Mes premiers pas

2

Ce chapitre est destiné à des personnes n'ayant aucune expérience de programmation. Il s'agit d'un chapitre de "recettes de cuisine" à suivre pas à pas. La plupart des notions abordées ici sont revues dans les autres chapitres.

Plutôt que d'étudier une longue liste rébarbative de commandes et de fonctionnalités du langage Eiffel, dans ce chapitre, on suivra le développement d'une classe décrivant un atome. L'atome est en effet la brique élémentaire servant pour la construction de toute molécule.

1. Ma première classe

1.1. Saisie du texte

A l'aide de l' éditeur de texte SciTE (ou votre éditeur favori), créez un nouveau fichier « atom.e » (nom en minuscules avec l'extension « .e »), puis saisissez le texte suivant.

```
class ATOM

creation
→   make

feature
→   make is
→   →   do
→   →   →   io.put_string("I'm a new atom!")
→   →   →   io.put_new_line
→   →   end

end -- end of class ATOM
```

Remarque: Il est impératif de respecter la casse (majuscule/minuscule) pour recopier le texte précédent.

Remarque: le caractère →symbolise une tabulation (touche Tab). Il faut configurer votre éditeur de texte pour que la tabulation équivaut à quatre espaces.

Après avoir sauvegardé votre texte, lancez la compilation en cliquant sur *Outils* > *Compiler* dans SciTE (ou utilisez le raccourci clavier Ctrl+ F7). La phrase suivante s'affiche dans la fenêtre inférieure. Sinon, dans un shell (ou une invite de commandes), tapez la ligne suivante:

```
se c atom.e -o atom -clean
```

Si le compilateur ne proteste pas, un fichier exécutable (votre programme) nommé *atom* (ou

atom.exe sous Winxx) est créé. Il reste à le lancer en cliquant sur *Outils* > *Exécuter* (ou taper le raccourci clavier F5) . La phrase suivante s'affiche dans la fenêtre d'état :

```
./atom
```

Si tout s'est correctement déroulé, vous devez lire dans la fenêtre d'état, la phrase:

```
I'm a new atom!
```

Félicitations, vous venez de créer votre premier atome!

1.2. Comment ça marche?

Une classe dans sa version la plus simple nécessite l'utilisation de quatre mots clés comme le montre la syntaxe suivante:

```
class <NOM_DE_LA_CLASSE>

creation

feature

end -- end of class <NOM_DE_LA_CLASSE >
```

Dans l'exemple précédent, il existe une seule façon de créer un objet[1] à partir de la classe ATOM et c'est grâce à la routine *make*; on appelle ces routines, des **constructeurs**. De plus, la classe ATOM (à écrire **obligatoirement** en majuscules) possède dans la rubrique *feature* une seule routine *make* qui permet d'afficher la phrase "I'm a new atom!".

A l'exécution du programme, SmartEiffel crée un objet de classe ATOM avec le constructeur *make*. Comme *make* contient des instructions permettant l'affichage de la phrase « I'm a new atom! », celle-ci s'affiche à l'écran.

1.2.1. Les conventions de définition de classes en Eiffel

La définition de la classe est délimitée d'une part par le mot clé ***class*** qui indique le début et d'autre part par le mot clé ***end*** qui termine la définition de la classe. Il est d'usage de mettre un commentaire à la fin de la ligne indiquant que c'est la fin de la classe. Ce commentaire est de la forme:

```
-- end of class [nom_de_la_classe].
```

A l'intérieur de la classe, on trouve deux rubriques: (i) la première annoncée par le mot clé *creation* permet d'énumérer les différents moyens de créer un objet à partir de cette classe. (ii) la deuxième rubrique *feature* (*caractéristique* en anglais) contient la définition des attributs et des routines (blocs de code permettant de définir le comportement de l'objet).

Les conventions suivantes doivent être respectées:

- ✔ Le nom de la classe est toujours en majuscule.
- ✔ Le nom du fichier texte doit être identique au nom de la classe mais en minuscules et avec l'extension « .e ». Par exemple, la classe MA_SEQUENCE doit être saisie dans le fichier ma_sequence.e
- ✔ Un fichier texte ne contient qu'une classe.
- ✔ Les caractères accentués ne sont pas autorisés dans le nom des fichiers et des classes.

1 On dit aussi *instance* c'est à dire un exemplaire créé à partir de la classe – le plan – ATOM

1.2.2. La rubrique creation

Elle indique quel(s) est(sont) le(s) constructeur(s) de la classe, c'est à dire les routines associées lors de la création de l'objet. Un constructeur est une routine qui sert à l'initialisation de l'objet lors de sa création. Dans la classe ATOM, il existe un constructeur *make* qui est automatiquement reconnu par le compilateur comme étant la routine principale (équivalent de main() en langage C) à exécuter en premier.

Une routine appelée aussi méthode dans d'autres langages à objets soit se conformer à la syntaxe suivante:

```
nom_de_la_routine is
→   →   →   -- On peut ajouter un commentaire
→   →   →   -- sur plusieurs lignes
→   →   →   -- si on veut
→   →   do
→   →   end
```

La première ligne correspond à la déclaration de la routine. Le bloc délimité par **do** et **end** serviront à programmer une action (un comportement) en tapant plusieurs lignes d'instructions compréhensibles pour le compilateur.

Remarque: Rien n'empêche d'utiliser un autre nom de constructeur mais il faudra le spécifier au compilateur. Par exemple, si on remplace *make* par *my_atom* dans le fichier atom.e, il faudra taper la commande suivante:

```
compile atom.e my_atom -o atom
```

1.2.3. La rubrique feature

Elle contient la définition proprement dite de la classe. On y trouve les attributs et les routines.

Dans la classe ATOM, il n'y a qu'une routine correspondant à la définition du constructeur *make*.

1.2.4. Autres conventions

Un commentaire est une ligne commençant par deux tirets « -- ». Si on souhaite faire des commentaires sur plusieurs lignes, on doit répéter les deux tirets au début de chaque nouvelle ligne.

2. Plusieurs atomes...

Les atomes étant reliés par des liaisons chimiques, créons une classe BOND.

2.1. Et une liaison, une !

La classe BOND contient deux atomes qui seront caractéristiques de notre .

Ouvrez un fichier bond.e (dans le même dossier que atom.e) et saisissez les lignes suivantes:

```
class BOND

creation
→   make

feature
```

```
	with(atom1, atom2 : ATOM) is
		do
			first_atom := atom1
			second_atom := atom2
			io.put_string("I'm a new bond composed of :%N")
			io.put_string(atom1.out + "%N")
			io.put_string(atom2.out + "%N")
		end

	first_atom : ATOM
	second_atom : ATOM

end -- end of class BOND
```

Pour utiliser la classe BOND, il nous faut un troisième fichier. Ouvrez un nouveau fichier nommé "test.e" (toujours dans la même directory que *atom.e* et *bond.e*) et tapez les lignes suivantes:

```
class TEST

creation
	make

feature
	make is
		local
			atom1, atom2 : ATOM
			a_bond : BOND
		do
			create atom1.make
			create atom2.make
			create a_bond.with(atom1, atom2)
		end

end -- end of class TEST
```

La classe BOND contient deux atomes nommés *first_atom* et *second_atom.* On appelle ces deux variables, des attributs de la classe BOND. Le constructeur de la classe BOND est une routine nommé *with* qui "attend" deux arguments de type ATOM. n objet de la classe ATOM et au moment de sa création (dans le constructeur *make*, on crée avec le mot clé *create* l'objet *first_atom* avec le constructeur de ATOM qui se nomme *make*.

Il faut lancer la compilation en tapant Ctrl+F7, la ligne suivante doit s'afficher:

```
compile test.e -o bond_test -clean
```

Puis exécuter le programme avec F5:

```
./bond_test
```

Si tout est correct, la ligne suivante s'affiche:

```
I'm a new bond composed of :
ATOM#0x9eee040[]
ATOM#0x9eee088[]
```

Les chiffres hexadécimaux peuvent varier et indiquent simplement que les objets de classe ATOM existent bien quelque part en mémoire.

2.1.1. Un peu de vocabulaire...

Pour pouvoir utiliser *first_atom*, il faut indiquer au compilateur à quel type d'objet correspond cette variable (appelé aussi attribut); cela constitue la **déclaration**. On **déclare**, ici, un **attribut** nommé *first_atom* comme étant un objet de classe ATOM. *first_atom* est aussi appelé une **variable d'instance**. La déclaration doit se faire en respectant la syntaxe suivante:

```
nom_attribut : <Nom_de_la_classe>
```

3. Des attributs et des méthodes

3.1. On modifie....

Un atome est défini par sa position 3D exprimée par ses coordonnées atomiques (x,y,z). On va donc ajouter trois nouvelles caractéristiques (*feature* en anglais) à la classe ATOM.

3.1.1. Les attributs de ATOM

La classe ATOM aura les caractéristiques suivantes:

- ✔ x
- ✔ y
- ✔ z

On appelle ces caractéristiques, des **attributs**. Comme en Eiffel, tout est objet, on doit indiquer à quelle classe appartiennent ces attributs: *x, y, z* sont des nombres "à virgule" (plus mathématiquement parlant des nombres réels); on les définit comme des objets de classe REAL.

Modifions la classe ATOM de la manière suivante:

```
class ATOM

creation
	make

feature
	make is
		do
			io.put_string("I'm a new atom with coordinates:")
		end

	x : REAL

	y : REAL

	z : REAL


end -- end of class ATOM
```

Trois nouvelles lignes apparaissent dans la classe ATOM:

```
x : REAL
y : REAL
z : REAL
```

Ces trois lignes correspondent à la déclaration ainsi que nous l'avons vu pour *first_atom*.

Les divers objets en interaction dans la classe TEST sont présentés dans le schéma ci-dessous:

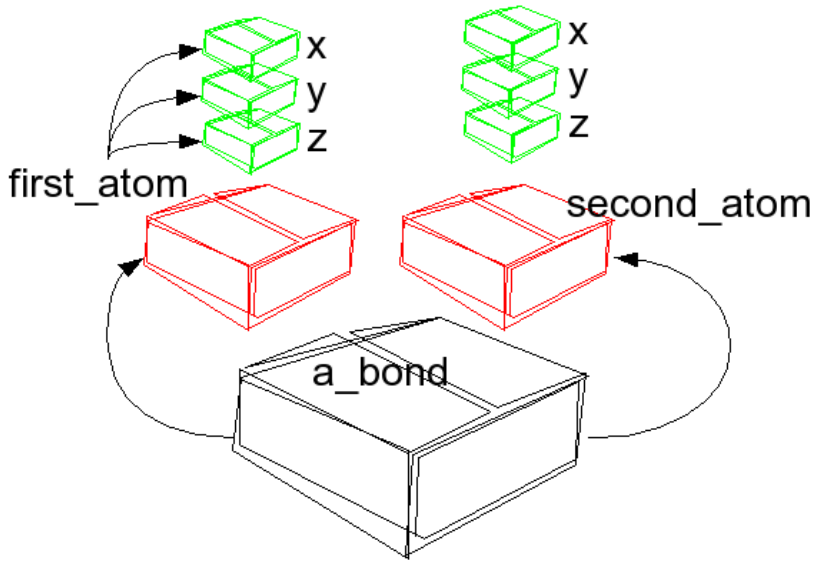

Figure xxx: Contenu de la classe TEST. Les objets sont représentés ici par des boîtes qui contiennent des attributs.

Sauvez la modification (Ctrl+S); puis dans *test.e,* complétez le fichier avec les lignes suivantes: ....

```
class TEST

creation
    make

feature
    make is
        local
            atom1, atom2 : ATOM
            a_bond : BOND
        do
            create atom1.make
            create atom2.make

            atom1.x := 10.0

            create a_bond.with(atom1, atom2)
        end

end -- end of class TEST
```

Compilez *test.e* (Ctrl + F7) ... ☹ malheureusement, le compilateur indique une erreur au niveau de la nouvelle ligne "atom1.x := 10.0". En effet, il est impossible de modifier **directement** la valeur d'un attribut d'un objet en Eiffel, il faut passer par des routines nommées "accesseurs" qui sont des sortes de sas permettant de faire transiter des valeurs de/vers les attributs d'un objet. Pour pouvoir accéder aux attributs d'un objet, il faut pouvoir "ouvrir" la boîte pour en modifier (lire ou écrire) le contenu.

3.1.2. Accéder aux attributs

Dans atom.e, on ajoute ces deux « accesseurs" (pour d'une part, lire l'attribut et d'autre part écrire dans l'attribut) qu'on appelle get_x et set_x.

```
class ATOM

creation
→   make

feature
→   make is
→   →   do
→   →   →   io.put_string("I'm a new atom with coordinates:")
→   →   end

→   x : REAL
→   y : REAL
→   z : REAL

feature -- Access

→   set_x(value : REAL) is
→   →   do
→   →   →   x := value
→   →   end

→   get_x : REAL is
→   →   do
→   →   →   Result := x
→   →   end

end -- end of class ATOM
```

Par habitude, les noms des accesseurs sont souvent composés des préfixes "set_" ou "get_" + nom de l'attribut. L'accesseur "set_" permet d'écrire une valeur dans l'attribut de l'objet et "get_" de lire l'attribut.

L'accesseur *set_x* est aussi une routine comme le constructeur *make* et on retrouve d'ailleurs la même syntaxe constituée des mots clés **is do** ... **end**. Toutefois, en plus on trouve entre parenthèses une liste du (des) argument(s) de la routine qui est constituée de la déclaration des variables de la routine. On peut compléter notre syntaxe d'une routine:

```
nom_de_la_routine(arg1 : MY_CLASS; arg2, arg3, arg4 : YOUR_CLASS) is
→   →   →   -- On peut ajouter un commentaire
→   →   →   -- sur plusieurs lignes
→   →   →   -- si on veut
→   →   do
→   →   end
```

Remarque 1: quand la routine n'a pas d'arguments, on ne met pas de parenthèses. Par exemple, *make* n'a pas d'arguments, il est simplement défini comme *make* **is**

Remarque 2: quand la routine a plusieurs arguments, ils sont séparés par des points virgules ";" . Par exemple, (arg1 : MY_CLASS; arg2 : YOUR_CLASS; arg3 : OUR_CLASS)

Remarque 3: quand la routine a plusieurs arguments et qu'ils appartiennent à la même classe, ils sont séparés par des virgules "," puis la liste est terminée par ":" NOM_DE_CLASSE. Par exemple, (arg1, arg2, arg3 : MY_CLASS; arg4 : OUR_CLASS).

Ici, il existe un seul argument nommé *value* de classe REAL.

```
→   set_x(value : REAL) is
→   →   do
→   →   →   x := value
→   →   end
```

Ensuite, le bloc de code contient une seule ligne qui indique que la valeur contenue dans la variable value sera affecté à la variable x. Le signe ":=" est le symbole d'affectation en Eiffel et permet d'initialiser ou de modifier le contenu de la variable x.

En clair, cela signifie que pour fonctionner correctement *set_x* attend un nombre réel qu'il placera ensuite dans la variable x.

L'accesseur g*et_x* est aussi une routine qui ne possède aucun argument (il n'y a pas de parenthèses juste après son nom). Par contre, cette routine permet de lire la valeur de x; elle doit donc retourner une valeur de classe REAL, c'est pourquoi on ajoute à la déclaration ":" REAL. On appelle ces routines, des **fonctions**.

```
nom_de_la_fonction(a : MY_CLASS; b, c, d : YOUR_CLASS) : OUR_CLASS is
→   →   →   -- On peut ajouter un commentaire
→   →   →   -- sur plusieurs lignes
→   →   →   -- si on veut
→   →   do
→   →   end
```

Remarque: En fait, il s'agit de la même syntaxe que la déclaration d'une variable. Il n'y a pas de différence entre une variable et une routine retournant une valeur (hormis le mot clé **is** terminal).

A l'intérieur du bloc délimité par **do** et **end**, il faut indiquer que le résultat doit être retournée par la fonction: c'est le rôle de la variable locale nommée **Result.** Elle n'a pas besoin d'être explicitement déclarée, car elle possède le même type (classe) que celui de la fonction.

Modifiez bond.e et ajoutez les lignes suivantes après la création de *first_atom*

```
first_atom.set_x(10.5)
io.put_real(first_atom.get_x)
io.put_new_line
```

Compilez (Ctrl+ F7) et exécutez (F5), il s'affiche:

```
I'm a new atom with coordinates: 10.0
```

Exercice 1: Complétez cette classe pour implanter les accesseurs de y et z.

Exercice 2: Ajoutez à la classe ATOM, une routine *set_coordinates(xx, yy, zz : REAL)* qui affectent les valeurs de xx, yy, et zz à x, y et z, respectivement. Modifiez BOND pour tester votre nouvelle routine.

3.2. On ajoute ....une routine

Modifions la classe ATOM de la manière suivante:

```
class ATOM
```

```
creation
→   make

feature
→   make is
→   →   do
→   →   end

→   display_contents is
→   →   do
→   →   →   io.put_character('(')
→   →   →   io.put_real(x); io.put_character(';')
→   →   →   io.put_real(y); io.put_character(';')
→   →   →   io.put_real(z);
→   →   →   io.put_character(')')
→   →   →   io.put_new_line
→   →   end

→   x : REAL

→   y : REAL

→   z : REAL

end -- end of class ATOM
```

Puis, dans la classe BOND, remplacez dans le constructeur, les lignes suivantes:

```
→   →   →   io.put_string(atom1.out + "%N")
→   →   →   io.put_string(atom2.out + "%N")
```

par

```
→   →   →   atom1.display_contents
→   →   →   atom2.display_contents
```

Puis, dans test.e, modifiez le constructeur make par les lignes:

```
create first_atom.make
create second_atom.make
first_atom.set_coordinates(10.0,20.05,2005.0)
```

Sauvegardez (Ctrl+S), compilez (Ctrl + F7) et exécutez (F5), il s'affiche:

```
I'm a new bond composed of :
(10.0;20.05;2005)
(0.0;0.0;0.0)
```

Dans ce nouvel exemple, on définit une routine *display_contents*

Le problème de notre classe ATOM est que les coordonnées de tous les atomes sont rigoureusement identiques. Il serait plus logique de pouvoir au moment de la création d'un objet de classe ATOM d'affecter des coordonnées différentes.

On va donc créer un autre constructeur nommé *with* qui possède trois arguments de classe REAL correspondant respectivement à nos coordonnées x, y et z.

```
class ATOM

creation
→   make, with
```

```
feature
    with(xx, yy, zz : REAL) is
        do
            set_coordinates(x,y,z)
        end

    make is
        do
        end

    display_contents is
        do
            io.put_character('(')
            io.put_real(x); io.put_character(';')
            io.put_real(y); io.put_character(';')
            io.put_real(z);
            io.put_character(')')
            io.put_new_line
        end

    x : REAL

    y : REAL

    z : REAL

end -- end of class ATOM
```

Puis mettez à jour BOND en remplaçant le constructeur *make* par *with* pour *second_atom:*

```
create first_atom.make
create second_atom.with(1.0,3.0,6.0)
```

Exercice 4: Ajoutons une deuxième routine qui permet d'appliquer une translation aux coordonnées de l'atome. Cette routine nommée translate possède un argument *value* de classe REAL. Cette routine ajoute à aux variables d'instance x,y et z de ATOM, une quantité correspondant à la valeur de *value*.

3.3. On ajoute ....une fonction

Dans *bond.e,* ajoutez une nouvelle routine *length*:

```
    length : REAL is
        local
            dx, dy, dz : REAL
        do
            dx := (second_atom.get_x - first_atom.get_x)
            dy := (second_atom.get_y - first_atom.get_y)
            dz := (second_atom.get_z - first_atom.get_z)
            Result := (dx * dx + dy * dy + dz * dz).sqrt
        end
```

Pour tester cette routine, on crée un fichier *test.e* qui va permettre de créer un objet de classe BOND à partir de deux atomes puis de calculer la longueur de cette liaison:

```
class TEST

creation
	make

feature
	make is
		local
			atom1, atom2 : ATOM
			a_bond : BOND
		do
			create atom1.with(5.287, 16.725, 4.830)
			create atom2.with(5.776, 17.899, 5.595)
			create a_bond.with(atom1, atom2)
			io.put_real(a_bond.length)
			io.put_new_line
		end

	an_atom : ATOM

end -- end of class TEST
```

Sauvegardez (Ctrl+S), compilez (Ctrl + F7) et exécutez (F5), la longueur de la liaison s'affiche:

```
compile test.e -o test -clean
```

Contrairement à la routine *add* précédente, *add_between* s'utilise différemment.

```
i := add_between(2005,5)
```

Elle a besoin de deux arguments 2005 et 5 et de plus, le résultat de l'opération doit être placée dans une variable (ici, i). Ce type de routine est appelé fonction car elle retourne une valeur (pas forcément de type INTEGER, n'importe quelle classe convient).

Par rapport à *add*, la routine *add_between* a une déclaration légèrement différente:

- Pour la déclaration des arguments:
  - ✔ Par défaut, on déclare les arguments comme des attributs (c'est à dire en écrivant le nom de l'argument suivi du séparateur ":" et terminé par le nom de la classe) qu'on sépare par un point virgule ";". Exemple: a : INTEGER; b : CHARACTER; c : INTEGER
  - ✔ Toutefois lorsque les arguments sont de même classe: il suffit de les énumérer et de les séparer par une virgule "," et terminer cette énumération par le séparateur ":" et le nom de la classe. Exemple: a, b, c : CHARACTER
  - ✔ Toutes les combinaisons sont possibles. Exemple: a, b : INTEGER; c, d, e : CHARACTER
- Après la déclaration des arguments, il y a en plus un séparateur ":" (deux points) suivie de la classe retournée (ici INTEGER); le mot clé **is** terminant la déclaration de la fonction.

4. La grande boucle

4.1. Du nouveau...

Complétons la classe ATOM et ajoutons lui deux nouveaux attributs:

- ✔ *name* correspond au nom de l'atome selon la nomenclature de la Protein Data Bank et peut contenir un ou plusieurs caractères, ce qui correspond à une chaîne de caractères: on appelle cela une STRING.

✔ *symbol* est le symbole chimique de l'atome et on se contentera des 6 atomes les plus courants: C (carbone), H (hydrogène), O (oxygène), N (azote), S (soufre) et P (phosphore). Comme *symbol* ne contient qu'un seul caractère, on le déclare comme CHARACTER.

```
class ATOM

creation
	with

feature
	with_symbol(a_symbol : CHARACTER; xx, yy, zz : REAL) is
		do
			x := xx
			y := yy
			z := zz
			symbol := a_symbol
		end

	display_contents is
		do
			io.put_character('(')
			io.put_real(x); io.put_character(';')
			io.put_real(y); io.put_character(';')
			io.put_real(z);
			io.put_character(')')
			io.put_new_line
		end

	x : REAL

	y : REAL

	z : REAL

	symbol : CHARACTER -- Example: C, H, O, N, S or P

	name : STRING -- Example: CA, CZ2, O2*, NE2, SG

feature -- Access

	-- To do implementation of accessors

end -- end of class ATOM
```

4.2. from ... until ... loop ... end

En Eiffel, la boucle est composée de trois parties:

Une zone d'initialisation encadrée par **from** et **until**

Une zone contenant une condition permettant d'interrompre l'exécution de la boucle

Le corps de la boucle encadrée par **loop** et **end**

```
class TEST

creation
```

```
    make

feature
    make is
        local
            atom1, atom2, atom3, atom4 : ATOM
            atom_array : ARRAY[ATOM]
            a_bond : BOND
            i : INTEGER
        do
            -- Create an array of 4 atoms
            create atom1.with(5.287, 16.725, 4.830)
            create atom2.with(5.776, 17.899, 5.595)
            create atom3.with(7.198, 18.266, 5.104)
            create atom4.with(7.301, 19.067, 4.161)
            atom_array := << atom1, atom2, atom3, atom4 >>

            -- Calculate all the lengths between atom1 and the others
            from i := atom_array.lower
            until i > atom_array.upper
            loop
                create a_bond.with(atom1, atom_array.item(i))
                io.put_real(a_bond.length); io.put_new_line
                i := i + 1
            end


end -- end of class TEST
```

La classe TEST montre un exemple d'utilisation de la boucle permettant d'accomplir une tâche répétitive (ici, le calcul de longueur de liaison). La routine *make* contient deux parties: (i) la création et la définition d'un tableau *atom_array* contenant quatre atomes et (ii) l'utilisation d'une boucle **from ... until ... loop** pour calculer toutes les longueurs de liaison entre *atom1* et les autres atomes du tableau.

Exercice: Modifiez la boucle pour calculer toutes les combinaisons possibles de longueurs de liaisons en évitant de calculer les longueurs des liaisons composées d'atomes identiques (ex : *atom1* <-> *atom1*).

```
class ATOM_SET_2

creation
    make

feature
    make is
        local
            i : REAL
            an_atom : ATOM
        do
            from  i := 1
            until i >  10
            loop
                create an_atom.with('C', i, i + 1, i + 2)
                an_atom.display_contents
                io.put_new_line
                i := i + 1
            end
```

```
    end


end -- end of class ATOM_SET_2
```

5. Avec des si,...

```
class BOND

creation
    with

feature -- Access

    get_type : STRING is
        do
            if first_atom.get_symbol = 'S' and
               second_atom.get_symbol = 'S' then
               Result := "Disulfide Bridge"
            elseif first_atom.get_symbol = 'H' and
               second_atom.get_symbol = 'O' then
               Result := "Hydrogen Bond"
            elseif first_atom.get_symbol = 'O' and
               second_atom.get_symbol = 'H' then
               Result := "Hydrogen Bond"
            else
               Result := "Unknown"
            end
        end


end -- end of class BOND
```

```
class TEST

creation
    make

feature
    make is
        local
            atom1, atom2 : ATOM
        do
            create atom1.with_symbol('S', 1.0, 2.0, 3.0)
            create atom2.with_symbol('S', 5.776, 17.899, 5.595)
            create a_bond.with(atom1, atom2)
            io.put_string(a_bond.get_type)
            io.put_new_line
        end


end -- end of class TEST
```

Exercice 8: Modifiez *make* pour pouvoir afficher par ordre décroissant les nombres divisibles par 3 et par 7 de 100 jusqu'à 0.

Exercice 9: Modifiez *make* pour pouvoir afficher par ordre décroissant les nombres divisibles par 3 ou par 7 de 100 jusqu'à 0.

Exercice 10: Modifiez *make* pour pouvoir afficher par ordre décroissant les nombres qui ne sont ni divisibles par 3 ni par 7 de 100 jusqu'à 0.

6. Un peu plus loin...

6.1. But: Construire une classe ATOM

Une protéine est constituée d'atomes qui sont définis par leur symbole chimique (C, H, O, N, Na, Cl, Ca, Fe, etc.) et par leurs coordonnées atomiques (x, y, z).

6.1.1. Les attributs de ATOM

La classe ATOM aura les caractéristiques suivantes:

- ✔ symbol
- ✔ x
- ✔ y
- ✔ z

On appelle ces caractéristiques, des **attributs**. Comme en Eiffel, tout est objet, on doit indiquer à quelle classe appartiennent ces attributs:

- ✔
- ✔ *x, y, z* sont des nombres "à virgule"; on les définit comme des objets de classe REAL.

6.1.2. Un constructeur pour ATOM

Il nous faut un **constructeur** pour pouvoir initialiser notre objet au moment de sa création, c'est à dire définir son symbole chimique et ses coordonnées atomiques. Pour cela, on va définir un constructeur nommé *with.*

6.1.3. Une routine pour visualiser le contenu d'un objet de classe ATOM

Pour savoir ce qu'on a réellement créé, on va définir une routine *display_contents* qui comme son nom l'indique affichera le contenu de notre objet de classe ATOM.

6.2. Ecrire la classe ATOM

On peut passer à l'écriture en ouvrant d'abord un fichier atom.e:

```
class ATOM

creation
    with

feature
    with(a_symbol: STRING; xx, yy, zz : REAL) is
        do
            symbol := a_symbol
            x := xx
            y := yy
            z := zz
```

```
	end

	symbol : STRING

	x : REAL

	y : REAL

	z : REAL

	display_contents is
	do
		io.put_string(symbol)
		io.put_new_line
		io.put_real(x)
		io.put_new_line
		io.put_real(y)
		io.put_new_line
		io.put_real(z)
		io.put_new_line
	end


end -- end of class ATOM
```

Enregistrez le fichier (Ctrl+S).

Quelques explications sont nécessaires:

Le constructeur *with* possède quatre arguments nommés a_symbol, xx, yy et zz placés entre parenthèses. De plus, on indique la classe de chaque argument

6.3. Classe TEST

La classe TEST va nous permettre de créer une instance de la classe ATOM. Pour cela, on déclare *an_atom* comme appartenant à la classe ATOM et dans le constructeur *make*, on crée avec le mot clé *create* l'objet *an_atom* avec le constructeur de ATOM qui se nomme *with*.

Ouvrez un fichier test.e (dans le même dossier que atom.e) et saisissez les lignes suivantes:

```
class TEST

creation
	make

feature
	make is
	do
		create an_atom.with("Fe",1.0,2.0,3.0)
		an_atom.display_contents
	end

	an_atom : ATOM

end -- end of class TEST
```

Il faut lancer la compilation en tapant Ctrl+F7, la ligne suivante doit s'afficher:

```
compile test.e -o test -clean
```

Puis exécuter le programme avec F5:

Si tout est correct, les quatre lignes suivantes s'affichent:

```
Fe
1.000000
2.000000
3.000000
```

7. Et maintenant...

8. glossaire

attribut

constructeur

déclaration

instance

routine

variable

variable d'instance

9. Solutions

9.1. Exercice 4:

Ajoutons une deuxième routine qui permet d'appliquer une translation aux coordonnées de l'atome. Cette routine nommée translate possède un argument *value* de classe REAL. Cette routine ajoute à aux variables d'instance x,y et z de ATOM, une quantité correspondant à la valeur de *value*.

```
	translate(value : REAL) is
		do
			x := x + value
			y := y + value
			z := z + value
		end
```

9.2. Exercice 8:

Sujet: Modifiez *make* pour pouvoir afficher par ordre décroissant les nombres divisibles par 3 et par 7 de 100 jusqu'à 0.

```
class TEST

creation
	make

feature
	make is
		do
			from  i := 100
			until i <=  0
			loop
				if (i // 3 = 0) and (i // 7 = 0) then
					io.put_integer(i)
					io.put_new_line
				end
				i := i - 1
			end -- end of loop
		end

end -- end of class TEST
```

9.3. Exercice 9:

Sujet: Modifiez *make* pour pouvoir afficher par ordre décroissant les nombres divisibles par 3 ou par 7 de 100 jusqu'à 0.

```
class TEST

creation
	make

feature
	make is
		do
			from  i := 100
			until i <=  0
			loop
				if (i // 3 = 0) or (i // 7 = 0) then
					io.put_integer(i)
					io.put_new_line
```

```
→   →   →   →   end
→   →   →   →   i := i - 1
→   →   →   end -- end of loop
→   →   end

end -- end of class TEST
```

9.4. Exercice 10:

Sujet: Modifiez *make* pour pouvoir afficher par ordre décroissant les nombres qui ne sont ni divisibles par 3 ni par 7 de 100 jusqu'à 0.

```
class TEST

creation
→   make

feature
→   make is
→   →   do
→   →   →   from  i := 100
→   →   →   until i <=  0
→   →   →   loop
→   →   →   →   if not ( (i // 3 = 0) and (i // 7 = 0) )  then
→   →   →   →   →   io.put_integer(i)
→   →   →   →   →   io.put_new_line
→   →   →   →   end
→   →   →   →   i := i - 1
→   →   →   end -- end of loop
→   →   end

end -- end of class TEST
```

Remarque: on peut aussi mettre

```
if (i // 3 /= 0) and (i // 7 /= 0) then
```