# Apprenez le Rapid-Q

Un cours intégral de programmation pour débutant

Par Sydney ROUSSEL

www.progordi.com



Version 1.05 du 12.08.08

# Introduction du cours

Bienvenue dans ce cours. Celui-ci a pour vocation de vous aider à apprendre et à manipuler le langage Rapid-Q. Au cas où vous ne le sauriez pas, le Rapid-Q (abrégé bien souvent en RQ) est un langage de type BASIC. N'entendez pas par là qu'il ne sait rien faire ! Bien au contraire, vous pourrez tout faire (ou presque) avec. La seule véritable limitation sera votre créativité. À l'origine, ce langage a été créé par William Yu. Celui-ci avait précédemment travaillé sur un autre langage BASIC qu'il a par la suite abandonné pour créer le Rapid-Q. Aujourd'hui, le développement de ce langage a cessé depuis le début des années 2000. Mais ne craignez rien, Rapid-Q est toujours d'actualité et sa communauté (notamment francophone) est encore très active.

Pourquoi choisir le Rapid-Q ? Je vais être très franc avec vous. Lorsque l'on débute en programmation, bien des notions nous échappent. Aussi, il est préférable de débuter par un langage qui va vous permettre d'appréhender toutes ces notions, sans vous embrouiller avec sa syntaxe. Imaginez que vous souhaitiez devenir écrivain, et que vous vous décidiez à prendre des cours de langues. Autant prendre des cours dans votre langue natale que dans une langue étrangère, non ? Et bien, c'est un peu la même chose pour la programmation. Et dans le domaine de la clarté, le Rapid-Q fait des merveilles (ou presque). En effet, il vous sera difficile de trouver une syntaxe plus simple que celle d'un langage BASIC. Vous allez me dire : « Oui, mais il y a d'autres BASIC, et je ne veux pas perdre du temps à apprendre un langage dépassé et surtout limité ». Détrompez-vous ! Contrairement à bien d'autres langages BASIC, le Rapid-Q permet de pratiquement tout faire. Alors pourquoi allez chercher ailleurs, quand on vous propose un langage simple à appréhender et puissant ?

Et si vous n'êtes pas un débutant. Alors vous verrez par vous-même combien les qualités de ce langage sont appréciables. Pas la peine de vous en dire plus, vous vous en apercevrez rapidement.

Passons-en maintenant au vif du sujet, à savoir ce cours. Le cours sera très progressif. Chaque partie sera illustrée par des exemples amplement commentés. Je vous recommande très fortement de ne pas faire de simple copier-coller de ces exemples : refrappez-les. Il n'y a pas de meilleurs exercices que de se confronter soi-même aux difficultés que peut rencontrer le programmeur. Vous maîtriserez d'autant plus rapidement la syntaxe et les subtilités de ce langage que vous effectuerez avec attention ces exercices. En parlant d'exercices, je vous réserve quelques perles… Mais rien de bien méchant, vous verrez. Et surtout ne perdez jamais de vue qu'il s'agit avant tout d'apprendre en s'amusant.

**Bonne lecture et bon apprentissage !**

# Sommaire

# 1. Installer le compilateur et l'éditeur

Avant toute chose, il faut installer le compilateur, c'est-à-dire, le programme qui va vous permettre de créer les vôtre. Une fois l'archive téléchargée (voir lien en bas de page), il ne vous reste plus qu'à la décompresser dans le dossier que vous souhaitez. Ne choisissez pas n'importe comment l'emplacement de ce dossier, il est préférable que le chemin d'accès ne soit pas trop long, et qu'il ne comporte pas d'espaces. Installez le compilateur de préférence sur votre partition principale (ce n'est qu'une préférence).

Une fois cela fait, vous trouverez un fichier nommé RC.exe, qui n'est rien d'autre que le compilateur original de Rapid-Q. Pour ce qui est de l'éditeur, il y en a un inclus dans l'archive, mais vous pouvez aussi utiliser l'un des autres éditeurs existants. Dans ce cours, on considérera l'éditeur original, c'est-à-dire, celui inclus dans l'archive d'installation.

Il ne vous reste plus qu'à lancer l'éditeur RAPIDQ.exe pour démarrer ce cours.

Nota : pour télécharger l'archive contenant le compilateur et l'éditeur, rendez-vous sur cette page du site de la communauté francophone de Rapid-Q. Si vous êtes sous Windows XP, téléchargez le fichier riched32.dll pour enlever le panneau d'alerte lors du lancement de l'éditeur.

# 2. Les commentaires

Les commentaires sont toujours d'une très grande utilité. Que l'on veuille partager un code source, ou que l'on veuille pouvoir se rappeler aisément de certaines subtilités du code même si l'on ne le rouvre que plusieurs mois après sa création, il est toujours grandement recommandé de bien commenter son code.

En Rapid-Q, les commentaires se présentent tous de la même manière. Il suffit de placer une apostrophe ( ' ) avant le début du commentaire :

```
'ceci est un commentaire
```

Bien sûr, vous pouvez aussi placer un commentaire après du code, comme ceci :

```
dim form as QForm 'création d'une fenêtre
```

Il est également possible de faire des commentaires sur plusieurs lignes en répétant la méthode :

```
'voici un commentaire plutôt long
'donc on l'a mis sur deux lignes
```

Cela peut s'avérer pratique, notamment au début du fichier source, pour en indiquer le contenu.

Bien entendu, les commentaires ne sont pas pris en compte lors de compilation (génération de l'exécutable).

Cette particularité peut s'avérer des plus utiles pour débugger un programme. Si une ligne de code semble être la cause du plantage, il est possible de la commenter en ajoutant une apostrophe à son début. Tout se passe alors comme si la ligne avait été supprimée, à ceci près qu'il suffit de la décommenter (en enlevant l'apostrophe) pour la rétablir.

# 3. Premier programme

Pour débuter, ouvrez l'éditeur. Maintenant, saisissez le code suivant :

```
‘Premier programme

dim message as string
message = "Bienvenue !"

showmessage message
```

Sauvegardez-le. Attention : si vous utilisez l'éditeur original, laissez l'extension .bas à votre fichier.

Appuyez sur F5 : une fenêtre apparaît avec le texte "Bienvenue !" et un bouton OK.

Analysons un peu le code. La commande `dim` permet de créer une variable, ici de type `string`. La syntaxe est la suivante : **`dim`** `variable` **`as`** `type`

Ensuite, on stocke le texte à afficher dans la variable `message` que l'on vient de créer. Le texte doit toujours être compris entre des guillemets `"`.

Enfin, on affiche une fenêtre contenant le texte et un bouton OK. Syntaxe : **`showmessage`** `texte`

# 4. Les types de données et les variables

Comme tout langage de programmation, le Rapid-Q définit des types de données. Ceux-ci vont imposer une délimitation de la plage de valeurs que pourra prendre une variable. Autrement dit, une variable d'un type donné ne pourra prendre pour valeur que l'une de celle comprise entre la valeur minimale et la valeur maximale de ce type.

En effet, la valeur d'une variable est représentée en mémoire sous sa forme binaire. Chaque variable va donc occuper un certain nombre d'octet en mémoire. Plus le nombre d'octet qu'elle occupe est important, plus elle pourra prendre des valeurs dans un intervalle grand.

Voici les différents types supportés par le langage RapidQ, ainsi que la taille en octet de chacun de ces types et la plage de valeurs correspondantes :

`Byte` : de 0 à 255 (1 octet)
`Word` : de 0 à 65535 (2 octets)
`Short` : de -32768 à 32767 (2 octets)
`Integer` : de -2147483648 à 2147483647 (4 octets)
`Long` : de -2147483648 à 2147483647 (4 octets)
`Single` : de $1,5.10^{45}$ à $3,4.10^{38}$ (4 octets)
`Double` : de $5,0.10^{324}$ à $1,7.10^{308}$ (8 octets)

Les types `Integer` et `Long` sont équivalents. Malgré cela, les deux types sont maintenus pour cause de compatibilité avec d'autres langages.

D'autre part, il existe aussi le type `String`, qui permet de créer une variable contenant du texte, comme celle du premier programme.

Pour choisir le type d'une variable, il faut la créer à l'aide de la commande `dim` comme dans le premier programme. En Rapid-Q, la variable est à la fois déclarée et initialisée à zéro par cette commande.

Attention, une fois qu'une variable possède un type, elle ne peut en changer. Il faut donc veiller à bien choisir le type des variables d'un programme en fonction des valeurs qu'elles sont susceptibles de prendre.

Lorsqu'une variable d'un programme n'est pas déclarée, elle l'est systématiquement par le compilateur, qui lui attribue le type `integer`. Il est donc fortement préférable de toujours déclarer l'ensemble des variables utilisées dans le programme. Ainsi, vous vous éviterez de chercher une erreur qui n'est peut-être dûe qu'à un mauvais type de variable. De plus, en choisissant vous-même le type de chacune des variables, vous aurez un meilleur contrôle de votre code.

# 5. Les opérateurs

Les opérateurs sont simples et plutôt intuitifs à utiliser. Il est possible de les classifier en 3 catégories : les opérateurs arithmétiques, les opérateurs d'évaluation et les opérateurs logiques.

Tout d'abord, commençons avec les opérateurs arithmétiques :

le + pour l'addition : `2 + 3`
le * pour la multiplication : `2 * 9`
le - pour la soustraction : `1 - 3`
le / pour la division : `5 / 8`

Bien entendu, il existe aussi, l'opérateur permettant d'élever une variable ou une constante à une puissance donnée : `2 ^ 3` (ce qui équivaut à $2^3$=8).

A noter également, que le signe de l'addition + peut être remplacé par un &.

La catégorie suivante rassemble l'ensemble des opérateurs permettant d'évaluer une variable, en la comparant à une valeur de référence. Ces opérateurs sont utilisés dans toutes les structures de contrôles d'un programme.

= : égalité (ex : 2=2 sera vrai, mais pas 2=5)
<> : différence (ex : 2<>3 sera vrai, mais pas 2<>2)
< : infériorité stricte (ex : 2<3 sera vrai, mais pas 3<2, ni 2<2)
> : supériorité stricte (ex : 3>2 sera vrai, mais pas 2>3, ni 3>3)
<= : infériorité large (ex : 2<=3 sera vrai, 2<=2 aussi, mais pas 3<=2)
>= : supériorité large (ex : 3>=2 sera vrai, 3>=3 aussi, mais pas 2>=3)

Il faut distinguer la nuance entre l'inférieur strict et l'inférieur large (de même pour le supérieur). L'opérateur "large" considère comme vrai le cas d'égalité, contrairement à l'opérateur "strict", qui lui considère l'égalité comme fausse.

Enfin, les opérateurs logiques sont destinés à manipuler des bits ou à combiner les tests d'évaluation précédemment vus.

`NOT` renvoit le complément.
`AND` compare deux valeurs et renvoit 1 si elles sont identiques.
`OR` renvoit la valeur 1 si l'un ou l'autre des deux termes est à 1.
`XOR` fait de même, sauf si les deux valeurs sont à 1.

# 6. Les boucles

Les boucles sont certainement les structures les plus couramment utilisées dans un programme. En effet, imaginez que vous décidiez de faire le programme suivant :

```
dim i as integer

i = i + 1
i = i + 1
i = i + 1
i = i + 1
i = i + 1
i = i + 1
i = i + 1
i = i + 1
i = i + 1
i = i + 1
```

Bien entendu, vous pouvez faire :

```
dim i as integer

i = i + 10
```

Maintenant, si vous voulez affichez les valeurs successives de i, vous devrez utiliser la première version. Ainsi, vous allez écrire :

```
dim i as integer

i = i + 1
print i
i = i + 1
print i
i = i + 1
print i
i = i + 1
print i
i = i + 1
print i
i = i + 1
print i
i = i + 1
print i
i = i + 1
print i
i = i + 1
print i
```

Vous imaginez aisément que votre code risque de rapidement s'alourdir si vous devez effectuer des opérations répétitives. C'est là qu'interviennent les boucles. Elles vont vous permettre de condenser votre code et ainsi de le rendre plus facilement compréhensible.

## 6.1 La boucle for :

Reprenons l'exemple précédant, et utilisons la boucle for :

```
dim i as integer

for i = 1 to 10
      print i
next
```

Vous obtiendrez la même sortie qu'avec le code source précédant.

On commence par placer le mot-clé `for`. On le fait suivre du nom de la variable que l'on va utiliser pour notre boucle, ici la variable `i`. On lui indique sa valeur de départ, ici la valeur `1`. Puis, on rajoute le mot-clé `to`. Et enfin, on finit par la valeur finale que doit atteindre la variable à la fin de la boucle.

Il faut ensuite placer les instructions (lignes de codes) que l'on souhaite répéter.

Finalement, on place le mot-clé `next` pour indiquer au programme où s'arrête les instructions à exécuter.

En somme, il nous faut respecter la syntaxe suivante :

```
for variable = valeur_initiale to valeur_finale
      instructions
next
```

Dans une boucle for, la valeur est incrémentée (augmentée) une fois les instructions contenues dans la boucle exécutées.

## 6.2 La boucle do :

Une autre possibilité est d'utiliser la boucle do.

Le code devient alors :

```
dim i as integer

do
      print i
      i = i +1
loop while i < 10
```

Ici, on indique une valeur limite à notre variable, qui, lorsqu'elle sera atteinte, entraînera la fin de l'exécution de la boucle.

La syntaxe est la suivante :

```
do
      instructions
loop while condition
```

La condition se constitue d'une variable suivit d'un opérateur et enfin d'une valeur. Ainsi, il est possible de faire des boucles où la variable est décrémentée.

```
dim i as integer

i = 10

do
      print i
      i = i -1
loop while i > 0
```

Bien entendu, si vous le voulez, vous pouvez faire la même chose avec une boucle for :

```
dim i as integer

i = 10

for j = 0 to 10
      print i – j
next
```

On choisit donc d’utiliser une boucle for ou une boucle do suivant la clarté du code. En effet, dans le dernier exemple, si l’on souhaite que i porte la valeur 0 à la sortie de la boucle, il est préférable d’utiliser une boucle do. Cependant, les deux possibilités restent toutes deux correctes. Seul le contexte permet de choisir entre les deux boucles.

# 7. Les instructions conditionnelles

Bien souvent, un programme doit prendre en considération des paramètres imprévisibles par le programmeur lors de la création du programme. Par exemple, on ne peut savoir sur quel bouton l'utilisateur va cliquer, ou encore quelle action il va souhaiter effectuer en premier. Pour résoudre ce problème, le programme doit vérifier différentes conditions et effectuer les instructions codées par le programmeur en fonction de ces conditions.

Prenons le cas d'un menu. Le programme va devoir effectuer des tâches en fonctions des éléments du menu sélectionnés par l'utilisateur.

Supposons que le menu se présente sous la forme d'une liste, où chaque élément possède un numéro qui lui est propre :

Démarrer 1
Ouvrir 2
Enregistrer 3
Quitter 4

Lorsque l'utilisateur va cliquer sur l'un de ces éléments, le système d'exploitation va transmettre au programme la valeur correspondant à l'élément choisi.

Ainsi, si le programme reçoit 2, il ne devra pas faire les mêmes actions que s'il reçoit la valeur 3.

C'est là tout l'intérêt des instructions conditionnelles. Elles vont permettre à notre programme de choisir entre les différentes actions qu'il sait faire, en fonction de la valeur qu'on lui transmet.

## 7.1. Le if :

C'est la structure conditionnelle la plus simple et la plus fréquente dans un programme.

Voici sa syntaxe :

```
If condition then
      instructions
End if
```

Imaginons que nous ayons dans notre programme, une portion de code ainsi écrite :

```
numero = element_selectionne
if numero = 1 then
      showmessage "Vous avez cliqué sur Démarrer"
end if
if numero = 2 then
      showmessage "Vous avez cliqué sur Ouvrir"
end if
if numero = 3 then
      showmessage "Vous avez cliqué sur Enregistrer"
end if
if numero = 4 then
      showmessage "Vous avez cliqué sur Quitter"
end if
```

Ici, on suppose que `element_selectionne` contient le numéro de l'élément du menu que l'utilisateur a selectionné.

On récupère donc cette valeur dans une variable `numero`. Ensuite, grâce à l'utilisation du `if`, on va afficher le titre de l'élément sélectionné dans une fenêtre avec un bouton Ok.

Ainsi, à chaque fois que l'utilisateur cliquera sur un élément du menu, seule la fenêtre contenant le titre de l'élément sélectionné apparaitra.

Toutefois, il est dommage de devoir écrire une structure `if` pour chaque cas possible. Fort heureusement, on peut compresser cette écriture de la manière suivante :

```
numero = element_selectionne

if numero = 1 then
      showmessage "Vous avez cliqué sur Démarrer"
elseif numero = 2 then
      showmessage "Vous avez cliqué sur Ouvrir"
elseif numero = 3 then
      showmessage "Vous avez cliqué sur Enregistrer"
elseif numero = 4 then
      showmessage "Vous avez cliqué sur Quitter"
end if
```

Cela revient exactement au même que le code précédent. On peut ainsi regrouper plusieurs cas dépendant d'une même condition. Il faut cependant veiller à utiliser le mot-clé `elseif` pour les cas que l'on écrit entre le `if` initial et le `end if` final.

La syntaxe est donc :

```
If condition then
      instructions
elseif condition then
      instructions
End if
```

Un autre cas de figure existe encore. Imaginons que notre programme demande à l'utilisateur de répondre par oui ou par non à une question (par exemple, souhaitez-vous sauvegarder les modifications apportées au document), on pourrait alors procéder ainsi :

```
numero = element_selectionne

if numero = 1 then
      showmessage "Vous avez cliqué sur Oui"
elseif numero = 0 then
      showmessage "Vous avez cliqué sur Non"
end if
```

Dans ce cas, il n'y a que deux choix possibles. Il n'est pas alors nécessaire de vérifier les deux valeurs possibles de numero. A la place, on peut utiliser cette structure :

```
numero = element_selectionne

if numero = 1 then
      showmessage "Vous avez cliqué sur Oui"
else
      showmessage "Vous avez cliqué sur Non"
end if
```

En effet, si numero ne vaut pas 1, cela signifie qu'il vaut 0. On utilise donc le `else` qui signifie « sinon ».

La syntaxe est la suivante :

```
If condition then
      instructions
else
      instructions
End if
```

Ce genre de structure est très utilisée avec les fonctions. Effectivement, une fonction renvoit généralement une valeur pour indiquer si l'opération qu'elle devait faire a réussi ou pas.

Prenons le cas d'une fonction qui doit enregistrer un fichier. Pour cela, il va falloir d'abord créer un fichier et y écrire les données que l'on souhaite sauvegarder. Dans le cas où le fichier n'aurait pu être créé, le programme s'arrêterait brusquement lorsqu'il essaiera d'y écrire quelque chose. Il faut donc contrôler si la création a réussi ou non :

```
if creerfichier = 1 then
      'on écrit les données
else
      showmessage "Le fichier n'a pas pu être créé"
end if
```

Ici, il n'y a plus de risque de voir le programme se terminer soudainement. En cas de problème lors de la création du fichier, le programme se contentera d'en informer l'utilisateur via un message.

## 7.2. Le select :

Cette structure sert lorsqu'il y a plusieurs valeurs possibles d'une même variable servant de condition.

Reprenons notre exemple concernant le menu. A la place du code avec l'utilisation du if, nous pouvons écrire :

```
numero = element_selectionne

select case numero
      case 1
            showmessage "Vous avez cliqué sur Démarrer"
      case 2
            showmessage "Vous avez cliqué sur Ouvrir"
      case 3
            showmessage "Vous avez cliqué sur Enregistrer"
      case 4
            showmessage "Vous avez cliqué sur Quitter"
end select
```

On commence par `select case` suivi du nom la variable que l'on teste. Il suffit ensuite de mettre `case` suivi de la valeur du cas.

On a donc la syntaxe suivante :

```
Select case variable_test
      case valeur1
            instructions
      case valeur2
            instructions
      ....
      case derniere_valeur
            instructions
End select
```

Il est à noter que l'on peut également mettre en dernier cas : `case else`. Les instructions qui s'y situeront ne seront exécutées que si aucune des autres valeurs ne correspond à celle de la variable test. Cela évite que le programme ne prenne pas en compte une valeur non prévue par le programmeur.

# 8. Les types de données avancés

Nous avons vu au chapitre 4 que chaque variable possédait un type, permettant de spécifier la plage de valeurs, ou le type de valeurs qu'elle peut prendre. Néanmoins, nous n'avons pas encore parlé de deux types de données très particuliers, mais indispensables : les tableaux et les objets.

## 8.1. Les tableaux :

Les tableaux sont très utilisés en programmation. Ils vont nous permettre de stocker en mémoire de grandes quantités de données, et d'y accéder simplement.

Supposons que nous souhaitions créer un programme, qui demande l'âge de l'individu n°1, puis de l'individu n°2, etc. Puis, nous voudrions, qu'une fois ces données enregistrées, que le programme nous donne l'âge de l'individu n°i, où i serait un nombre entre 1 et le nombre total d'individus enregistrés.

Créons un premier programme pour quatre individus :

```
dim age1 as integer
dim age2 as integer
dim age3 as integer
dim age4 as integer

dim i as integer

input age1
input age2
input age3
input age4

input i 'on demande l'âge de l'individu numéro i

select case i
      case 1
            print age1
      case 2
            print age2
      case 3
            print age3
      case 4
            print age4
end select
```

Je vous laisse imaginer la longueur et la répétitivité d'un tel code dans le cas d'une centaine d'individu. C'est là que les tableaux vont être d'une grande aide.

Pour créer un tableau, il suffit de suivre cette syntaxe :

```
dim nom_tableau (nombre_d_element) as type
```

Cela revient quasiment au même que de créer une nouvelle variable, à la différence près qu'il faut indiquer entre parenthèse, à la suite du nom du tableau, le nombre d'éléments que le tableau peut contenir (il peut contenir des cases vides).

Pour accéder à la valeur d'un élément du tableau, on doit indiquer le numéro de l'élément du tableau auquel on souhaite accéder entre parenthèses :

```
nom_tableau(numero_element)
```

On accède ainsi au contenu de la case `numero_element` du tableau. Pour y inscrire quelque chose ou pour lire ce contenu, tout se passe comme si l'on manipulait une variable « classique » :

dim valeur_element as integer
dim tableau(4) as integer  ‘on crée un tableau pouvant contenir jusqu’à 4 valeurs de type integer

tableau(2)=2  ‘on place la valeur 2 dans la case 2 du tableau

valeur_element=tableau(2)  ‘on copie la valeur de tableau(2) dans valeur_element

print valeur_element

On peut alors simplifier notre premier programme :

```
dim age(1 to 4) as integer
dim i as integer

for i=1 to 4
      input age(i)
next

input i ‘on demande l’âge de l’individu numéro i

print age(i)
```

Notre code est bien plus court, et surtout, il marchera pour autant d’individu que l’on veut : il suffit de remplacer 4 par 10, 20, 50, 100, etc.

Dans cet exemple, on a écrit `dim age(1 to 4 as integer)`. En effet, on peut préciser quel numéro portera le premier élément (qui par défaut porte le numéro 0). Ici le premier élément porte le numéro 1 et le dernier, le numéro 4. Utiliser cette écriture est une bonne habitude à prendre, afin d’être sûr des numéros du premier et du dernier élément du tableau.

Les tableaux doivent contenir des valeurs correspondants à leur type. On peut créer des tableaux dans tous les types cités dans le chapitre 4.

## 8.2. Les objets :

Les objets sont un genre de données très particulier. Ils seront présents dans quasiment tous vos programmes, et vous permettront de réaliser des actions relativement complexes en toute simplicité (ou presque).

Comme pour tout type de donnée, créer un objet se fait en utilisant la commande `dim` :

dim form as QForm

Dans cet exemple, nous créons un objet (`form`) de type `QForm`, c’est-à-dire, une fenêtre. Tous les noms de types d’objet commencent par un Q.

Il faut maintenant comprendre comment fonctionne un objet. Pour cela, nous allons passer en revue les différentes catégories d’éléments qui le composent :

Les propriétés :

Comme leur nom l’indique, ce sont des valeurs qui définissent différentes propriétés d’un objet. Certaines d’entre elles peuvent être lues, d’autres écrites, d’autres encore lues et/ou écrites.
Par exemple, pour le cas de notre fenêtre, ses dimensions (largeur, hauteur) sont des propriétés. Elles peuvent être soit lues (pour connaître les dimensions de la fenêtre, par exemple), soit écrites (pour modifier ses dimensions).
Pour accéder à une propriété, on utilise la syntaxe suivante : `nom_objet.nom_propriété`

`nom_objet` est le nom attribué à l’objet lors de sa création (dans notre exemple, `form`)
`nom_propriété` est le nom de la propriété de l’objet (par exemple, `height`, pour la hauteur de la fenêtre).

Pour modifier la valeur d'une propriété, il suffit de faire comme avec une variable :

```
form.height = 400  'modifie la largeur de la fenêtre form
```

De même pour lire la valeur d'une propriété :

```
dim i as integer

i = form.height  'i contient désormais la valeur de la largeur de la fenêtre
```

Nous pouvons alors compléter notre exemple, en donnant un titre à notre fenêtre :

```
dim form as QForm

form.caption = "Une fenêtre"
```

Les méthodes :

Les méthodes vont vous permettre d'agir sur les objets. Ce sont des actions préprogrammées qui vont vous simplifier la vie. Pour utiliser une méthode, on procède comme suit :

```
form.center  'centre la fenêtre sur l'écran
```

Bien souvent les méthodes ont besoin de paramètres, c'est-à-dire, de valeurs qu'on lui transmet pour pouvoir effectuer la tâche demander. Prenons la méthode `line` d'un objet `QForm` :

```
form.line (5,5,100,5,0)  'dessine une ligne noire sur la fenêtre form
```

La méthode `line` permet de tracer une ligne sur la fenêtre. Pour le faire, elle a besoin qu'on lui précise les positions des deux points entre lesquels elle va tracer la ligne (ici entre le point de coordonnées 5,5 et celui de coordonnées 100,5), ainsi que la couleur du trait (ici, 0 pour le noir).

Complétons notre exemple du début :

```
dim form as QForm

form.caption = "Une fenêtre"
form.center
form.showmodal
```

La méthode `showmodal` permet d'afficher la fenêtre, qui va rester afficher jusqu'à ce que l'utilisateur clique sur la croix rouge de la fenêtre pour la fermer.

Les évènements :

Ils permettent de capturer les évènements (actions de l'utilisateur entre autres), et donc de pouvoir faire réagir le programme en conséquence.
Par exemple, l'objet `QForm` possède l'évènement `onclick`, qui permet de détecter de savoir que l'utilisateur a cliqué sur la fenêtre.
Nous reviendrons en détails sur les évènements lorsque nous aborderons la partie concernant la réalisation d'interface graphique.

Pour résumer, à chaque objet, on associe des propriétés, des méthodes et des évènements.

Dans le cadre de notre exemple, en utilisant les propriétés et méthodes de l'objet `QForm`, nous avons pu afficher une fenêtre, la centrer, et lui donner un titre :

```
dim form as QForm

form.caption = “Une fenêtre”
form.center
form.showmodal
```

Toutefois, vous remarquerez, qu’à chaque fois que l’on souhaite attribuer une valeur à l’objet `form`, il faut écrire `form.nom_propriété` ou `form.nom_méthode`. Vous ne pourrez pas y couper, si ce n’est qu’à une exception près : lors de la création de l’objet.

Vous pouvez alors utiliser la syntaxe du `create` :

```
Create nom_objet as type_objet
      nom_objet.propriété1
      nom_objet.propriété2
      …
      nom_objet.méthode1
      nom_objet.méthode2
      …
End Create
```

Ainsi, nous aurions pu écrire notre exemple de la manière suivante :

```
Create form as QForm
      caption = “Une fenêtre”
      center
      showmodal
End Create
```

Ou encore, en combinant :

```
Create form as QForm
      caption = “Une fenêtre”
      center
End Create

form.showmodal
```

N.B. : dans le cas d’une fenêtre, je vous encourage à faire comme dans le deuxième exemple, c’est-à-dire, de placer `form.showmodal` en fin de code, ce qui permet d’éviter certaines erreurs.

# 9. Le préprocesseur.

Le préprocesseur est une étape du processus de compilation. En pratique, on s'adresse à lui par le biais de différentes directives, qui débutent toutes par le symbole $.

Nous présenterons ici les plus utiles.

La directive `$include` :

Elle permet d'inclure le code contenu dans un fichier à l'endroit même où la directive est placée.

Prenons un premier fichier :

```
dim i as integer
dim j as integer

i = 20
j = 40
```

Et un deuxième :

```
$include "fichier1.bas"

print i + j
```

Au final, cela reviendrait à ce code :

```
dim i as integer
dim j as integer

i = 20
j = 40

print i + j
```

Vous pouvez utiliser la directive `$include` à n'importe quel endroit de votre code. Vous pourrez ainsi alléger votre code source (d'un point de vue visuel), en plaçant par exemple vos `subs` et `functions` dans un fichier d'inclusion.

La directive `$typecheck` :

`$typecheck` a par défaut la valeur `OFF`. Si vous mettez la valeur `ON`, toutes les variables devront être déclarées (en utilisant `dim`) avant de les utiliser.

```
$typecheck on

dim i as integer  'si on ne declare pas i, il y aura une erreur de générer

i = 20

print i
```

La directive `$option` :

Cette directive peut être suivie de différentes valeurs.

`$option explicit` donne le même effet que `$typecheck on`,

`$option icon "path\file.ico"` permet d'ajouter une icône à votre programme,

`$option vbdll on` permet de déclarer les dll de la même manière qu'en VB,

`$option GTK` active l'utilisation de GTK.

# 10. Les constantes.

Voici un mini-chapitre sur les constantes.

Lorsque vous déclarez une variable, vous pouvez demander au compilateur d'en faire une constante, c'est-à-dire, une variable dont la valeur ne changera pas au cours du programme.

Vous éviterez ainsi certaines erreurs, lorsqu'une valeur précise doit être utilisée à plusieurs reprises dans le programme.

Voici un exemple d'utilisation d'une constante :

```
const pi=3.14 as float

input i

print 2*pi*i
```

Le code parle de lui-même. Vous pouvez créer des constantes de tout type numérique.

# 11. Les sub et les function.