vraiment besoin du téléphone pour me parler !"
			else
				continue tel(qui) -- soyons galant !
			end if
		else
			return "Allô ?"
		end if
	end tel

end script

---------------- état 6 : dispo mais fatigué
script etat6
	property parent : deflt
	property estat : "6-Après l'effort, le réconfort."

	on start()
		if my jour then
			set laCopine to ""
			continue start()
		else
			tell my etat7 to start() -- ou je vais me coucher
		end if
	end start

	on sonne(qui)
		if qui is in mescopines then
			return "Un autre jour, s'il te plaît !"
		else
			return "Entrez donc !"
		end if
	end sonne

	on tel(qui)
		if qui is in mescopines then
			return "Un autre jour, d'accord ?"
```

```
			else
				return "Allô ?"
			end if
	end tel

	on dodo()
			tell my etat7 to start()
	end dodo

end script

----------------- état 7 : un repos bien gagné !
script etat7
	property parent : deflt
	property estat : "7-Zzzzzzzz ..."

	on start()
			set timer to 7 * 60 -- sept heures de sommeil, c'est ma dose
			continue start()
	end start

	on dring()
			tell my etat1 to start() -- et c'est reparti pour une belle journée !
	end dring

end script

------------------------------------

(* mais pour les tests, on fournit l'interface directe suivante : *)
set personnes to mescopines & {"Daniel Varlet", "Steve Jobs", "Quiconque"}
set jour to true
tell etat1 to start()

repeat
	set rep to choose from list {"1-Téléphone", "2-Sonnette", "3-Réveil", "4-Jour/nuit", "5-
Terminer"} ¬
			with prompt (estat of ptr)
	try
			set rep to the first item of rep
	on error
			exit repeat
	end try
	if rep begins with "1" then
			choose from list personnes with prompt "Qui est au bout du fil ?"
			display dialog appelTel(first item of result)
	else if rep begins with "2" then
			choose from list personnes with prompt "Qui frappe à la porte ?"
			display dialog sonnePorte(first item of result)
	else if rep begins with "3" then
			dringDring()
	else if rep begins with "4" then
			aubeAuxDoigtsdeRose()
	else if rep begins with "5" then
			exit repeat
	end if

end repeat
```

Optimisation et magie blanche.

Ici, il est effectivement question d'accélérer l'exécution d'un script. Le truc est assez simple, mais malheureusement le fonctionnement intime du processus nous est resté assez mystérieux[67] . Il est parfois un peu délicat à manier, donc, à employer avec prudence sur des algorithmes déjà bien au point.

L'astuce s'applique lorsque l'on a de grosses listes à traiter, c'est à dire que l'on veut les créer, y ajouter des éléments, les trier ou les traiter d'une manière ou d'une autre. Voyons tout d'abord l'influence des diverses syntaxes possibles. Commençons par tester les différentes manières de construire une liste en y ajoutant un élément à la fois, soit par la fin, soit par le début. Les performances respectives sont mises en commentaire.

```
set liste to {}

set aTime0 to current date
repeat 5000 times
  set liste to liste & "a" -- 42 s, inimaginable ! c'est un ordinateur ? vraiment ?
  --set the end of liste to "a" -- 18 s
  --copy "a" to the end of liste -- 17 s
  set liste to {"a"} & liste -- 39 s
  --set the beginning of liste to "a" -- 18 s
  --copy "a" to the beginning of liste -- 18 s
end repeat

set aTime1 to current date
set laps to aTime1 - aTime0
display dialog laps
```

Il apparaît déjà que la concaténation avec l'opérateur & est le pire choix possible. Les autres syntaxes sont légèrement préférables.

Maintenant, déclarons la variable **liste** comme property d'un script-objet créé spécialement pour cela.

```
script sto
  property liste : {}
end script

set aTime0 to current date
repeat 5000 times
  set sto's liste to sto's liste & "a" -- 40 s , :-((((
  --set the end of sto's liste to "a" -- < 1s , Aaaaaaaaah !
  --copy "a" to the end of sto's liste -- < 1s , Oooooooh !
  --set sto's liste to {"a"} & sto's liste -- 29 , Beuaaaaaark !
  --set the beginning of sto's liste to "a" -- < 1s , Ouuuiiiiiiii !
  --copy "a" to the beginning of sto's liste -- < 1s , … Encoooooooooooooore !!!!
end repeat
```

---

[67] Il est néanmoins probable qu'un certain nombre de contrôles et de vérifications dans la gestion des variables sont abandonnés lorsqu'on passe la frontière d'un objet-script. Cette simplification, rendant la tâche du gestionnaire de mémoire moins lourde pourrait expliquer l'accélération.

```
set aTime1 to current date
set laps to aTime1 - aTime0
display dialog laps
```

L'opération améliore considérablement les performances sauf si l'on utilise l'opérateur de concaténation & (pas beaucoup de changement dans ce cas). Comme le compteur de temps est imprécis, il faut d'ailleurs faire tourner le test sur 10 000 ou 20 000 pas de boucle pour constater la différence avec une meilleure précision.

Il en est de même sur les manipulations d'items à l'intérieur de la liste (sans allongement ni rétrécissement de celle-ci).

```
set liste to {}

repeat 5000 times --créons une liste de test
  set liste to {"a"} & liste
end repeat

set aTime0 to current date

set n to (count of liste) div 2 --et tripotons la
repeat with i from 1 to n
  set item i of liste to item (i + n) of liste
end repeat

set aTime1 to current date
set laps to aTime1 - aTime0
display dialog laps -- 17 s, je rêve ! que dis-je, j'hallucine !
```

Cette routine prend 17 secondes pour déplacer la seconde moitié des éléments de la liste dans sa première moitié, soit 2 500 déplacements d'items. En utilisant un objet-script :

```
script sto
  property liste : {}
end script

repeat 5000 times
  set the end of sto's liste to "a"
end repeat

set aTime0 to current date

set n to (count of sto's liste) div 2
repeat with i from 1 to n
  set sto's liste's item i to sto's liste's item (i + n)
end repeat

set aTime1 to current date
set laps to aTime1 - aTime0
display dialog laps -- < 1 s ,  ah bon !
```

Le temps d'exécution est inférieur à la seconde. Comme on dit, il n'y a pas photo …

Prenons l'exemple d'une routine de tri vanilla déjà excellente en elle-même :

```
(* tri par tas (arbre binaire complet ordonné)
Ce tri figure parmi les plus efficaces et n'exige aucune mémoire supplémentaire.
Algorithme :  L. Sebilleau
*)

property Compare : "" -- pointeur de fonction de comparaison

on run
  set maListe to {1, 5, 15, 2, 7, 3, 23, 125, 4, 15, 8, 7, 6, 5, 27, 35, 12, 22, 16, 47}

  repeat 5 times
        set maListe to maListe & maListe  -- pour 640 éléments
  end repeat

  set aTime0 to current date
  my heapsort(maListe, my numcmp)  -- ou  my numcmp2 pour un tri descendant
  set aTime1 to current date
  set laps to aTime1 - aTime0
  display dialog laps  -- 20 secondes, bof … (381 secondes pour 2 560 éléments)
end run

on heapsort(laListe, fcmp)
  set Compare to fcmp
  set ct to length of laListe
  if ct ≤ 1 then return laListe

  -- création du tas  par insertion fictive des éléments présents dans la liste : ils sont
  --"phagocytés" un à un par la croissance du tas dont la dimension est i.
  repeat with i from 2 to ct
        set j to i div 2  -- le premier parent du nouvel inséré
        set k to i
        set elem to item k of laListe
        repeat while k > 1
              if my Compare(item j of laListe, elem) then
                    set item k of laListe to item j of laListe
                    set k to j
                    set j to j div 2
              else
                    exit repeat
              end if
        end repeat
        set item k of laListe to elem  -- installé dans sa position finale
  end repeat

  (* destruction du tas : l'extraction répétée donne la liste triée à l'envers :
 comme le tas diminue par la fin de la liste, on peut réinsérer les éléments extraits à rebours
dans les emplacements libérés. Ce retournement explique pourquoi un utilise un tas max pour
trier en ordre croissant et vice-versa. *)

  repeat with i from ct to 2 by -1
        set elem to item i of laListe  -- diminution du tas
        set item i of laListe to item 1 of laListe  -- extraction de la racine qui est
        -- recopiée dans la position libérée

        -- réorganisation du tas en réinsérant l'élément de queue à la racine
        set j to 1
```

```
		set k to j * 2
		repeat while k < i
			if (k + 1) < i then
				if my Compare(item k of laListe, item (k + 1) of laListe) then
					set k to k + 1
				end if
			end if
			if my Compare(elem, item k of laListe) then
				set item j of laListe to item k of laListe
				set j to k
				set k to k * 2
			else
				exit repeat
			end if
		end repeat
		set item j of laListe to elem
	end repeat
	return laListe  --  facultatif : heapsort travaille sur la liste originale
end heapsort

------------ les fonctions de comparaison ------------
on numcmp(n1, n2)-- tri ascendant
	return n1 < n2
end numcmp

on numcmp2(n1, n2)-- tri descendant
	return n1 > n2
end numcmp2
```

En plaçant la liste dans une propriété d'un script-objet, on accélère la routine de façon époustouflante (facteur 50) rejoignant ainsi et dépassant même les performances de certains OSAX...

```
(* tri par tas (arbre binaire complet ordonné)
Ce tri figure parmi les plus efficaces et n'exige aucune mémoire supplémentaire.
Algorithme :  L. Sebilleau
*)

on run
	set maListe to {1, 5, 15, 2, 7, 3, 23, 125, 4, 15, 8, 7, 6, 5, 27, 35, 12, 22, 16, 47}
	
	repeat 7 times
		set maListe to maListe & maListe  --  pour 2 560 éléments au lieu de 640
	end repeat
	
	set aTime0 to current date
	my heapsort(maListe, my numcmp)
	set aTime1 to current date
	set laps to aTime1 - aTime0
	display dialog laps -- 6 secondes seulement. 60 fois plus rapide que la précédente
end run

on heapsort(laListe, fcmp)
	set ct to length of laListe
	if ct ≤ 1 then return laListe
```

```
script lstock
	property liste : laListe
	property Compare : fcmp -- le pointeur de fonction est reporté ici
end script
-- le reste est identique

-- création du tas  par insertion fictive des éléments présents dans la liste : ils sont
--"phagocytés" un à un par la croissance du tas dont la dimension est i.
repeat with i from 2 to ct
	set j to i div 2 -- le premier parent du nouvel inséré
	set k to i
	set elem to item k of lstock's liste
	repeat while k > 1
		if lstock's Compare(item j of lstock's liste, elem) then
			set item k of lstock's liste to item j of lstock's liste
			set k to j
			set j to j div 2
		else
			exit repeat
		end if
	end repeat
	set item k of lstock's liste to elem -- installé dans sa position finale
end repeat

(* destruction du tas : l'extraction répétée donne la liste triée à l'envers :
 comme le tas diminue par la fin de la liste, on peut réinsérer les éléments extraits à rebours
dans les emplacements libérés. Ce retournement explique pourquoi un utilise un tas max pour
trier en ordre croissant et vice-versa. *)

repeat with i from ct to 2 by -1
	set elem to item i of lstock's liste -- diminution du tas
	set item i of lstock's liste to item 1 of lstock's liste -- extraction de la racine qui est
	-- recopiée dans la position libérée

	-- réorganisation du tas en réinsérant l'élément de queue à la racine
	set j to 1
	set k to j * 2
	repeat while k < i
		if (k + 1) < i then

			if lstock's Compare(item k of lstock's liste, item (k + 1) of lstock's liste)
then
				set k to k + 1
			end if
		end if
		if lstock's Compare(elem, item k of lstock's liste) then
			set item j of lstock's liste to item k of lstock's liste
			set j to k
			set k to k * 2
		else
			exit repeat
		end if
	end repeat
	set item j of lstock's liste to elem
end repeat
return lstock's liste -- facultatif : heapsort travaille sur la liste originale
end heapsort
```

```
----------- les fonctions de comparaison ------------
on numcmp(n1, n2)-- tri ascendant
  return n1 < n2
end numcmp

on numcmp2(n1, n2)-- tri descendant
  return n1 > n2
end numcmp2
```

Pour éviter les lourdeurs de répétitions dans le script, il est effectivement possible d'indiquer la cible par défaut par un **Tell** . Mais attention... ce **Tell** doit cibler le script-objet lui-même, comme ceci :

```
------------------------------------------ Extrait...
tell lstock
  repeat with i from 2 to ct
	set j to i div 2 -- le premier parent du nouvel inséré
	set k to i
	set elem to item k of its liste
	repeat while k > 1
		if Compare(item j of its liste, elem) then
			set item k of its liste to item j of its liste
			set k to j
			set j to j div 2
		else
			exit repeat
		end if
	end repeat
	set item k of its liste to elem -- installé dans sa position finale
  end repeat
...
------------------------------------------ Extrait...
```

et non pas la propriété de l'objet-script comme ceci :

```
------------------------------------------ Extrait...
tell lstock's liste
  repeat with i from 2 to ct
	set j to i div 2 -- le premier parent du nouvel inséré
	set k to i
	set elem to item k of it
	repeat while k > 1
		if Compare(item j of it, elem) then
			set item k of it to item j of it
			set k to j
			set j to j div 2
		else
			exit repeat
		end if
	end repeat
	set item k of it to elem -- installé dans sa position finale
  end repeat
...
------------------------------------------ Extrait...
```

Sinon adieu le gain de vitesse. La magie est parfois abracadabrante...

Enfin, pour terminer, mentionnons que l'on peut avoir intérêt à transformer les chaînes de caractères en listes pour accélérer les choses (car la magie de l'objet-script n'opère pas sur les chaînes !).

Dans l'exemple suivant, nous souhaitons remplacer certains caractères par d'autres dans une chaîne **entree**. Pour ce faire, nous construisons une table de conversion, puis nous l'appliquons à la chaîne :

```
property listea : "abcdefg" -- les caractères à remplacer
property listeb : "ABCDEFG" -- par ceux-ci

set aTime0 to current date

-- Construit la table de conversion
set table to ""

repeat with i from 0 to 255
  set x to (offset of ASCII character i in listea)
  if (x = 0) then
        set table to table & (ASCII character i)
  else
        set table to table & (character x of listeb)
  end if
end repeat

set aTime1 to current date
set laps to aTime1 - aTime0
display dialog laps -- 14 secs
set aTime0 to current date

-- Conversion
set entree to "12345ACFGTZhsdk sjhdlkf hskdhfks hqkfjhd hkfljhskqjhf hskdjfhklsjhgk hdkfgjhkd
fhgkjh "

set fin to count characters of entree
set sortie to "" as string

repeat with i from 1 to fin
  set sortie to sortie & (character ((ASCII number (character i of entree)) + 1) of table)
end repeat

set aTime1 to current date
set laps to aTime1 - aTime0
display dialog laps -- 2 secs
--
log sortie
```

Ce script est tout sauf rapide, mais l'analyse montre que la construction de la table de conversion prend 14 secondes, bien plus que la conversion de la chaîne. C'est pourquoi on peut essayer l'idée suivante où l'on transforme les deux chaînes **listea** et **listeb** en listes pour accélérer la construction de la table :

```
property listea : "abcdefg" -- les caractères à convertir
property listeb : "ABCDEFG" -- les caractères de remplacement

on creaStock()
  script
```

```
            property table : {}
            property la : {}
            property lb : {}
    end script
end creaStock

set sto to creaStock() -- création de l'objet-script de stockage

set aTime0 to current date

-- Construit la table de conversion sous forme de liste
set sto's table to {}
set sto's la to characters 1 thru -1 of listea
set sto's lb to characters 1 thru -1 of listeb
set n to the count of listea

repeat with i from 0 to 255
  set tmp to (ASCII character i) -- par défaut aucune conversion
  repeat with x from 1 to n
            if (sto's la's item x = tmp) then
                      set tmp to sto's lb's item x
                      exit repeat
            end if
  end repeat
  set the end of sto's table to tmp
end repeat

set aTime1 to current date
set laps to aTime1 - aTime0
display dialog laps -- 5 secs
set aTime0 to current date

-- Conversion
set entree to "12345ACFGTZhsdk sjhdlkf hskdhfks hqkfjhd hkfljhskqjhf hskdjfhklsjhgk hdkfgjhkd 
fhgkjh "

set fin to count characters of entree
set sortie to ""

repeat with i from 1 to fin
  set sortie to sortie & (item ((ASCII number (character i of entree)) + 1) of sto's table)
end repeat

set aTime1 to current date
set laps to aTime1 - aTime0
display dialog laps -- 2 secs

log sortie
```

On constate que le temps de construction de la table a été divisé par trois. Le temps de conversion n’a pas bougé. (On peut imaginer d’utiliser la même technique sur la chaîne **entree** et la chaîne **sortie** pour les traiter toutes les deux comme des listes, mais le charme n’opère pas de la même manière, sans doute à cause de la fonction ASCII).

Rappelons pour finir l’importante différence qui existe dans le traitement des

références que nous avons signalé page 52 : des algorithmes peuvent fonctionner parfaitement dans une configuration normale et renvoyer des erreurs lorsque les listes sont placées dans un objet-script. Utiliser des scripts-objets accélérateurs réclame une programmation plus rigoureuse.

Annexes.

Types de base et formats.

Scripts d’exemple complets.[68]

[68] Archive à télécharger séparément.

Types, conversions et coercions.

Nous avons plusieurs fois évoqué le type d'une variable. Il est temps de préciser en détail ce que c'est.

Rappel.

Nous avons vu que le micro-processeur ne distinguait pas un octet censé représenter un point sur un image d'un autre censé représenter un caractère typographique.

Cette situation présente évidemment un avantage, celui de la souplesse, mais le gros inconvénient de ne pas aider le programmeur à détecter les erreurs qu'il fait dans le traitement de ses données. Pour donner un exemple, supposons que nous additionnions deux nombres en écrivant en assembleur quelque chose d'équivalent à :

```
set x to y + z
```

L'idée paraît tout à fait innocente et fonctionne parfaitement tant que les nombres sont de même nature. Si l'un d'eux est un nombre en virgule flottante et l'autre un entier, le résultat sera probablement erroné, mais pas toujours. Erreur probablement difficile à trouver et corriger.

C'est la raison pour laquelle les langages introduisent la notion de type. De la même manière qu'on ne peut additionner des torchons et des serviettes (même si l'addition des nombres est toujours possible), on pourra composer ou affecter les variables entre elles si et seulement si elles sont de même type, ou d'un type compatible.

Il existe deux "politiques" de typage, appelées typage fort et typage faible. Le deuxième est moins contraignant mais au prix de quelques inconvénients. Mais de toutes manières, le typage faible n'élimine pas la notion de type.

Typage fort.

Comme c'est le plus simple, nous commencerons par là. Le typage fort consiste à s'imposer d'attribuer un type à toutes les variables que l'on définit et même un type aux valeurs retournées par les fonctions.

Toutes les variables doivent donc être déclarées préalablement à leur utilisation et ne peuvent recevoir que des valeurs de type correspondant. Par exemple :

char lettre, mot[128], *reference; [69]

peut se lire : la variable lettre contient un caractère (char = caractère sur 1 octet), la variable mot est un tableau de 128 caractères, et reference est une référence à une variable de type char.

[69] Nous utilisons ici la syntaxe du C car AppleScript ne possède pas de déclaration.

Sachant que char désigne un caractère codé sur un octet, on voit que le compilateur dispose ainsi de toutes les informations nécessaires pour organiser la mémoire.

Il va ainsi réserver un octet pour lettre, 128 pour mot, et 4 (une adresse 32 bits) pour reference.

En dehors de cela, il pourra détecter d'éventuelles erreurs d'utilisation (fautes d'étourderie en général) et donner des messages d'erreur à la compilation si l'on tente de stocker dans lettre une valeur de type entier par exemple. La raison s'explique par le fait que les entiers ont des valeurs occupant deux ou quatre octets en général. Si l'on ne dispose que d'un octet, le résultat pourra être correct si l'entier est assez petit, sinon on perdra de l'information.

Conversions, définitions de types.

Cette règle est un peu trop rigide, car il existe des correspondances classiques entre types qui n'engendrent pas d'erreur : par exemple un caractère et son code ASCII. On peut utiliser l'un pour l'autre sans inconvénient car la représentation interne est la même et utilise un octet dans les deux cas.

Dans d'autres cas la conversion d'un type à l'autre peut se faire de plusieurs manières ou ne réussit pas dans tous les cas. Le compilateur réclame alors du programmeur une instruction particulière signalant qu'il entend bien réaliser l'opération de conversion à ses risques et périls. Un exemple de cette situation peut être donné avec l'âge d'une personne : un octet suffit pour enregistrer des âges allant jusqu'à 255 ans ce qui paraît largement suffisant. Si on l'obtient comme résultat d'un calcul sur des entiers, le compilateur réclamera une confirmation que la conversion, pour des raisons qu'il ne peut connaître (ici la limitation de la durée de la vie humaine) réussira toujours.

Notez que nous ne parlons pas ici de conversions qui réclament un calcul ou une modification de la donnée, mais qui signalent un changement de représentation : par exemple, considérer un caractère comme un tableau de huit bits permet, en forçant le bit n°5 à 0 ou 1 par une opération booléenne, de modifier la casse en exploitant une particularité du code ASCII (autrement dit, on peut ainsi facilement passer des majuscules aux minuscules et vice versa). Donc on considère temporairement les caractères comme des tableaux de bits pour effectuer cette opération, mais sans changer leur valeur.

Il va sans dire que l'on ne peut convertir n'importe quel type en n'importe quel autre, mais uniquement ceux qui partagent la même représentation ou à peu près.

La coercion ou coercition AppleScript relève de la même idée, mais elle peut aussi entraîner une modification ou un réarrangement des données. Elle est donc plus large qu'un simple changement de représentation.

Enfin, signalons que les langages à typage fort permettent de définir ses propres types à partir des types de base. C'est une option facultative mais qui permet au programmeur de laisser au compilateur plus de latitude pour contrôler ce qu'il fait.

Par exemple, un programme qui gère des stocks de torchons et de serviettes

peut parfaitement se servir d'un type de base comme l'entier. Mais dans ce cas, le compilateur autorisera sans sourciller l'addition des torchons et des serviettes. Si l'on définit un type torchon et un type serviette, tous les deux basés sur le type "entier", le compilateur sera à même de détecter l'erreur.

Typage faible.

Dans le typage faible, ce n'est pas la variable qui est typée, mais sa valeur seulement. Dans une variable AppleScript, on peut stocker n'importe quoi, mais on ne peut pas additionner des chaînes de caractères et des nombres par exemple. Il s'ensuit que des contrôles de type sont toujours effectués car certaines opérations sont impossibles.

Dans le typage faible, le contrôle n'est pas effectué a priori par le compilateur, mais au cours de l'exécution. En AppleScript, le langage "fait de son mieux" pour effectuer l'opération demandée : si les valeurs à composer ne sont pas de même type, il tente de trouver dans son catalogue de conversions quelque chose qui lui permette de ramener les deux valeurs au même type. Si cette tentative échoue, le programme est interrompu avec l'erreur classique : … ne peut renvoyer les données dans le type demandé.

Les coercions, qui sont des conversions explicitement demandées par le programmeur, permettent de "guider" le langage au cas où il ne peut se débrouiller tout seul ou prendrait une mauvaise option.

Typage et Objets.

Quels sont les types ? Il y a d'abord ce qu'on appelle les types de base.

- Nombres entiers.
- Nombres en virgule flottante.
- Caractères alphanumériques.
- Valeurs booléennes.
- Dates.

On retrouve ces types de base dans tous les langages et dans tous les logiciels de base de données. Ce qui n'est pas surprenant, car à l'exception des dates, le µP possède des instructions pour traiter directement ce type de données.

Ensuite, il y a les types composés. Par exemple, en AppleScript, le type "string" ou "text" qui désigne les chaînes de caractères, est une suite de caractères alphanumériques. Il est donc basé sur le type "character". (Ce dernier type n'existe pas en AppleScript pour des raisons de simplicité).

Record et list sont présentés dans la documentation sous la même forme que les types de base, mais ce ne sont pas des types à proprement parler : ce sont des structures de données qui peuvent entrer dans la définition d'un type. Par exemple, le handler "display dialog" des compléments de pilotage renvoie un [record

contenant deux "string" et une valeur booléenne]. La partie entre crochets désigne le type de ce résultat.

Il existe enfin une foule d'autres types que l'on peut trouver dans les dictionnaires. En toute rigueur, il s'agit de classes, autrement dit, de concepts plus riche que les types, mais ils en sont très proches. Sachez en particulier que tous les langages considèrent qu'un objet (au sens informatique du terme) a pour type un type spécial qui porte le nom de sa classe.. Tous les noms de classes sont donc des noms de types.

Si vous consultez la définition d'une classe, vous y trouverez une liste de propriétés, chacune ayant un type ou une classe bien définis.. Ceci vous indique sous quelle forme est fournie l'information et quelles sont les techniques appropriées pour les traiter.

Vous ne pouvez pas définir vos propres types en AppleScript, même avec des scripts-objets. En effet, contrairement à ce qui se passe dans d'autres langages, les scripts-objets ont tous le même type "script" (ou appartiennent à la même classe).

Représentations courantes.

Booléens.

Les booléens sont des éléments qui peuvent prendre deux valeurs seulement ordinairement désignées par vrai et faux. Or justement, les bits qui constituent l'élément de base de la mémoire d'un ordinateur ont exactement les propriétés d'un booléen.

Un octet peut donc être considéré comme un tableau contenant huit booléens auquel le programmeur attribuera un sens en fonction de son application. Par exemple, un booléen de ce genre peut être utilisé pour distinguer les moments où votre éditeur AppleScript est en mode d'enregistrement ou non. Selon sa valeur, il enregistrera vos actions dans le script courant ou non.

Les µP disposent d'un jeu d'instructions très complet pour manipuler le contenu de leurs registres en les considérant comme des tableaux de valeurs booléennes.

D'abord les opérations ET, OU inclusif, OU exclusif et NEGation, qui permettent de composer des valeurs booléennes entre elles. Ensuite tout une série d'instructions qui permettent très rapidement de faire des tests sur des valeurs booléennes et d'effectuer des actions en conséquence.

En AppleScript comme dans la plupart des langages, il est impossible d'accéder directement au niveau bit. Il donc impossible de stocker huit valeurs booléennes dans un seul octet. Il est tout aussi impossible de convertir un booléen en un autre type de donnée. Il n'est donc pas très intéressant de savoir exactement quelle représentation ce langage utilise, car on ne peut pas en tirer parti comme en assembleur ou en C.

Nombres.

Représentations binaires.

Le principe adopté pour représenter des nombres consiste à considérer que les bits ont pour valeur 0 ou 1. Dans ces conditions, on peut aussi considérer qu'un octet qui contient huit bits peut représenter un nombre écrit en base 2.

7	6	5	4	3	2	1	0
v7	v6	v5	v4	v3	v2	v1	v0

Le nombre représenté par cet octet sera :
v0 + (v1 * 2) + (v2 * 4) + (v3 * 8) + (v4 * 16) + (v5 * 32) + (v6 * 64) + (v7 * 128)
où v0…v7 peuvent prendre les valeurs 1 ou 0. Par exemple l'octet :

7	6	5	4	3	2	1	0
0	0	0	0	1	0	1	0

représentera le nombre :
0 + (1 * 2) + (0 * 4) + (1 * 8) + (0 * 16) + (0 * 32) + (0* 64) + (0 * 128)
ou encore :
(1 * 2) + (1 * 8) = 10.

On voit donc qu'à chaque bit est associé un coefficient multiplicateur qui est la puissance de 2 associée à sa position : en d'autre termes, le bit numéroté 4 (qui vaut 1 ou 0) sera multiplié par 2 puissance 4 = 16, pour calculer sa contribution au nombre final. C'est pourquoi on numérote les bits dans l'octet (de 0 à 7 pour correspondre exactement à la puissance de 2 qui va avec). Cette numérotation ne correspond à rien dans l'électronique : elle précise simplement la convention utilisée pour les nombres et est universellement utilisée pour désigner un bit précis dans un octet : on parle ainsi du bit 4. Par ailleurs les bits portant des numéros élevés (7 et 6) sont dit bits de poids fort (anglais :most significant bit or high-order bit) et ceux qui portent des numéros faibles (0 ou 1) bits de poids faible (anglais : least significant bit or low order bit).

On pourra vérifier assez aisément qu'on peut représenter ainsi dans un octet des nombres de 0 à 255. Pas mal, mais notoirement insuffisant pour des quantités de choses. La solution consiste bien sûr à juxtaposer des octets pour créer des ensembles plus vastes, comme dans la figure suivante :

15	14	13	12	11	10	9	8
v15	v14	v13	v12	v11	v10	v9	v8

7	6	5	4	3	2	1	0
v7	v6	v5	v4	v3	v2	v1	v0

Les registres des processeurs modernes (quatre octets, voire huit), permettent de traiter en une seule fois des nombres assez importants.

Cette représentation correspond à ce que l'on appelle les entiers non signés (anglais : unsigned integer) dans les langages évolués. Nous allons voir maintenant comment on représente des entiers relatifs.

Représentations signées.

Lorsqu'on a besoin d'utiliser des entiers munis d'un signe, on utilise une représentation très proche de la précédente. La différence est que le bit de poids fort représente le signe et non une partie de la valeur. La convention (universelle) est que ce bit prend la valeur zéro si le signe est + et la valeur 1 si le signe est -.

Ensuite, on pourrait se dire que les nombres négatifs se déduisent des positifs en changeant simplement le signe. Si :

7	6	5	4	3	2	1	0
0	0	0	0	1	0	1	0

vaut 10, alors -10 se représente par :

7	6	5	4	3	2	1	0
1	0	0	0	1	0	1	0

Ce serait logique, mais un autre argument entre en ligne de compte. En procédant ainsi, on serait amené à donner au µP deux jeux d'instructions arithmétiques distincts, un pour les entiers naturels (sans signe) et un autre pour les entiers signés. En effet, lorsque j'additionne les deux octets précédents, cela correspond aux deux sommes :

10 - 10 = 0 (nombres signés), et 10 + 138 = 148 (nombres sans signe).

On utilise donc une autre convention de représentation des nombres négatifs appelée "en complément à deux" qui permet de tourner cet inconvénient : dans cette représentation, l'addition et la soustraction ordinaire donnent un résultat correct qu'il s'agisse de nombres signés ou non. Le nombre négatif correspondant à un nombre positif est obtenu en inversant tous les bits et en ajoutant 1. Une autre manière de voir les choses est de considérer que les nombres négatifs sont obtenus pas soustraction successive à partir du zéro : - 1 = 0 - 1

Comme les registres et les cases mémoires fonctionnent comme des totalisateurs journaliers (c'est à dire qu'ils bouclent indéfiniment sur la même série de valeurs et reprennent au départ lorsqu'on atteint la limite), retirer 1 à l'octet 0 0 0 0 0 0 0 0 donne la valeur maximum 1 1 1 1 1 1 1 1 = -1 . On obtient - 2 en retranchant encore 1, ce qui donne 1 1 1 1 1 1 1 0, etc…

De cette manière, l'hypothèque précédente est levée, car l'addition de mes deux octets ne représente plus 10 -10 en nombres signés mais 10 - 118 = -108.

Cette astucieuse disposition ne permet malheureusement pas d'avoir une seule instruction pour la multiplication et la division. En assembleur, le programmeur est obligé de faire un choix selon la représentation qu'il utilise.

Comment signe-t-on les nombres utilisant plus d'un octet ? On place le signe dans le bit de poids fort de l'octet de poids fort[70] .

En AppleScript, les valeurs de type integer sont des entiers signés.

BCD.

Pour compléter ce panorama, citons une représentation numérique qui a eu son

[70] Cette présentation se voulant succincte, nous n'aborderons pas le reste de l'arithmétique binaire, en particulier la question des retenues et des débordements.

succès dans le passé. Elle avait pour but de simplifier les conversions de nombres binaires vers des chaînes de caractères destinées à l'affichage. En effet, lorsque vous voyez un nombre comme 125 à l'écran, vous ne voyez pas directement l'octet qui contient ce nombre mais la chaîne constituée des caractères 1, 2 et 5. Un programme qui manipule des nombres est donc amené en permanence à convertir des chaînes saisies en nombres binaires et vice-versa.

La conversion d'un nombre en base 2 vers un nombre en base 10 n'est pas complètement évidente. La représentation BCD consiste à stocker chaque chiffre d'un nombre en base 10 dans un demi octet. Un demi octet permettant de représenter les nombres de 0 à 15, c'est largement suffisant pour représenter un chiffre de 0 à 9. Les valeurs supplémentaires sont considérées comme dépourvues de sens.

Pour convertir un de ces demi-octets en chiffre destiné à l'affichage, il suffit de lui ajouter 48 : on obtient ainsi le code ASCII du chiffre représenté. De la même manière, il est très facile de transformer un chiffre en nombre binaire BCD en lui retirant 48.

En BCD, notre nombre 125 sera donc représenté sur deux octets :

0000 0001 0010 0101

alors qu'en binaire, un seul suffirait :

0111 1101

Réels.

Les nombres dits réels (anglais : real or float) permettent de représenter des nombres décimaux. Ils sont toujours constitués de deux nombres entiers signés : le premier (la mantisse) est le nombre lui-même, dépourvu de virgule. Le second (exposant) est la puissance de 10 par lequel il faut multiplier la mantisse pour rétablir la position correcte de la virgule. Pour ceux qui connaissent, c'est le décalque exact de la "notation ingénieur" des calculettes.

La mantisse occupe typiquement 6 octets et l'exposant 1. L'intérêt de cette représentation est de pouvoir représenter des nombres très élevés en acceptant une précision limitée de la mantisse (chiffres significatifs). Représenter un nombre de l'ordre de 10 puissance 128 (128 zéros) ne demanderait pas 64 octets, mais environ 48 (je n'ai pas fait le calcul exact) si on le représentait en binaire car chaque octet ne peut enregistrer que deux ou trois chiffres. On voit donc l'économie réalisée.

Tous les processeurs modernes disposent d'un coprocesseur intégré, c'est à dire d'un circuit destiné à réaliser directement les opérations arithmétiques courantes sur ce type de nombre. C'est donc assez rapide, mais tout de même nettement moins que les opérations sur les entiers (signés ou pas). Il convient donc d'éviter le type réel lorsqu'on n'en n'a pas vraiment besoin, par exemple pour des index de boucle ou des numéros de fiche.

AppleScript connaît le type real tout comme le type integer, mais il connaît aussi un type number qui est un fourre-tout pouvant contenir soit l'un soit l'autre. Si

l'on veut s'assurer qu'une valeur sera bien considérée comme integer, il vaut mieux effectuer une coercion, par exemple :

```
set x to 1 as integer
```

Ordre des octets dans un nombre.

Le paragraphe suivant signale une particularité dont vous entendrez peut-être parler, mais qui concerne rarement le programmeur.

Nous avons évoqué l'utilisation de plusieurs octets pour représenter un nombre en binaire. Ces octets ne peuvent évidemment être pris dans n'importe quel ordre :

0000 0000 0000 0001 représente 1 et 0000 0000 0001 0000 représente 256.

Lorsque le processeur charge quatre octets d'un coup dans l'un de ses registres, on voit que l'ordre des octets en mémoire a de l'importance. On pourrait penser que tous les processeurs vont prendre le premier octet et le charger dans la partie "la plus à gauche" de son registre (poids fort) et continuer ainsi dans l'ordre "logique".

C'est effectivement le cas de tous les processeurs Motorola[71] . Les processeurs Intel font l'inverse : le premier octet rencontré en mémoire est l'octet de poids faible et non l'octet de poids fort comme on pourrait s'y attendre.[72]

Ceci commence à nous concerner sérieusement lorsque des machines communiquent par réseau, car les données voyagent toujours octets par octets. Donc lire les octets en séquence pour les envoyer dans un tuyau peut conduire à des erreurs si des groupes d'octets sont des nombres entiers et sont lus par des processeurs de marque différente de ceux qui les ont émis. Les adresses IP par exemple, sont des entiers sur quatre octets.

Caractères & chaînes.

Code ASCII.

Le code Ascii est une convention de représentation des caractères alphanumériques sur un octet. C'est pratiquement la seule utilisée sur les ordinateurs actuellement[73] . Malheureusement, il en existe des variantes.

Cette convention a initialement été définie pour les télétypes ou telex. Pour ceux qui ne sauraient pas ce que c'est, disons qu'il s'agit de deux machines à écrire reliées entre elles par une ligne analogique comme le téléphone. Lorsqu'on tape une

---

[71] En anglais cette convention est désignée comme "high endian numbers"
[72] Convention "low endian numbers"
[73] On ne rencontre plus guère son concurrent EBCDIC sauf dans les applications bancaires où il persiste pour des raisons purement historiques.

touche sur l'une des machines, un code est transmis à l'autre qui répète l'action transmise. Chacun de ces codes était constitué d'un octet, dont le bit de poids fort était réservé au contrôle de parité qui est une technique rudimentaire de contrôle des erreurs de transmission. Si bien que la table originale ne contenait que 128 codes distincts. En passant dans les ordinateurs, les 128 autres codes ont été réutilisés. Voici un exemplaire de la table des codes ASCII :

ASCII	Chr	ASCII	Chr	ASCII	Chr	ASCII	Chr	ASCII	Chr	ASCII	Chr	ASCII	Chr	ASCII	Chr	ASCII	Chr	ASCII	Chr
0	NUL	26	SUB	52	4	78	N	104	h	130	Ç	156	ú	182	∂	208	-	234	Í
1	SOH	27	ESC	53	5	79	O	105	i	131	É	157	ù	183	Σ	209	—	235	Î
2	STX	28	FS	54	6	80	P	106	j	132	Ñ	158	û	184	Π	210	"	236	Ï
3	ETX	29	GS	55	7	81	Q	107	k	133	Ö	159	ü	185	π	211	"	237	Ì
4	EOT	30	RS	56	8	82	R	108	l	134	Ü	160	†	186	∫	212	'	238	Ó
5	ENQ	31	US	57	9	83	S	109	m	135	á	161	°	187	ª	213	'	239	Ô
6	ACK	32	spc	58	:	84	T	110	n	136	à	162	¢	188	º	214	÷	240	
7	BEL	33	!	59	;	85	U	111	o	137	â	163	£	189	Ω	215	◊	241	Ò
8	BS	34	"	60	<	86	V	112	p	138	ä	164	§	190	æ	216	ÿ	242	Ú
9	HT	35	#	61	=	87	W	113	q	139	ã	165	•	191	ø	217	Ÿ	243	Û
10	LF	36	$	62	>	88	X	114	r	140	å	166	¶	192	¿	218	⁄	244	Ù
11	VT	37	%	63	?	89	Y	115	s	141	ç	167	ß	193	¡	219	¤	245	ı
12	FF	38	&	64	@	90	Z	116	t	142	é	168	®	194	¬	220	‹	246	ˆ
13	CR	39	'	65	A	91	[	117	u	143	è	169	©	195	√	221	›	247	˜
14	SO	40	(	66	B	92	\	118	v	144	ê	170	™	196	ƒ	222	ﬁ	248	¯
15	SI	41	)	67	C	93	]	119	w	145	ë	171	´	197	≈	223	ﬂ	249	˘
16	DLE	42	*	68	D	94	^	120	x	146	í	172	¨	198	∆	224	‡	250	˙
17	DC1	43	+	69	E	95	_	121	y	147	ì	173	≠	199	«	225	·	251	˚
18	DC2	44	,	70	F	96	`	122	z	148	î	174	Æ	200	»	226	‚	252	¸
19	DC3	45	-	71	G	97	a	123	{	149	ï	175	Ø	201	…	227	„	253	˝
20	DC4	46	.	72	H	98	b	124	|	150	ñ	176	∞	202	nbs	228	‰	254	˛
21	NAK	47	/	73	I	99	c	125	}	151	ó	177	±	203	À	229	Â	255	ˇ
22	SYN	48	0	74	J	100	d	126	~	152	ò	178	≤	204	Ã	230	Ê		
23	ETB	49	1	75	K	101	e	127	▯	153	ô	179	≥	205	Õ	231	Á	Decimal	
24	CAN	50	2	76	L	102	f	128	Ä	154	ö	180	¥	206	Œ	232	Ë	Format	
25	EM	51	3	77	M	103	g	129	Å	155	õ	181	µ	207	œ	233	È		

On peut voir sur cette table que les 32 premiers "caractères" ne sont pas des caractères typographiques. Ils ne déclenchaient pas une frappe sur la machine distante, mais une action ou servaient à délimiter les messages. La liste des caractères "imprimables" commence donc à 32 spc (l'espace).

Par exemple 13 CR (= carriage Return) est le retour à la ligne ou retour chariot, 10 LF (line feed) est le passage à la ligne suivante. Un simple CR permettait de surcharger les caractères déjà écrits sur la ligne précédente. Pour un véritable aller à la ligne, il fallait spécifier le couple CR LF. C'est toujours comme cela que se présentent les textes au sens de Windows ou d'Unix. Chez Macintosh, on a considéré qu'un simple CR suffisait. D'où les questions plus ou moins ésotériques que vous avez probablement rencontré en convertissant des textes en provenance d'une autre machine : faut-il transformer les CR en CR+LF ?

Un dernier exemple est le 7 BELL qui déclenche un coup de sonnette (équivalent du beep) au lieu d'afficher quoi que ce soit.

Beaucoup d'applications de traitement de texte utilisent les caractères de contrôles comme balises[74] en donnant un sens nouveau aux caractères en principe

---

[74] Par exemple pour signaler un changement de police ou de corps.

destinés à la communication (ACK et NACK par exemple, qui permettaient au départ de confirmer qu'un message avait été correctement reçu ou pas).

Signalons quelques particularités de ce code. Il est assez rationnel car les lettres sont classées en ordre alphabétique et que l'on peut donc passer des majuscules aux minuscules en ajoutant ou retranchant 32[75] . L'ordre alphabétique peut donc être implémenté de manière assez simple comme une comparaison entre codes ASCII considérés comme des nombres. Ceci a pour conséquence que tous les caractères sont ordonnés dans l'ordre de la table ASCII, pas seulement les lettres. Si un nom de fichier dans le Finder commence par un espace (spc dans la table) ou un $, il apparaîtra avant tous les autres fichiers ayant un nom composé uniquement de lettres, l'espace et le $ ayant un code plus petit que n'importe quelle lettre.

Tous les tris alphabétiques sont basés sur cette propriété fondamentale. Ils peuvent être améliorés par la confusion des majuscules et minuscules, la réduction des caractères accentués à la forme de base, mais fondamentalement, le principe reste le même.

Code ASCII étendu.

Comme le contrôle de parité n'est pas nécessaire dans les micro ordinateurs, le réemploi du bit de poids fort a permis de définir 128 nouveaux codes.

Malheureusement, tout le monde en a fait à sa tête en affectant ces codes à des caractères différents (divers signes typographiques, minuscules accentuées appartenant à divers alphabets, caractères graphiques etc…), si bien que cette partie du code ASCII (on parle souvent du code ASCII étendu), n'est pas la même d'un ordinateur à l'autre ni même d'une police à l'autre.

Pour réduire le désastre, on a tout de même standardisé un certain nombre de jeux de caractères dont vous pouvez découvrir la liste dans les préférences de votre navigateur ou de votre utilitaire de courrier. Le français s'écrit par exemple en ISO Latin.

Par conséquent, pour transmettre un texte d'un ordinateur à l'autre (ou maintenant d'un système d'exploitation à un autre), on ne rencontre aucun problème tant que l'on utilise le code ASCII standard. Dès que l'on utilise des caractères étendus, on doit en général passer par un standard commun si la conversion directe n'existe pas. Les navigateurs et utilitaires de courrier le font en principe automatiquement, mais on peut rencontrer des cas où la conversion est incorrecte malgré tout.

Unicode.

Nous ne décrirons pas Unicode en détail. C'est une proposition visant à sortir

[75] On peut même positionner directement le bit 5 à 0 pour les majuscules et à 1 pour les minuscules si l'on utilise un langage permettant de le faire (ce n'est pas le cas d'AppleScript). Les changements de casse sont donc très simples.

des limitations que comporte ASCII en codant les caractères sur deux octets au lieu d'un. On dispose ainsi de 65 000 combinaisons et quelque, ce qui permet non seulement d'espérer avoir un standard pour pour tous les caractères accentués de tous les alphabets, mais aussi de traiter des idéogrammes comme les caractères chinois ou les alphabets phonétiques japonais (kanji).

Signalons que le code ASCII est un sous ensemble d'Unicode. Les caractères ASCII standard ont en effet pour Unicode leur code ASCII, le deuxième octet étant mis à zéro. Une chaîne Unicode peut donc être facilement convertie en ASCII tant qu'elle ne contient pas de caractère exotique, en éliminant simplement l'octet supplémentaire. Une chaîne ASCII peut être convertie en Unicode en rajoutant un octet à zéro à chaque code ASCII.

Chaînes et textes.

Chaînes et textes sont des suites de caractères typographiques. Les caractères sont codés en utilisant l'une des conventions exposées ci-dessus.

Il reste naturellement à trouver un moyen de signaler la fin de la chaîne ou du texte. Deux solutions sont possibles : faire commencer la suite par un nombre indiquant le nombre de caractères de la suite. La deuxième consiste à placer tout à la fin un caractère servant de balise ou de limite (On utilise naturellement un caractère de contrôle qui ne risque pas trop d'avoir d'autre emploi : NUL ou EOT = end of text en ASCII). Les deux solutions présentent des avantages et des inconvénients et sont donc utilisées.

Les chaînes ainsi encodées, ne comportant que des caractères ASCII et des caractères de contrôle également définis par le code ASCII sont souvent appelés textes ASCII ou textes. Sous MacOS, le type TEXT (quelque soit l'application créatrice) est réservé en principe à des fichiers contenant des chaînes ou textes de ce type.

Rien dans le code ASCII ne permet de spécifier des attributs du texte comme la police de caractère ou le corps. Pour dépasser cette limitation, on a donc défini des formats comme RTF (rich text file). En utilisant des balises, on peut donc enrichir le texte d'un certain nombre d'attributs de mise en page. Le texte lui-même est codé en ASCII de façon ordinaire.

Par exemple :

```
     {\rtf0\mac\deff21{\fonttbl{\f1024 \fnil Verdana;}{\f2944 \fnil Viking;}{\f14213
\fnil Webdings;}{\f18 \fscript Zapf Chancery;}{\f13 \ftech Zapf Dingbats;}{\f2052 \fnil
Zeal;}}
     {\colortbl\red255\green255\blue255;\red0\green0\blue0;\red255\green0\blue0;\red0\gre
en255\blue0;\red0\green0\blue255;\red0\green255\blue255;\red255\green0\blue255;\red255\gr
een255\blue0;}
     {\stylesheet{\sbasedon222\snext Normal;}}
     \paperw11900\paperh16840\margt1140\margb1140\margl1140\margr1140\widowctrl\ftnbj\ftn
restart\ftnstart1\pgnstart1\deftab720\sectd\linemod0\linex0\cols1\colsx0
     \pard\plain\pard\ql{\plain\f34\fs24\cf1 Ceci est un texte RTF.}
     }
```

est le texte RTF contenant la phrase "Ceci est un texte RTF" qui apparaît à la fin.

On pourra constater en ouvrant des fichiers produits par des traitements de texte, que la même approche est utilisée, mais cette fois, le balisage ne respecte aucun standard : chaque application a le sien. Par exemple, si on ouvre un fichier AppleWorks contenant simplement la phrase[76] :

Le texte proprement dit finit là :

le fichier contient :


```
      í BOBO  í           0î      J S 9 9 9 9 9 9                             €Ï
ˇˇ            X±
              01
ˇˇ              0¨          ˇ                         -         Z
 ˇˇˇ                SG              {    f  ˇˇ   B    LOM!                 `
&G                                                                    ˇ
x     H H
      ˇ·ˇ‰ + 7 G { ‡     H H     ÿ (     d          ˇ  '              `   ê
ˇˇ                                <∞                                       Ü
[   R   /     ¿ W       Î                             <D          <Ñ
<\         Î W                       /       ©d                        0   4¥
ˇˇˇˇˇˇˇˇ e
      _ p  DSET   T     !                !          0Ä  0t  0,        .      · ÿ  <ú
         "  0ê   "Le texte proprement dit finit là:    Z  ˇˇ   N                  ·
DSET   Ç  ˇˇ   . H                    ˇˇˇ ˇˇˇ  <|               ·   ÿ
B              ÿ   ·                       ˇˇ       6  ˇˇ   *
B        DSET   Ç  ˇˇ   . H                *ˇ ˇˇˇ ˇˇˇ  <à
B              ÿ   ·                       ˇˇ       6  ˇˇ   *
B            DSUM                    SG       HDNI      STYL   ‰      h
     ˇˇ     D  STYL  0à
     Ád      0<  -x  0Ï  0¿  .H  .t                       ˇˇ      ˇˇ
       ˇˇˇˇ         ˇˇ            ˇˇ          ˇˇˇˇ            ˇˇ            ˇˇ
       ˇˇ           ˇˇ
       ˇˇ           ˇˇ
       ˇˇ
             ˇˇ
       ˇˇ           ˇˇ            ˇˇˇˇ                        ˇˇ          ˇˇˇˇ
ˇˇ
           ˇˇ   F  ˇˇ         HASH¿˚D‟  ı·/1       (
     (
     P¢G   ˚Ωè  g∫ f  u∂or  {‘,í       .
       CHAR  .  ˇ  ˇˇˇˇ  ˇˇ            ˇˇˇˇ       ˇˇ      ˇˇ          ˇ.          ˇˇˇˇ
ˇÎ         ˇˇˇˇ    "  ˇfl         ˇˇˇˇ       ˇ˚
      ˇˇ       Ä           ˇˇ      ˇˇ                      ˇˇ      ˇˇ          ˇÈ      ˇˇ
"  ˇ>        ˇˇ          ˇ˚
      ˇˇˇˇˇˇ       ˇ˛       ˇˇ          ˇˇ          :  ˇˇ        HASH  Ñ


        R  ˇˇ
       CELL  0$  ˇˇ      ˇ            ˇˇ       ˇ         ˇˇ                    "  ˇˇ
HASH            ˇ            j  ˇˇ
       GRPH  1  nˇˇ      ˇ                  ˇˇ       ˇ                ˇˇ
"  ˇˇ        HASH                     ∆  ˇˇ   1
```


[76] L'opération exacte consiste à changer le type du fichier en TEXT (avec Resedit) puis de le rouvrir avec un éditeur comme SimpleText ou même Appleworks. Les caractères de contrôle non affichables apparaissent sous forme d'espaces.

RULR ., ˇˇˇ ˇ
ˇˇ ˇ
ˇˇ ˝ @
ˇˇ
(ˇˇ HASH ˇ ˇ @ ˝ (LKUP
ˇˇ ˇˇ $ NAME Défaut Défaut TABL
En-tête Corps Pied de page
Note de renvoi N° de note de renvoi d
D FNTM NicolasCocTBla Geneva
ˇˇ Geneva Û Interlude
> Hoefler Text MCRO MCRO oBlN
oBlN BBAR BBAR ˇˇMARK MRKS ˇˇ MOBJWMBT ETBL XDSUM
ÀHDNI ÓSTYL ¯MCRO ‰oBlN ÙBBAR MARK WMBT :ETBL Fˇ˛˝¸˚˙˘¯ ÒÚÛÙı^˜

on y retrouve effectivement la phrase originale parfaitement lisible. Le reste est plus ésotérique. On peut y reconnaître pourtant des noms de polices à la fin.

Graphiques.

Mémoire et adaptateur vidéo.

Commençons par exposer rapidement comment est fabriquée l'image écran sur un micro-ordinateur.

Le moniteur est constitué d'un canon à électrons enfermé dans une enceinte sous vide en verre dont la partie visible à l'utilisateur est recouverte d'une matière fluorescente. Lorsque des électrons viennent frapper cette couche, celle-ci émet une lumière visible que nous pouvons apercevoir. Le canon à électrons produit un faisceau très fin qui, s'il frappait toujours la couche sensible au même endroit, produirait un point unique de taille très réduite. Par le moyen de l'électronique, on peut produire un faisceau plus ou moins intense et obtenir de la sorte un point plus ou moins brillant.

Pour produire une image et non un point unique, on oblige le faisceau à parcourir toute la surface de l'écran en suivant des lignes horizontales très rapprochées, exactement de la même manière que le curseur parcourt un texte lorsque l'on garde la touche -> enfoncée. Arrivé en bas à droite, le faisceau revient en haut à gauche et tout reprend.

Ce balayage est répété régulièrement à une cadence définie par une horloge. S'il est suffisamment rapide[77] , l'œil humain ne distingue pas un point qui se déplace, mais la figure produite par le déplacement du point (persistance rétinienne). Cette cadence est ce que l'on appelle la fréquence de balayage, elle est mesurée en hertz (= nombre de fois par seconde), et vous pourrez constater que c'est l'un des réglages offerts dans le tableau de bord moniteurs. Plus elle est élevée, plus l'image paraît stable.

La résolution est le nombre de points affichés à l'écran, autrement dit le nombre de points dans chaque ligne x le nombre de lignes.

---

[77] En moyenne 50 fois par seconde, plus ou moins selon les personnes. Lorsque l'on est au voisinage de la limite, l'image paraît trembler ou scintiller légèrement.

Si l'on module l'intensité du faisceau d'électrons en même temps qu'il se déplace, on peut obtenir une image en niveaux de gris, à condition que le signal électrique qui décrit la luminosité des points de l'image soit rigoureusement synchronisé avec le balayage pour que l'image soit stable.

En télévision, ce signal est envoyé par l'émetteur par ondes hertziennes.

Dans un micro ordinateur, ce signal de luminosité est produit par un circuit spécial d'après le contenu de la mémoire vidéo. Ce circuit lit répétitivement une zone de mémoire, exactement à la même cadence que le faisceau d'électrons parcourt l'écran, et envoie au moniteur le signal de luminosité d'après les valeurs qu'il lit dans la mémoire. En fait, sur les écrans couleur, il crée trois signaux de luminosité correspondant à trois couleurs fondamentales et trois couches sensibles superposées à l'écran : le rouge, le vert et le bleu.

Dans la mémoire vidéo, chaque point de l'écran à son correspondant. Dans les temps héroïques où les écrans ne "savaient" afficher que des points noirs ou blancs (pas de niveaux de gris), il suffisait d'un bit pour enregistrer l'information correspondant à un point. Aujourd'hui où nous disposons de la couleur, c'est évidemment insuffisant, chaque point est représenté au moins par un octet. Il va de soi que plus la résolution est importante, plus la mémoire vidéo nécessaire est grande, et plus les circuits seront sollicités.

Avant d'étudier plus en détail les représentations utilisées, faisons deux remarques :

1- Pour modifier l'image à l'écran, donc afficher ses résultats, un programme va modifier le contenu de la mémoire vidéo, ce qui aura pour effet de modifier le signal envoyé au moniteur. Dans les temps héroïques, les programmes effectuaient ces modifications directement et configuraient le circuit qui élabore le signal. C'est maintenant l'affaire de procédures implémentées dans le système, de manière à ce que l'utilisateur puisse choisir sa résolution d'écran.

2- Presque tous les adaptateurs graphiques modernes (i.e. le couple mémoire vidéo + circuit de lecture) sont également associés à un processeur graphique spécialisé. Nous verrons plus en détail quel est son but, mais disons tout de suite, qu'il a pour but d'aider le processeur principal à effectuer des modifications rapides dans la mémoire vidéo, ces processeurs graphiques sont en effet capables de tracer des figures simples (lignes, polygones) bien plus vite que le processeur principal.

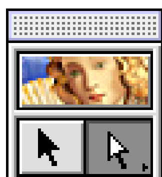
Il s'ensuit que trois circuits se partagent l'accès à la mémoire vidéo, et ceci doit se faire sans conflit d'accès, et sans perte de synchronisation avec le balayage. Bien entendu c'est l'électronique qui s'en charge, et le programmeur n'y a pas accès. Mais ce sont les performances de cet ensemble (vitesse d'affichage et résolutions et fréquences de balayage) qui sont au centre des discussions sur les cartes vidéo qui sont maintenant toutes produites par des constructeurs spécialisés.

Bitmaps.

Définition.

Une bitmap (en français carte ou tableau de bits) est une structure calquée sur le contenu de la mémoire vidéo où chaque point est représenté par un certain nombre de bits. La position de ces paquets de bit dans la mémoire vidéo spécifie donc implicitement la position des points correspondants à l'écran.
Par exemple, si on travaille avec une résolution de 640 x 480 et que chaque point est codé sur un octet, l'adresse du point de coordonnées 100, 100 à l'écran se calculera comme suit (origine du repère en haut à gauche) :
puisque c'est le centième point de la centième ligne et que chaque ligne contient 480 points (donc 480 octets), l'adresse vaut : (100 x 480) + 100 = 48 100
Si je modifie la valeur stockée dans cet octet; la couleur du point 100 x 100 sera modifiée à l'écran.

Une bitmap est un tableau de points colorés (appelés pixels). Sans rentrer dans les détails des dimensions du point et du problème des limites, disons qu'un dessin est représenté comme une série de taches de couleur de même dimension. A condition que les taches soient suffisamment petites, le pouvoir séparateur de l'œil est insuffisant pour les distinguer si bien que l'ensemble apparaît comme un graphisme continu.



L'agrandissement rend les pixels nettement visibles sous forme de petits carrés.

Cette forme de graphisme est très utilisée : les écrans (comme nous l'avons vu) et de nombreuses imprimantes ou dispositifs d'impression travaillent à partir de bitmaps.

Propriétés.

Une image en bitmap se caractérise par son nombre de points en largeur et en hauteur. Il s'ensuit qu'elle n'a pas de dimension intrinsèque et qu'elle apparaît plus ou moins grande selon le dispositif d'affichage que l'on utilise, écran ou imprimante. C'est ainsi qu'une image de ce type peut apparaître au format A4 à l 'écran et comme un timbre poste sur une bonne imprimante.
Les dispositifs d'affichage et les imprimantes se caractérisent eux par leur résolution (exprimée en pixels (ou points) par unité de longueur). En général, d'ailleurs, ils en ont plusieurs comme on peut s'en convaincre en ouvrant les réglages de l'écran. Ceux-ci sont exprimés en points car la taille de l'écran est fixe, mais cela revient au même. Ces résolutions représentent le nombre de points que l'écran est capable d'afficher en largeur et hauteur. Il est évident en jouant sur ces réglages que plus la résolution est élevée, plus la taille apparente d'une image est

faible.
La représentation interne dans l'ordinateur (dans un fichier aussi bien que dans la mémoire vidéo) est la suivante : chaque point est représenté par un code de couleur. La largeur (en points) et la hauteur de l'image sont stockées à part, ce qui permet à l'ordinateur de ranger ces points par rangées superposées.

Le code de couleur est composé de diverses manières : soit il contient plusieurs nombres qui représentent les proportions de couleurs fondamentales dans la couleur résultante (voir modèles de couleur ci-dessous) : on parle alors de couleurs réelles. Soit un index qui désigne une couleur dans une palette. Une palette est une collection de couleurs numérotées. Dans l'image, on décrit la couleur d'un point par le numéro de sa couleur dans la palette. On parle alors de couleurs indexées. Le deuxième mode est plus économique en mémoire vidéo au prix d'une limitation dans le nombre de couleurs différentes affichables simultanément[78] .

La profondeur de couleur est le nombre de couleurs distinctes que l'on peut stocker dans un code de couleurs. Si on le dimensionne à un octet, il existe seulement 256 valeurs possibles. Dans ce cas un fichier image (ou sa représentation en mémoire) occupera exactement largeur x hauteur octets. Ce mode (utilisé seulement en couleurs indexées ou en niveaux de gris) permet d'avoir seulement 256 couleurs différentes à l'écran. L'image est accompagnée d'une palette qui comporte 256 couleurs, mais chaque couleur peut y être définie avec une plus grande précision. C'est la signification des mentions du genre : 256 couleurs parmi des milliers. Elle indique que 256 couleurs différentes sont affichables à l'écran et que celles-ci sont choisies parmi 65 000 couleurs possibles (description de la couleur sur deux octets dans la palette).

Lorsque les images sont codées en couleurs réelles, trois octets au minimum sont utilisés (on parle alors d'images 8 bits). Le nombre total de couleurs théoriquement possible est alors de 256 x 256 x 256 = 16 777 216. Il s'agit d'une valeur théorique car les systèmes d'affichage ou d'impression et l'œil humain lui-même ne sont pas obligatoirement capables de les distinguer. Il est évident que ce type de représentation est nettement plus gourmand à la fois en mémoire vidéo et en espace disque.

Résolutions d'images, zooms.

Certains logiciels et certains formats ajoutent à la description de l'image une résolution. En toute rigueur, ce renseignement n'a pas de sens. Il s'agit seulement d'une résolution conseillée ou préférée qui permettra aux logiciels et dispositifs d'impression qui en sont capables, de reproduire l'image à une dimension précise (c'est le cas de Photoshop par exemple). Si le dispositif d'impression peut adopter la résolution demandée, il n'y a pas besoin de transformer l'image. Sinon, une transformation de l'image est indispensable pour garantir une dimension fixée en sortie.

Par exemple, une image de trois mille pixels de large et d'une résolution de 150 points par pouces aura une dimension réelle de 20 pouces sur un dispositif capable d'adopter une résolution de 150 points par pouce. Si le dispositif ne dispose que d'une résolution de 300 points par pouces, il faudra que l'image fasse six mille points de large pour atteindre la même dimension de 20 pouces. Ces trois mille points supplémentaires doivent être créés d'une manière ou d'une autre (la plus simple est de dupliquer simplement tous les points), et ce n'est donc plus l'image

[78] Le format JPEG est un format en couleurs réelles, alors que le format GIF est en couleurs indexées.

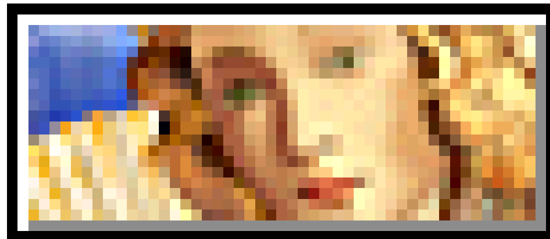
originale qui est affichée, mais une autre, construite plus ou moins automatiquement.

Il existe donc deux sortes de mises à l'échelle : la première est de jouer sur la résolution de la sortie (quand c'est possible, ce qui modifie la taille réelle du pixel élémentaire). Dans ce cas l'image originale n'est pas transformée. Elle apparaît simplement plus ou moins grande. L'autre manière de faire est de transformer réellement l'image, non pas en modifiant la taille du point mais en concentrant plusieurs points en un seul (réduction) ou en remplaçant un pixel par un carré de couleur homogène (agrandissement). C'est ce dernier cas qui conduit aux effets d'escalier très vite apparents lorsque l'on agrandit des images de ce type.

<u>Limitations des images bitmap.</u>

La limitation des images bitmap tient d'abord à leur manque de structure. Elles sont constituées d'une collection de points de couleur individuels sans liens avec leurs voisins. La seule information dont dispose l'ordinateur pour les associer en ensembles plus vastes est leur couleur. Comme les scènes réelles sur une photo par exemple, sont constituées d'objets dont la couleur n'est pas uniforme et dont les frontières sont souvent mal identifiées, il est souvent impossible de les individualiser par un processus automatique[79] .

Par exemple sur cette image volontairement grossière, nous identifions sans difficulté une tête de femme :



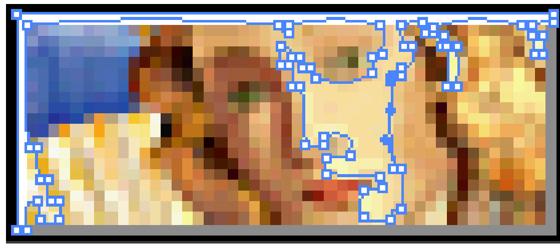
Si l'on demande à l'ordinateur d'isoler le visage des cheveux, le découpage en plages de couleur ne donnera pas un résultat satisfaisant :

- ou bien on demandera un découpage strict basé sur l'identité presque complète des couleurs, et l'on n'obtiendra pas la totalité du visage en une seule fois :

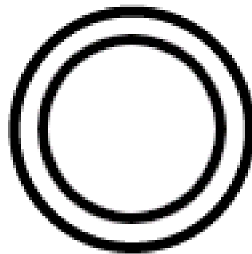
- ou bien on demandera un découpage basé sur une proximité plus lâche et, la région sélectionnée débordera du visage :

---

[79] En fait la reconnaissance de formes est parfois possible, mais c'est encore un domaine de recherche peu développé, sauf en reconnaissance de caractères typographiques ou de dessins au trait.



Ce problème de détourage est bien connu des utilisateurs de Photoshop.
Le deuxième inconvénient réside dans les mises à l'échelle. Même un dessin très simple comme une ligne oblique se transforme en escalier car l'ordinateur ne peut que remplacer les points originaux par des carrés de plus en plus gros qui ne tardent pas à devenir perceptibles à l'œil.
Ces deux cercles concentriques paraissent avoir la même finesse de tracé :



L'agrandissement ci dessous montre la différence. Le cercle intérieur est une image bitmap, le cercle extérieur un objet vectoriel comme nous allons les décrire ensuite.

Représentations vectorielles.

Principe.

Le but de la représentation dite vectorielle est de décrire des images qui ne présentent pas les limitations des images bitmap exposées ci-dessus. Au lieu d'une

liste de points colorés, on utilise une liste d'affichage comprenant des objets définis de manière plus ou moins géométrique. Quelque chose comme : “un cercle, de 5 cm de rayon, centré à 10 x 10 cm des bords de la page, circonférence de couleur 210, 25, 32 et d'épaisseur 0,3 mm, intérieur vide”.

Des programmes comme Appleworks (mode dessin vectoriel) ou Illustrator sont capables d'utiliser cette description pour créer une image bitmap afin que le moniteur l'affiche. C'est aussi le rôle des processeurs graphiques présents sur les cartes vidéo. Mais si on leur demande un agrandissement de l'objet, ils calculeront (selon les méthodes géométriques ordinaires) de nouvelles propriétés pour l'objet, puis en déduiront une nouvelle image bitmap qui sera beaucoup plus précise que si on avait agrandi l'image écran.

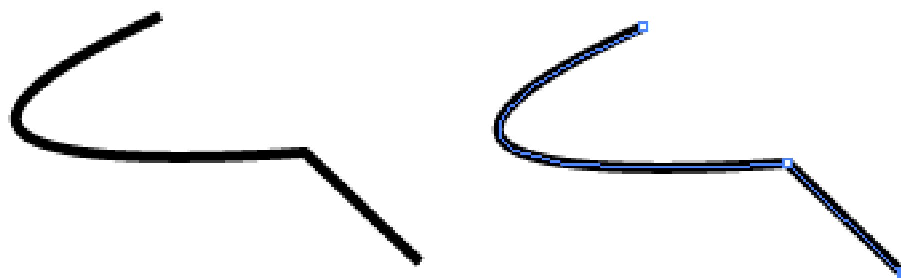
De la même manière, ils seront en mesure de calculer une image bitmap de résolution différente pour un périphérique d'impression. L'image produite aura donc les mêmes dimensions à l'écran et sur le papier, même si l'écran et l'imprimante n'ont pas du tout la même résolution.

Courbes de Bézier.

Illustrator, comme la plupart des programmes de ce genre, n'utilise pas la collection des objets géométriques usuels comme le cercle ou le triangle. Il utilise des segments de droite, et des courbes géométriques connues sous le nom de courbes de Bézier (plus précisément, il s'agit de splines ou de ß-splines, mais la distinction dépasse le cadre de cet exposé).

Ces courbes ont été mises au point par Mr. Bézier pour les concepteurs de carrosseries chez Renault. L'idée était de trouver une description géométrique simple et rapide à traiter par un ordinateur pour résoudre le problème de créer une courbe lissée passant obligatoirement par un certain nombre de points.

Il se trouve que la paire segment de droite/courbe de Bézier permet de définir par morceaux n'importe quelle figure, y compris des choses simples comme les cercles et les triangles. Cela donne également un moyen de les transformer facilement à l'écran.



Le dessin ci dessus est composé d'un segment de droite et d'une courbe. Lorsque ce tracé est sélectionné, les points de contrôle ou d'ancrage apparaissent sous forme de carrés (qui n'apparaissent jamais dans le dessin final). La courbe, quelque soit sa forme, passera toujours par ces points. C'est donc leur position qui permet de définir le tracé et ce sont eux qui permettent de le manipuler.

Ces tracés peuvent être ouverts comme celui de l'exemple ci-dessus dont on identifie clairement les extrémités ou fermés comme un cercle ou un carré. Dans tous les cas, ils possèdent un intérieur. L'intérieur d'un tracé fermé est une notion évidente. L'intérieur d'un tracé ouvert est obtenu en reliant les deux extrémités du tracé pour le refermer fictivement comme dans l'exemple ci-dessous où l'intérieur a

été coloré en bleu :

Un tracé est donc toujours composé d'un contour et d'un intérieur (ou fond).

Conversions, langage Postscript, RIPs, etc.

Pixellisation.

La pixellisation ou rasterisation est la conversion d'une image vectorielle en image bitmap. Cette conversion ne présente aucune difficulté technique, puisque les programmes comme Illustrator l'effectuent régulièrement pour l'affichage écran. Le seul renseignement à fournir est la résolution désirée.

Cette opération est souvent nécessaire pour permettre à d'autres logiciels ou périphériques d'exploiter l'image (quand ils ne "connaissent" pas les images vectorielles et à plus forte raison le format de l'application qui a produit l'image). En particulier, les images destinées aux navigateurs et aux sites Web doivent être pixellisées. Elle peut également se montrer utile pour alléger la tâche de l'ordinateur s'il se montre trop lent à afficher un document composé d'un très grand nombre d'objets. On peut alors en pixelliser un certain nombre pour continuer le travail sur un fond de page pixellisé, puis le rétablir sous forme vectorielle avant d'imprimer.

Imprimantes Postscript, RIPs.

De nombreux périphériques d'impression, tout en ayant une technique de reproduction de type bitmap (en fait toutes les imprimantes à l'exception des traceurs) possèdent les ressources logicielles nécessaires pour traiter des descriptions vectorielles et en faire une image bitmap. On leur envoie donc une description vectorielle qu'elles adaptent à leurs réglages et à leurs capacités en résolution.

Ces descriptions vectorielles sont pratiquement toujours rédigées en langage Postscript qui est devenu un standard de fait. Une imprimante est dite Postscript lorsqu'elle sait interpréter ce langage. Il arrive aussi que cette traduction soit confiée à un logiciel extérieur à l'imprimante et s'exécutant sur un ordinateur du réseau, ce programme s'appelle un RIP Postscript.

Par ailleurs, il est également fréquent de transformer des images vectorielles dans leur description en Postscript comme si on les envoyait à une imprimante, puis de ranger le résultat dans un fichier. Ceci a pour but de transférer les images vers des logiciels qui comprennent aussi le langage Postscript, ou au moins, sont capables de stocker cette description dans un bloc. Cette description ne sera pas toujours visible à l'écran, mais elle apparaîtra correctement dans le document imprimé puisque l'imprimante est capable de l'interpréter. Le format EPS (Encapsulated Postscript Source) est reconnu et traité par de multiples logiciels et sert par conséquent de format d'échange pour les graphiques vectoriels[80] .

Il faut également mentionner le format PDF (Postscript Description File) qui est

[80] Ce format peut aussi intégrer d'autres choses comme des aperçus en basse résolution, mais fondamentalement, il contient toujours une description de page en Postscript.

très proche mais légèrement différent dans sa destination : ces fichiers sont en effet surtout destinés à être affichés et imprimés par le logiciel Acrobat qui ne possède que des capacités de modification très limitées, alors qu'un EPS peut être entièrement réinterprété et permettre de retoucher tous les objets qui composent l'image.

Vectorisation.

La vectorisation consiste dans l'opération inverse de la pixellisation : on tente d'obtenir une description vectorielle d'une image bitmap. Pour les raisons que nous avons déjà exposées, le succès de l'opération est très variable, même lorsqu'il est effectué par des programmes spécialisés et très perfectionnés. On obtient de bons résultats lorsqu'il s'agit de graphismes aux couleurs nettement délimitées (l'idéal est le dessin au trait en noir et blanc) avec un minimum de plans distincts. Les graphismes en demi-teintes aux frontières floues donnent des résultats généralement très décevants.

Le plus souvent, il faut retravailler à la main les tracés produits. On peut obtenir de meilleurs résultats en traitant l'image bitmap au préalable pour augmenter son contraste et la définition des frontières entre les objets que la scène contient. Les modifications de couleur que cela entraîne n'ont pas d'importance, puisque la vectorisation ne reprend pas les couleurs de fond, mais seulement les contours.

Modèles de couleurs.

Couleurs des objets.

Le langage courant considère que la couleur d'un objet est une propriété immuable de cet objet, mais en réalité, c'est un peu plus complexe. La couleur d'un objet est la couleur de la lumière qu'il nous envoie et celle-ci peut largement varier selon les circonstances :

- Certains objets produisent de la lumière : c'est le cas des écrans d'ordinateur ou de télévision, du soleil ou d'un feu. On les reconnaît à ce qu'ils sont visibles dans l'obscurité, en l'absence d'autre source d'éclairage. La puissance lumineuse rayonnée par ces objets ne dépend que de leur énergie propre et peut par conséquent être aveuglante. La couleur de ces objets est celle de la lumière qu'ils émettent, plus ou moins parasitée par la lumière ambiante (les deux s'additionnent).

+ D'autres objets ne produisent aucune lumière (c'est le plus courant). Ils ne sont pas visibles dans l'obscurité et dépendent donc d'une source d'éclairage pour être vus. La lumière que nous recevons d'eux est donc de la lumière ambiante qu'ils renvoient vers nous et ce de deux manières qui ne s'excluent pas :

-- en la réfléchissant à la manière d'un miroir. Les reflets ont alors la même couleur que la lumière ambiante. C'est ce que l'on appelle la réflexion spéculaire.

-- en diffusant cette lumière (réflexion diffuse). C'est ce qui crée notre impression visuelle de couleur : la lumière ambiante n'est pas réémise en totalité, l'objet joue le rôle d'un filtre en absorbant définitivement certaines couleurs. Un objet qui nous paraît bleu absorbe dans la lumière blanche les couleurs rouge, jaune et vert de l'arc en ciel. Il ne renvoie que le bleu. Ceci explique qu'il paraisse noir en lumière rouge : en effet, la lumière rouge ne contenant aucune composante de bleu, l'objet ne renvoie alors aucune lumière.

Dans ce cas, on le voit, le phénomène est soustractif : l'objet élimine de la

lumière qu'il reçoit d'une autre source, et par conséquent, sa couleur va dépendre largement de l'éclairage sous lequel on le voit.

Il faut ajouter que beaucoup d'objets ont une structure en couche qui rend les choses encore plus complexes : comme on le sait, les vernis modifient radicalement l'apparence des objets, qu'ils soient colorés ou non. C'est encore plus vrai des tableaux à l'huile où des dizaines de couches minces et différentes peuvent se superposer.

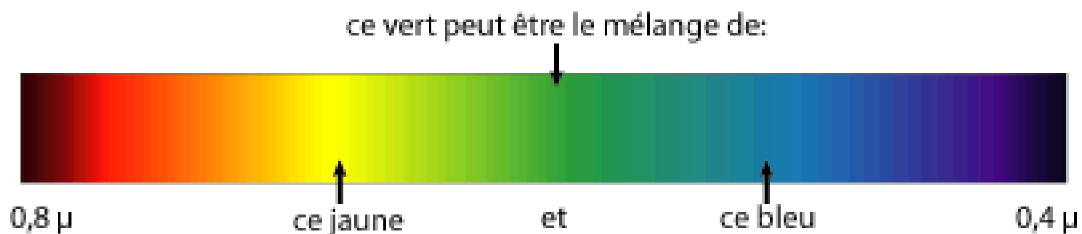
Notre système ordinaire de reproduction des couleurs (tant à l'écran qu'en imprimerie) est donc extrêmement rudimentaire par rapport à la complexité des objets à représenter : nous nous limitons en effet à trois couleurs de base (alors que la lumière blanche en contient une énorme quantité), et d'autre part nous ne pouvons reproduire ni les effets de couches superposées ni la puissance lumineuse réellement émise par les objets (l'exemple du soleil est caricatural : personne ne risque d'être ébloui par l'image du soleil sur un écran d'ordinateur, car celui-ci est bien incapable d'émettre suffisamment de lumière pour nous éblouir).

- Pour mémoire, on peut mentionner une troisième catégorie d'objets qui se montrent capables d'émettre des rayonnements lumineux qui ne sont pas présents dans la lumière ambiante, mais qui ont besoin d'une source d'éclairage pour être visible : en fait ils transforment la lumière qu'ils reçoivent. C'est le cas des objets fluorescents : l'exemple le plus connu étant la “lumière noire” des music-halls : sous cette lumière invisible à l'œil (des ultraviolets), certains objets comme les tissus blancs prennent une couleur éclatante. Ils transforment un rayonnement invisible en rayonnement visible (N.B. c'est loin d'être le seul cas de figure possible, mais une description même succincte de tous les phénomènes physiques susceptibles de produire de la lumière dépasse largement le cadre de cet exposé).

Lumière et couleur.

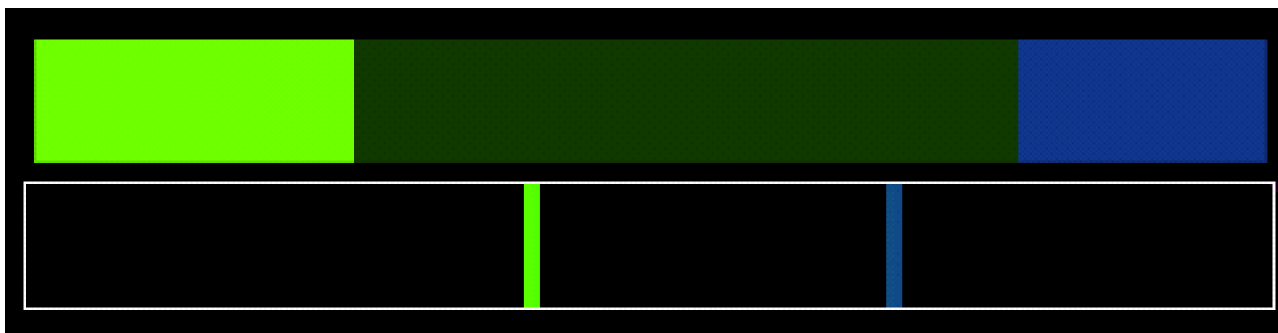
Rappelons ici que la lumière est un rayonnement électromagnétique dont une faible partie seulement nous est visible : entre les longueurs d'onde de 0,4 microns et 0,8 microns. Au delà de ces limites, de la lumière existe (c'est l'infra-rouge et l'ultra violet), mais notre œil n'y est pas sensible. L'impression visuelle de couleur que nous percevons est directement associée à la longueur d'onde de la lumière. L'arc en ciel (ou la décomposition par un prisme) permet d'observer un éventail complet des couleurs pures qui nous sont visibles.

Dans la pratique, nous n'observons jamais de couleurs pures, mais des mélanges assez complexes dont nous retirons une impression de couleur dominante :

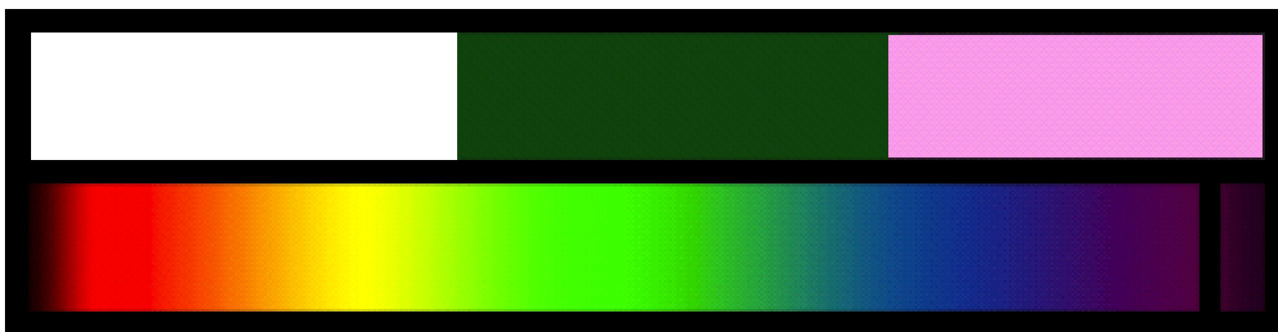


Notre œil n'est pas à même de distinguer si le vert que nous observons est une couleur pure ou un mélange de couleurs pures. Lorsque toutes les couleurs sont présentes en proportions égales dans une lumière, celle-ci nous paraît blanche. La même teinte peut être obtenue de différentes façons comme le montrent les figures suivantes.

Dans le premier cas, le vert sombre central a été obtenu en mélangeant deux couleurs pures. L'analyse de la lumière montre alors les deux composantes isolées. On parle alors de synthèse additive. Ceci peut être obtenu à partir de deux sources de lumière monochromes en proportion égale. Cette méthode de production d'une lumière colorée est celle qu'utilisent les écrans.



Dans le deuxième cas, on élimine une partie de la lumière ambiante (qui est supposée blanche et on voit quels problèmes cela peut poser si elle ne l'est pas). La couleur résultante est alors la couleur complémentaire de la couleur absorbée. Ici, pour obtenir le même vert sombre, on élimine de la lumière blanche cette composante violette. Comme le montre la couleur claire qui ne correspond à aucune couleur de l'arc en ciel, cette couleur n'est pas une couleur pure, mais le violet de l'arc en ciel additionné d'une bonne quantité de blanc (donc de l'ensemble des autres couleurs). Naturellement, on ne voit jamais directement ce violet clair, c'est seulement en analysant la lumière produite que l'on peut constater qu'il manque. On parle alors de synthèse soustractive car on absorbe de la lumière au lieu d'en créer et d'en ajouter.



C'est ainsi que fonctionnent les encres d'imprimerie. Une encre noire est en principe capable d'absorber toute la lumière qui lui parvient. Une encre vert sombre absorbe surtout dans le violet et notablement dans toutes les autres couleurs, etc…

RGB, CMJN, tons directs.

On appelle modèle de couleur la manière de représenter numériquement une couleur. Les deux modèles fondamentaux sont le modèle RGB et le modèle CMJN.

RGB = red, green, blue est une représentation des couleurs qui s'appuie sur trois couleurs fondamentales : rouge, vert et bleu. C'est un modèle additif où les nombres utilisés représentent des quantités de lumière des trois couleurs. Il est donc adapté aux écrans et aux documents destinés à être vus sur des écrans ou en vidéo.

Dans ce modèle, le noir correspond à 0,0,0 c'est à dire, aucune lumière sur les trois composantes. Le blanc est codé par la valeur maximale pour les trois

composantes. Autrement dit, le blanc n'est jamais un blanc “absolu”, mais le blanc le plus blanc que l'écran est capable de produire (ce qui dépend aussi des réglages de contraste et de luminosité que l'utilisateur peut faire sur son moniteur).

Un autre modèle additif est le modèle TSL= teinte, saturation, luminosité. La teinte représente la position dans une roue chromatique assez semblable à l'arc en ciel, la saturation représente une valeur entre cette teinte la plus éclatante et le gris de même luminosité (la teinte est complètement passée), et la luminosité s'explique d'elle-même. Ce modèle est entièrement équivalent au modèle RGB : autrement dit, on peut représenter les mêmes couleurs avec.

CMJN est l'abréviation pour Cyan, Magenta, Jaune, Noir. Contrairement à l'autre, ce modèle est soustractif, donc le noir est cette fois représenté par la valeur la plus élevée des quatre valeurs. Il est adapté à l'imprimerie car il correspond aux charges d'encres à appliquer. Plus on met d'encre et plus la couleur obtenue est sombre.

Ici aussi on rencontre des limites : on ne peut indéfiniment foncer une couleur en mélangeant des encres qui n'absorbent pas toute la lumière (et si elles sont colorées, elles ne peuvent pas le faire par définition), c'est la raison pour laquelle il est nécessaire de prévoir du noir pour obtenir les teintes les plus foncées. Par ailleurs le blanc pur (qui est donc une réserve), ne pourra être plus blanc que le support utilisé. D'où une différence à l'impression selon le papier employé.

Notons que le modèle CMJN n'est pas équivalent au modèle RGB, mais il ne s'agit pas d'une imperfection des logiciels ou des calculs : certaines teintes peuvent être obtenues en synthèse additive (par le mélange additif de lumières colorées) et pas en CMJN (par soustraction de couleurs de base en superposant des encres). C'est une limitation due aux supports de rendu. Elle est contournable en utilisant des encres spéciales pour les couleurs que l'on ne peut obtenir en CMJN. C'est ce que l'on appelle une couleur de ton direct : cette couleur est la couleur de base d'une encre, et n'est pas obtenue par superposition de plusieurs autres. On peut même ainsi utiliser des encres dorées ou argentées qui auront la véritable couleur du métal (car elles en contiennent). Naturellement, l'usage de ces couleurs suppose une plaque d'impression supplémentaire et n'est utile que dans les travaux destinés à l'impression. Le moniteur n'y est pas sensible.

Capacités des imprimantes : gamut.

Même si le modèle CMJN est fondé sur les possibilités des techniques d'impression, il reste un modèle théorique, c'est à dire qu'en pratique, les imprimantes ne seront pas forcément capables de reproduire toutes les couleurs du modèle (l'imprimante peut par exemple utiliser des encres qui sont différentes du cyan, magenta et jaune). L'ensemble des couleurs imprimables par une imprimante donnée est ce qu'on appelle son gamut (abréviation pour gamme utile). Il est nécessaire d'en tenir compte si l'on a des impératifs de reproduction des couleurs très élevés.

Calibration.

La calibration est l'opération qui permet d'ajuster aussi étroitement que possible les couleurs vues à l'écran et sur les documents imprimés. On aura peut-être soupçonné d'après les explications précédentes que la reproduction parfaitement fidèle d'un objet sur un écran comme sur le papier est théoriquement impossible car les méthodes de production de l'image sont différentes et ne peuvent donc produire

exactement les mêmes résultats. Par ailleurs, nous ne percevons pas tous les couleurs de la même manière. On peut donc tout au plus s'approcher d'un idéal inaccessible.

La calibration ne joue pas sur le réglage des appareils, mais sur la conversion des valeurs numériques des couleurs. On procède donc par comparaison entre deux états du même document et on détermine des ajustements des valeurs de couleur à utiliser quand on passe de l'un à l'autre. Par exemple, si l'on veut obtenir une impression visuelle identique pour un document sur papier et son aspect à l'écran une fois scanné, on constituera une table de conversion qui permettra de remplacer les valeurs numériques de couleur en provenance du scanner par des valeurs plus appropriées. Le scanner peut ainsi renvoyer une valeur de 255,255,255 pour les points d'une feuille de papier blanc. Cette valeur sera utilisée pour l'imprimante où elle a une bonne chance de fonctionner, mais elle sera remplacée avant affichage à l'écran par une valeur de gris léger pour tenir compte du fait qu'un écran a toujours des blancs qui paraissent plus lumineux que ceux d'un document imprimé.

Dans la pratique, on construit une table par dispositif d'affichage (scanners, écrans, imprimantes). C'est ce qu'on appelle un profil colorimétrique.

Malheureusement, du fait que les appareils considérés changent de caractéristiques en vieillissant (on dit qu'ils dérivent), ces profils ne sont pas définitifs. Il faut les refaire ou les vérifier d'autant plus régulièrement qu'on a des exigences élevées.

Scripts complets de l'exemple développé.

Etape 2

Objet

```
(*	Script de démo des objets scripts
Etape 2 avec la méthode objet
*)

-- Stockage des données: une liste d'objets scripts.
-- Comme la property subsiste d'une fois sur l'autre je retrouverai ces données
property listeP : {}

-- --------------- le programme principal -------------
on run
  repeat
	choose from list {"1- Saisir une personne", "2- Voir les personnes"} ¬
		cancel button name "Quitter"
	if result = false then -- = bouton Quitter
		exit repeat
	else
		set rep to first word of item 1 of result
		if rep = "1" then
			newPersonne()
		else if rep = "2" then
			visuPersonne("Liste des personnes :")
		end if
	end if
  end repeat
end run

-- ------------- Création des fiches personnes ---------------

on newPersonne()
  display dialog ¬
	"Donner le nom de la personne" default answer ""
  set nnom to (text returned of result)
  display dialog ¬
	"Donner l'âge de la personne" default answer ""
  set aage to (text returned of result) as integer

  choose from list {"Femme", "Homme"}
  set rep to first item of result
  if rep = "Femme" then
	set the end of listeP to creaFemme(nnom, aage)
  else if rep = "Homme" then
	set the end of listeP to creaHomme(nnom, aage)
  end if
end newPersonne

on creaPersonne(sonNom, sonAge) -- la classe fondamentale
  script
```

```
		property nom : sonNom
		property age : sonAge

		on getInfos()
			return nom & " a " & age & " ans"
		end getInfos
	end script
end creaPersonne

on creaHomme(sonNom, sonAge) -- les hommes
	set zzz to creaPersonne(sonNom, sonAge) -- créer l'objet parent
	script
		property parent : zzz

		on getInfos()
			if my age > 21 then
				set tmp to "un homme"
			else
				set tmp to "un garçon"
			end if
			return my nom & ", " & tmp & ¬
				" âgé de " & my age & " ans"
		end getInfos

	end script
end creaHomme

on creaFemme(sonNom, sonAge) -- les hommes
	set zzz to creaPersonne(sonNom, sonAge) -- créer l'objet parent
	script
		property parent : zzz

		on getInfos() -- retourne une liste contenant l'âge et la nature de la personne
			if my age > 18 then
				set tmp to "une femme"
			else
				set tmp to "une jeune fille"
			end if

			if my age > 30 then  -- le calcul de courtoisie
				set x to (my age) div 5
			else
				set x to 0
			end if

			return my nom & ", " & tmp & ¬
				" âgée de " & ((my age) - x) & " ans"
		end getInfos

	end script
end creaFemme

-------------------- Visualisation --------------

on visuPersonne(aPrompt)
	-- 1- retour en cas de liste vide
	if listeP = {} then
		display dialog "Il n'y a aucune personne saisie !"
```

```
        return {}
    end if

    -- 2- constitution de la liste des informations des personnes
    set listeInfos to {}
    repeat with i from 1 to count listeP
        getInfos() of item i of listeP
        copy (i as string) & "- " & result to the end of listeInfos
    end repeat

    return choose from list listeInfos with prompt (aPrompt) ¬
        with multiple selections allowed and empty selection allowed

end visuPersonne
```

Traditionnel

```
(*      Script de démo des objets scripts
Etape 2 avec la méthode traditionnelle
*)

-- Stockage des données: une liste de records.
-- Comme la property subsiste d'une fois sur l'autre je retrouverai ces données
property listeP : {}

---------------- le programme principal -------------
on run
    repeat
        choose from list {"1- Saisir une personne", "2- Voir les personnes"} ¬
            cancel button name "Quitter"
        if result = false then -- = bouton Quitter
            exit repeat
        else
            set rep to first word of item 1 of result
            if rep = "1" then
                newPersonne()
            else if rep = "2" then
                visuPersonne("Liste des personnes :")
            end if
        end if
    end repeat
end run

-------------- Création des fiches personnes ---------------

on newPersonne()
    display dialog ¬
        "Donner le nom de la personne" default answer ""
    set nnom to (text returned of result)
    display dialog ¬
        "Donner l'âge de la personne" default answer ""
    set aage to (text returned of result) as integer

    choose from list {"Femme", "Homme"}
    set rep to first item of result
    if rep = "Femme" then
        set asexe to "F"
    else if rep = "Homme" then
```

```
		set asexe to "H"
	end if
	
	copy creaPersonne(nnom, aage, asexe) to the end of listeP
	
end newPersonne

on creaPersonne(sonNom, sonAge, sonSexe) -- création d'un record
	return {nom:sonNom, age:sonAge, sexe:sonSexe}
end creaPersonne

--------------- Les informations ---------------

on getInfos(pers)
	return ¬
		nom of pers & ", " & getGenre(sexe of pers, age of pers) & ¬
		" âgé de " & makiAge(sexe of pers, age of pers) & " ans"
end getInfos

on getGenre(sonSexe, sonAge) --le genre de la personne
	if sonSexe is "H" then
		if sonAge > 21 then
			return "un homme"
		else
			return "un garçon"
		end if
	else if sonSexe is "F" then
		if sonAge > 18 then
			return "une femme"
		else
			return "une jeune fille"
		end if
	end if
end getGenre

on makiAge(sonSexe, sonAge) -- le calcul de courtoisie
	if sonSexe is "F" and sonAge > 30 then
		return sonAge - sonAge div 5
	else
		return sonAge
	end if
end makiAge

------------------ Visualisation --------------

on visuPersonne(aPrompt)
	-- 1- retour en cas de liste vide
	if listeP = {} then
		display dialog "Il n'y a aucune personne saisie !"
		return {}
	end if
	
	-- 2- constitution de la liste des informations des personnes
	set listeInfos to {}
	repeat with i from 1 to count listeP
		getInfos(item i of listeP)
		copy (i as string) & "- " & result to the end of listeInfos
	end repeat
```

```
  return choose from list listeInfos with prompt (aPrompt) ¬
          with multiple selections allowed and empty selection allowed

end visuPersonne
```

Etape 3

Objet

```
(*        Script de démo des objets scripts
Etape 3 avec la méthode objet
*)

-- Stockage des données: une liste d'objets scripts.
-- Comme la property subsiste d'une fois sur l'autre je retrouverai ces données
property listeP : {}

------------------ le programme principal -------------
on run
  repeat
          choose from list {"1- Saisir une personne", "2- Voir les personnes", "3-Saisir les
mariages."} ¬
                  cancel button name "Quitter"
          if result = false then -- = bouton Quitter
                  exit repeat
          else
                  set rep to first word of item 1 of result
                  if rep = "1" then
                          newPersonne()
                  else if rep = "2" then
                          visuPersonne("Liste des personnes :")
                  else if rep = "3" then
                          newMariage()
                  end if
          end if
  end repeat
end run

---------------- Création des fiches personnes ---------------

on newPersonne()
  display dialog ¬
          "Donner le nom de la personne" default answer ""
  set nnom to (text returned of result)
  display dialog ¬
          "Donner l'âge de la personne" default answer ""
  set aage to (text returned of result) as integer

  choose from list {"Femme", "Homme"}
  set rep to first item of result
  if rep = "Femme" then
          set the end of listeP to creaFemme(nnom, aage)
  else if rep = "Homme" then
          set the end of listeP to creaHomme(nnom, aage)
```

```
  end if
end newPersonne

on newMariage()
  set listeChoix to visuPersonne("Choisissez une ou plusieurs personnes à marier :")
  if listeChoix = false then
          beep
  else if listeChoix = {} then

  else
          repeat with choix in listeChoix
                  set x to (first word of choix) as integer -- récupère la position de la fiche
                  tell (item x of listeP) to Convoler(x)
          end repeat
  end if
end newMariage

on creaPersonne(sonNom, sonAge) -- la classe fondamentale
  script
          property nom : sonNom
          property age : sonAge
          property conj : 0

          on getInfos()
                  return nom & " a " & age & " ans"
          end getInfos

          on getConjoint() -- retourne l'info mariage
                  if conj = 0 then  -- pas marié
                          return " est célibataire."
                  else
                          return " a épousé " & (nom of item (my conj) of listeP) & "."
                  end if
          end getConjoint

          on Mariable()
                  return (my conj = 0) -- évite la bigamie !!!
          end Mariable

          on Convoler(liste, moi)
                  if liste ≠ {} then
                          choose from list liste with prompt "Choisissez un conjoint."
                          set x to (first word of item 1 of result) as integer
                          set my conj to x
                          set conj of (item x of listeP) to moi
                  end if
          end Convoler
  end script
end creaPersonne

on creaHomme(sonNom, sonAge) -- les hommes
  set zzz to creaPersonne(sonNom, sonAge) -- créer l'objet parent
  script
          property parent : zzz

          on getInfos()
                  if my age > 21 then
                          set tmp to "un homme"
```

```
			else
				set tmp to "un garçon"
			end if
			return my nom & ", " & tmp & ¬
				" âgé de " & my age & " ans" & my getConjoint()
		end getInfos

		on Mariable(sex)
			if sex ≠ "H" then
				return false
			end if

			if my age > 18 then
				return continue Mariable()
			else
				return false
			end if
		end Mariable

		on Convoler(moi)
			-- vérifier l'âge du postulant
			if my Mariable("H") then
				-- créer une liste de conjoints possibles
				set liste to {}
				repeat with i from 1 to count listeP
					set pers to item i of listeP
					if (Mariable("F") of pers) then
						copy ((i as string) & "-" & nom of pers) to the end of liste
					end if
				end repeat
				continue Convoler(liste, moi)
			else -- la personne est trop jeune ou déjà marié
				display dialog (my nom & "  est trop jeune ou déjà marié.")
			end if
		end Convoler
	end script
end creaHomme

on creaFemme(sonNom, sonAge) -- les hommes
	set zzz to creaPersonne(sonNom, sonAge) -- créer l'objet parent
	script
		property parent : zzz

		on getInfos() -- retourne une liste contenant l'âge et la nature de la personne
			if my age > 18 then
				set tmp to "une femme"
			else
				set tmp to "une jeune fille"
			end if

			if my age > 30 then -- le calcul de courtoisie
				set x to (my age) div 5
			else
				set x to 0
			end if

			return my nom & ", " & tmp & ¬
				" âgée de " & ((my age) - x) & " ans" & my getConjoint()
```

```
		end getInfos

		on Mariable(sex)
			if sex ≠ "F" then
				return false
			end if

			if my age > 15 then
				return continue Mariable()
			else
				return false
			end if
		end Mariable

		on Convoler(moi)
			-- vérifier l'âge du postulant
			if my Mariable("F") then
				-- créer une liste de conjoints possibles
				set liste to {}
				repeat with i from 1 to count listeP
					set pers to item i of listeP
					if (Mariable("M") of pers) then
						copy ((i as string) & "-" & nom of pers) to the end of liste
					end if
				end repeat
				continue Convoler(liste, moi)
			else -- la personne est trop jeune ou déjà marié
				display dialog (my nom & "  est trop jeune ou déjà mariée.")
			end if
		end Convoler

	end script
end creaFemme

------------------- Visualisation --------------

on visuPersonne(aPrompt)
	-- 1- retour en cas de liste vide
	if listeP = {} then
		display dialog "Il n'y a aucune personne saisie !"
		return {}
	end if

	-- 2- constitution de la liste des informations des personnes
	set listeInfos to {}
	repeat with i from 1 to count listeP
		getInfos() of item i of listeP
		copy (i as string) & "- " & result to the end of listeInfos
	end repeat

	return choose from list listeInfos with prompt (aPrompt) ¬
		with multiple selections allowed and empty selection allowed

end visuPersonne
```

Traditionnel

```
(*	Script de démo des objets scripts
```

```
Etape 3 avec la méthode traditionnelle
*)

-- Stockage des données: une liste de records.
-- Comme la property subsiste d'une fois sur l'autre je retrouverai ces données
property listeP : {}

----------------- le programme principal -------------
on run
  repeat
    choose from list {"1- Saisir une personne", "2- Voir les personnes", "3-Saisir les
mariages."} ¬
      cancel button name "Quitter"
    if result = false then -- = bouton Quitter
      exit repeat
    else
      set rep to first word of item 1 of result
      if rep = "1" then
        newPersonne()
      else if rep = "2" then
        visuPersonne("Liste des personnes :")
      else if rep = "3" then
        newMariage()
      end if
    end if
  end repeat
end run

--------------- Création des fiches personnes ---------------

on newPersonne()
  display dialog ¬
    "Donner le nom de la personne" default answer ""
  set nnom to (text returned of result)
  display dialog ¬
    "Donner l'âge de la personne" default answer ""
  set aage to (text returned of result) as integer

  choose from list {"Femme", "Homme"}
  set rep to first item of result
  if rep = "Femme" then
    set asexe to "F"
  else if rep = "Homme" then
    set asexe to "H"
  end if

  copy creaPersonne(nnom, aage, asexe) to the end of listeP
end newPersonne

on newMariage()
  set listeChoix to visuPersonne("Choisissez une ou plusieurs personnes à marier :")
  if listeChoix = false then
    beep
  else if listeChoix = {} then

  else
    repeat with choix in listeChoix
      set x to (first word of choix) as integer -- récupère la position de la fiche
```

```
            Convoler(x)
        end repeat
    end if
end newMariage

on creaPersonne(sonNom, sonAge, sonSexe) -- création d'un record
    return {nom:sonNom, age:sonAge, sexe:sonSexe, conj:0}
end creaPersonne

--------------- Les informations ---------------

on getInfos(pers)
    return ¬
        nom of pers & ", " & getGenre(sexe of pers, age of pers) & " âgé de " & ¬
        makiAge(sexe of pers, age of pers) & " ans" & getConjoint(pers)
end getInfos

on getGenre(sonSexe, sonAge)
    if sonSexe is "H" then
        if sonAge > 21 then
            return "un homme"
        else
            return "un garçon"
        end if
    else if sonSexe is "F" then
        if sonAge > 18 then
            return "une femme"
        else
            return "une jeune fille"
        end if
    end if
end getGenre

on makiAge(sonSexe, sonAge)
    if sonSexe is "F" and sonAge > 30 then
        return sonAge - sonAge div 5
    else
        return sonAge
    end if
end makiAge

on getConjoint(pers) -- retourne l'info mariage
    if conj of pers = 0 then
        return " est célibataire."
    else
        return " a épousé " & (nom of item (conj of pers) of listeP) & "."
    end if
end getConjoint

------------------- Visualisation --------------

on visuPersonne(aPrompt)
    -- 1- retour en cas de liste vide
    if listeP = {} then
        display dialog "Il n'y a aucune personne saisie !"
        return {}
    end if
```

```
  -- 2- constitution de la liste des informations des personnes
  set listeInfos to {}
  repeat with i from 1 to count listeP
          getInfos(item i of listeP)
          copy (i as string) & "- " & result to the end of listeInfos
  end repeat

  return choose from list listeInfos with prompt (aPrompt) ¬
          with multiple selections allowed and empty selection allowed

end visuPersonne

---------------- Tout pour le Mariage ---------------

on Convoler(x)
  set pers1 to item x of listeP
  set liste to {}
  set i to 0
  repeat with i from 1 to count listeP
          set pers to item i of listeP
          if Mariable(pers1, pers) then
                  copy (i as string) & "-" & nom of pers to the end of liste
          end if
  end repeat

  if liste ≠ {} then
          choose from list liste with prompt "Choisissez un conjoint."
          set y to (first word of item 1 of result) as integer
          set pers2 to item y of listeP
          set conj of pers1 to y
          set conj of pers2 to x
  end if
end Convoler

on Mariable(pers1, pers2)
  set {sonAge1, sonSexe1, sonConjoint1} to {age, sexe, conj} of pers1
  set {sonAge2, sonSexe2, sonConjoint2} to {age, sexe, conj} of pers2
  --conditions
  set b to true
  set b to b and (pers1 ≠ pers2) -- les 2 records sont différents
  set b to b and (sonConjoint1 = 0) --pers1  est célibataire
  set b to b and (sonConjoint2 = 0) --pers2  est célibataire
  set b to b and (sonSexe1 ≠ sonSexe2) -- les sexes sont différents
  set b to b and isAgeMari(sonSexe1, sonAge1) --pers1 est en âge de se marier
  set b to b and isAgeMari(sonSexe2, sonAge2) --pers2 est en âge de se marier
  return b
end Mariable

on isAgeMari(sonSexe, sonAge)
  if sonSexe is "F" then
          return sonAge ≥ 18
  else if sonSexe is "H" then
          return sonAge ≥ 15
  end if
end isAgeMari
```

Etape 4

Objet

```
(*	Script de démo des objets scripts
Etape 4 avec la méthode objet
*)

-- Stockage des données: une liste d'objets scripts.
-- Comme la property subsiste d'une fois sur l'autre je retrouverai ces données
property listeP : {}

-- --------------- le programme principal -------------
on run
  repeat
		choose from list {"1- Saisir une personne", "2- Voir les personnes", "3-Saisir les mariages."} ¬
			cancel button name "Quitter"
		if result = false then -- = bouton Quitter
			exit repeat
		else
			set rep to first word of item 1 of result
			if rep = "1" then
				newPersonne()
			else if rep = "2" then
				visuPersonne("Liste des personnes :")
			else if rep = "3" then
				newMariage()
			end if
		end if
  end repeat
end run

-- ------------- Création des fiches personnes ---------------

on newPersonne()
  display dialog ¬
		"Donner le nom de la personne" default answer ""
  set nnom to (text returned of result)
  display dialog ¬
		"Donner l'âge de la personne" default answer ""
  set aage to (text returned of result) as integer

  choose from list {"Femme", "Homme"} & {"Martien", "Martienne", "Martien neutre"}
  set rep to first item of result
  if rep = "Femme" then
		set the end of listeP to creaFemme(nnom, aage)
  else if rep = "Homme" then
		set the end of listeP to creaHomme(nnom, aage)
  else if rep = "Martien" then
		set the end of listeP to creaMartien(nnom, aage)
  else if rep = "Martienne" then
		set the end of listeP to creaMartienne(nnom, aage)
  else if rep = "Martien neutre" then
		set the end of listeP to creaMartienNeutre(nnom, aage)
  end if
end newPersonne
```

```
on newMariage()
  set listeChoix to visuPersonne("Choisissez une ou plusieurs personnes à marier :")
  if listeChoix = false then
      beep
  else if listeChoix = {} then

  else
      repeat with choix in listeChoix
          set x to (first word of choix) as integer -- récupère la position de la fiche
          tell (item x of listeP) to Convoler(x)
      end repeat
  end if
end newMariage

on creaPersonne(sonNom, sonAge) -- la classe fondamentale
  script
      property nom : sonNom
      property age : sonAge
      property conj : 0

      on getInfos()
          return nom & " a " & age & " ans"
      end getInfos

      on getConjoint() -- retourne l'info mariage
          if conj = 0 then -- pas marié
              return " est célibataire."
          else
              return " a épousé " & (nom of item (my conj) of listeP) & "."
          end if
      end getConjoint

      on Mariable()
          return (my conj = 0) -- évite la bigamie !!!
      end Mariable

      on Convoler(liste, moi)
          if liste ≠ {} then
              choose from list liste with prompt "Choisissez un conjoint."
              set x to (first word of item 1 of result) as integer
              set my conj to x
              set conj of (item x of listeP) to moi
          end if
      end Convoler
  end script
end creaPersonne

on creaHomme(sonNom, sonAge) -- les hommes
  set zzz to creaPersonne(sonNom, sonAge) -- créer l'objet parent
  script
      property parent : zzz

      on getInfos()
          if my age > 21 then
              set tmp to "un homme"
          else
              set tmp to "un garçon"
```

```
            end if
            return my nom & ", " & tmp & ¬
                  " âgé de " & my age & " ans" & my getConjoint()
      end getInfos

      on Mariable(sex)
            if sex ≠ "H" then
                  return false
            end if

            if my age > 18 then
                  continue Mariable()
            else
                  return false
            end if
      end Mariable

      on Convoler(moi)
            -- vérifier l'âge du postulant
            if my Mariable("H") then
                  -- créer une liste de conjoints possibles
                  set liste to {}
                  repeat with i from 1 to count listeP
                        set pers to item i of listeP
                        if (Mariable("F") of pers) then
                              copy ((i as string) & "-" & nom of pers) to the end of liste
                        end if
                  end repeat
                  continue Convoler(liste, moi)
            else -- la personne est trop jeune ou déjà marié
                  display dialog (my nom & "  est trop jeune ou déjà marié.")
            end if
      end Convoler
  end script
end creaHomme

on creaFemme(sonNom, sonAge) -- les hommes
  set zzz to creaPersonne(sonNom, sonAge) -- créer l'objet parent
  script
      property parent : zzz

      on getInfos() -- retourne une liste contenant l'âge et la nature de la personne
            if my age > 18 then
                  set tmp to "une femme"
            else
                  set tmp to "une jeune fille"
            end if

            if my age > 30 then -- le calcul de courtoisie
                  set x to (my age) div 5
            else
                  set x to 0
            end if

            return my nom & ", " & tmp & ¬
                  " âgée de " & ((my age) - x) & " ans" & my getConjoint()
      end getInfos
```

```
    on Mariable(sex)
        if sex ≠ "F" then
            return false
        end if

        if my age > 15 then
            return continue Mariable()
        else
            return false
        end if
    end Mariable

    on Convoler(moi)
        -- vérifier l'âge du postulant
        if my Mariable("F") then
            -- créer une liste de conjoints possibles
            set liste to {}
            repeat with i from 1 to count listeP
                set pers to item i of listeP
                if (Mariable("H") of pers) then
                    copy (i as string) & "-" & nom of pers to the end of liste
                end if
            end repeat
            continue Convoler(liste, moi)
        else -- la personne est trop jeune ou déjà mariée
            display dialog (my nom & "  est trop jeune ou déjà mariée.")
        end if
    end Convoler

  end script
end creaFemme

--============== les martiens ====================
on creaMars(sonNom, sonAge) -- les personnes martiennes en général
  set zzz to creaPersonne(sonNom, sonAge) -- créer l'objet parent
  script
    property parent : zzz
    property conj2 : 0 -- nous avons besoin d'un second conjoint: le neutre

    on getConjoint() -- retourne l'info mariage
        if my conj2 = 0 then -- pas de deuxième conjoint
            continue getConjoint() -- on voit s'il en a un comme pour les humains
        else
            if my conj ≠ 0 then
                return " a épousé " & (nom of item (my conj) of listeP) & ¬
                    " et " & (nom of item (my conj2) of listeP)
            else
                return " est célibataire."
            end if
        end if
    end getConjoint

    on Mariable()
        return (my conj = 0) and (my conj2 = 0) -- évite la bigamie !!!
    end Mariable

    on Convoler(liste1, liste2, moi)
        if (liste1 ≠ {}) and (liste2 ≠ {}) then
```

```
				choose from list liste1 with prompt "Choisissez un premier conjoint."
				set x to (first word of item 1 of result) as integer
				set my conj2 to x
				set conj2 of (item x of listeP) to moi

				choose from list liste2 with prompt "Choisissez un deuxième conjoint."
				set y to (first word of item 1 of result) as integer
				set my conj to y
				set conj of (item y of listeP) to moi

				set conj of (item x of listeP) to y
				set conj2 of (item y of listeP) to x
			end if
		end Convoler

	end script
end creaMars

on creaMartien(sonNom, sonAge)
	set zzz to creaPersonne(sonNom, sonAge) -- créer l'objet parent
	script
		property parent : zzz

		on getInfos() -- retourne une liste contenant l'âge et la nature de la personne
			return my nom & ", " & "un martien " & ¬
				" âgé de " & ((my age) * 3) & " ans" & my getConjoint()
		end getInfos

		on Mariable(sex)
			if sex ≠ "mm" then
				return false
			end if
			return continue Mariable()
		end Mariable

		on Convoler(moi)
			-- créer une liste de conjoints possibles
			set liste1 to {}
			repeat with i from 1 to count listeP
				set pers to item i of listeP
				if (Mariable("mn") of pers) then
					copy (i as string) & "-" & nom of pers to the end of liste1
				end if
			end repeat

			set liste2 to {}
			repeat with i from 1 to count listeP
				set pers to item i of listeP
				if (Mariable("mf") of pers) then
					copy (i as string) & "-" & nom of pers to the end of liste2
				end if
			end repeat

			continue Convoler(liste1, liste2, moi)
		end Convoler

	end script
end creaMartien
```

```
on creaMartienne(sonNom, sonAge)
  set zzz to creaPersonne(sonNom, sonAge) -- créer l'objet parent
  script
	property parent : zzz

	on getInfos() -- retourne une liste contenant l'âge et la nature de la personne
		return my nom & ", " & "une martienne " & ¬
			" âgée de " & ((my age) * 3) & " ans" & my getConjoint()
	end getInfos

	on Mariable(sex)
		if sex ≠ "mf" then
			return false
		end if
		return continue Mariable()
	end Mariable

	on Convoler(moi)
		–– créer 2 listes de conjoints possibles
		set liste1 to {}
		repeat with i from 1 to count listeP
			set pers to item i of listeP
			if (Mariable("mn") of pers) then
				copy (i as string) & "-" & nom of pers to the end of liste1
			end if
		end repeat

		set liste2 to {}
		repeat with i from 1 to count listeP
			set pers to item i of listeP
			if (Mariable("mm") of pers) then
				copy (i as string) & "-" & nom of pers to the end of liste2
			end if
		end repeat

		continue Convoler(liste1, liste2, moi)
	end Convoler

  end script
end creaMartienne

on creaMartienNeutre(sonNom, sonAge)
  set zzz to creaPersonne(sonNom, sonAge) -- créer l'objet parent
  script
	property parent : zzz

	on getInfos() -- retourne une liste contenant l'âge et la nature de la personne
		return my nom & ", " & "un martien neutre" & ¬
			" âgé de " & ((my age) * 3) & " ans" & my getConjoint()
	end getInfos

	on Mariable(sex)
		if sex ≠ "mn" then
			return false
		else
			return continue Mariable()
		end if
```

```
		end Mariable

		on Convoler(moi)
			-- créer 2 listes de conjoints possibles
			set liste1 to {}
			repeat with i from 1 to count listeP
				set pers to item i of listeP
				if (Mariable("mf") of pers) then
					copy (i as string) & "-" & nom of pers to the end of liste1
				end if
			end repeat

			set liste2 to {}
			repeat with i from 1 to count listeP
				set pers to item i of listeP
				if (Mariable("mm") of pers) then
					copy (i as string) & "-" & nom of pers to the end of liste2
				end if
			end repeat

			continue Convoler(liste1, liste2, moi)
		end Convoler

	end script
end creaMartienNeutre

------------------- Visualisation --------------

on visuPersonne(aPrompt)
	-- 1- retour en cas de liste vide
	if listeP = {} then
		display dialog "Il n'y a aucune personne saisie !"
		return {}
	end if

	-- 2- constitution de la liste des informations des personnes
	set listeInfos to {}
	repeat with i from 1 to count listeP
		getInfos() of item i of listeP
		copy (i as string) & "- " & result to the end of listeInfos
	end repeat

	return choose from list listeInfos with prompt (aPrompt) ¬
		with multiple selections allowed and empty selection allowed

end visuPersonne
```

Traditionnel

```
(*	Script de démo des objets scripts
Etape 4 avec la méthode traditionnelle
*)

-- Stockage des données: une liste de records.
-- Comme la property subsiste d'une fois sur l'autre je retrouverai ces données
property listeP : {}

------------------ le programme principal -------------
```

```
on run
  repeat
        choose from list {"1- Saisir une personne", "2- Voir les personnes", "3-Saisir les
mariages."} ¬
              cancel button name "Quitter"
        if result = false then -- = bouton Quitter
              exit repeat
        else
              set rep to first word of item 1 of result
              if rep = "1" then
                    newPersonne()
              else if rep = "2" then
                    visuPersonne("Liste des personnes :")
              else if rep = "3" then
                    newMariage()
              end if
        end if
  end repeat
end run

--------------- Création des fiches personnes ---------------

on newPersonne()
  display dialog ¬
        "Donner le nom de la personne" default answer ""
  set nnom to (text returned of result)
  display dialog ¬
        "Donner l'âge de la personne" default answer ""
  set aage to (text returned of result) as integer

  choose from list {"Femme", "Homme", "Martien", "Martienne", "Martien neutre"}
  set rep to first item of result
  if rep is "Femme" then
        set asexe to "F"
  else if rep is "Homme" then
        set asexe to "H"
  else if rep is "Martien" then
        set asexe to "mm"
  else if rep is "Martienne" then
        set asexe to "mf"
  else if rep is "Martien neutre" then
        set asexe to "mn"
  end if

  copy creaPersonne(nnom, aage, asexe) to the end of listeP

end newPersonne

on newMariage()
  set listeChoix to visuPersonne("Choisissez une ou plusieurs personnes à marier :")
  if listeChoix = false then
        beep
  else if listeChoix = {} then

  else
        repeat with choix in listeChoix
              set x to (first word of choix) as integer -- récupère la position de la fiche
              Convoler(x)
```

```
        end repeat
  end if
end newMariage

on creaPersonne(sonNom, sonAge, sonSexe) -- création d'un record
  return {nom:sonNom, age:sonAge, sexe:sonSexe, conj:0, conj2:0}
end creaPersonne

--------------- Les informations ---------------

on getInfos(pers)
  return ¬
        nom of pers & ", " & getGenre(sexe of pers, age of pers) ¬
        & " âgé de " & makiAge(sexe of pers, age of pers) & " ans" & getConjoint(pers)
end getInfos

on getGenre(sonSexe, sonAge)
  if sonSexe is "H" then
        if sonAge > 21 then
              return "un homme"
        else
              return "un garçon"
        end if
  else if sonSexe is "F" then
        if sonAge > 18 then
              return "une femme"
        else
              return "une jeune fille"
        end if
  else if sonSexe is "mm" then
        return "un martien"
  else if sonSexe is "mf" then
        return "une martienne"
  else if sonSexe is "mn" then
        return "un martien neutre"
  end if
end getGenre

on makiAge(sonSexe, sonAge)
  if sonSexe is "F" and sonAge > 30 then
        return sonAge - sonAge div 5
  else
        return sonAge
  end if
end makiAge

on getConjoint(pers) -- retourne l'info mariage
  if conj of pers = 0 then
        return " est célibataire."
  else
        if sexe of pers is in {"H", "F"} then
              return " a épousé " & (nom of item (conj of pers) of listeP) & "."
        else if sexe of pers is in {"mm", "mf", "mn"} then
              return " a épousé " & (nom of item (conj of pers) of listeP) & " et " & ¬
                    (nom of item (conj2 of pers) of listeP) & "."
        end if
  end if
end getConjoint
```

```
------------------ Visualisation --------------

on visuPersonne(aPrompt)
  -- 1- retour en cas de liste vide
  if listeP = {} then
          display dialog "Il n'y a aucune personne saisie !"
          return {}
  end if

  -- 2- constitution de la liste des informations des personnes
  set listeInfos to {}
  repeat with i from 1 to count listeP
          getInfos(item i of listeP)
          copy (i as string) & "- " & result to the end of listeInfos
  end repeat

  return choose from list listeInfos with prompt (aPrompt) ¬
          with multiple selections allowed and empty selection allowed

end visuPersonne

---------------- Tout pour le Mariage ---------------

on Convoler1(x)
  set pers1 to item x of listeP
  set liste to {}
  repeat with i from 1 to count listeP
          set pers to item i of listeP
          if Mariable(pers1, pers) then
                  copy (i as string) & "-" & nom of pers to the end of liste
          end if
  end repeat

  if liste ≠ {} then
          choose from list liste with prompt "Choisissez un conjoint."
          set y to (first word of item 1 of result) as integer
          set pers2 to item y of listeP
          set conj of pers1 to y
          set conj of pers2 to x
  end if
end Convoler1

on Convoler(x)
  set pers1 to item x of listeP
  set liste to {}
  repeat with i from 1 to count listeP
          set pers to item i of listeP
          if Mariable(pers1, pers) then
                  copy (i as string) & "-" & nom of pers to the end of liste
          end if
  end repeat

  if liste = {} then return

  if sexe of pers1 is in {"H", "F"} then
          choose from list liste with prompt "Choisissez un conjoint."
          set y to (first word of item 1 of result) as integer
```

```
			set pers2 to item y of listeP
			set conj of pers1 to y
			set conj of pers2 to x
	else if sexe of pers1 is in {"mm", "mf", "mn"} then
			repeat
				set rep to choose from list liste with prompt ¬
					"Choisissez 2 conjoints de sexes différents." with multiple selections allowed
				if (count rep) = 2 then
					set y to (first word of item 1 of rep) as integer
					set z to (first word of item 2 of rep) as integer
					set pers2 to item y of listeP
					set pers3 to item z of listeP
					if (sexe of pers2) ≠ (sexe of pers3) then
						-- conjoints de pers1
						set conj of pers1 to y
						set conj2 of pers1 to z
						-- conjoints de pers2
						set conj of pers2 to x
						set conj2 of pers2 to z
						-- conjoints de pers3
						set conj of pers3 to x
						set conj2 of pers3 to y
						exit repeat
					end if
					beep
				end if
			end repeat
	end if
end Convoler

on Mariable(pers1, pers2)
	set {sonAge1, sonSexe1, sonConjoint1} to {age, sexe, conj} of pers1
	set {sonAge2, sonSexe2, sonConjoint2} to {age, sexe, conj} of pers2
	--conditions
	set b to true
	set b to b and (pers1 ≠ pers2) -- les 2 records sont différents
	set b to b and (sonConjoint1 = 0) --pers1  est célibataire
	set b to b and (sonConjoint2 = 0) --pers2  est célibataire
	set b to b and (sonSexe1 ≠ sonSexe2) -- les sexes sont différents
	set b to b and isAgeMari(sonSexe1, sonAge1) --pers1 est en âge de se marier
	set b to b and isAgeMari(sonSexe2, sonAge2) --pers2 est en âge de se marier
	if sonSexe1 is in {"H", "F"} then -- pers1 et pers2 sont de la même planète
			set b to b and sonSexe2 is in {"H", "F"}
	else if sonSexe1 is in {"mm", "mf", "mn"} then
			set b to b and sonSexe2 is in {"mm", "mf", "mn"}
	end if
	return b
end Mariable

on isAgeMari(sonSexe, sonAge)
	if sonSexe is "F" then
			return sonAge ≥ 18
	else if sonSexe is "H" then
			return sonAge ≥ 15
	else if sonSexe is in {"mm", "mf", "mn"} then
			return true
	end if
end isAgeMari
```