9 Comment créer une bibliothèque ?

Le jour viendra où vos connaissances vous permettront de réaliser des idées à vous, que d'autres n'auront peut-être pas encore eues. Mais peut-être souhaiterez-vous aussi améliorer un projet existant parce que votre solution est plus élégante et moins compliquée à transposer. D'innombrables développeurs de logiciels se sont penchés avant vous sur les questions les plus diverses et ont programmé des bibliothèques pour épargner du travail et du temps aux autres développeurs. À travers ce projet, nous allons découvrir les grands principes de ces bibliothèques et leur création. Si le langage de programmation C++ – programmation orientée objet incluse – vous a toujours intéressé, vous allez être servi !

Les bibliothèques

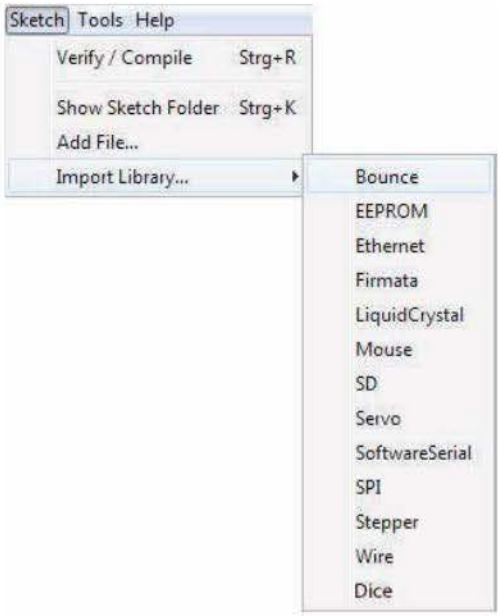
Une fois l'environnement de développement Arduino installé ou plutôt décompressé, vous disposez de quelques bibliothèques maison prêtes à l'emploi, appelées également librairies (*libraries* en anglais). Elles traitent de thèmes intéressants, tels que commander :

- un servomoteur ;
- un moteur pas-à-pas ;
- un écran LCD ;
- une EEPROM externe pour stocker des données…

Ces bibliothèques sont stockées dans le répertoire `libraries` du répertoire d'installation d'Arduino. Vous pouvez utiliser Windows Explorer ou passer par l'environnement de développement Arduino pour savoir quelles sont les bibliothèques disponibles. On y trouve

une entrée de menu spéciale `Sketch>Import Library` permettant d'afficher la liste correspondante.

Figure 9-1 ▶
Afficher ou importer des bibliothèques



Les entrées du menu coïncident avec les répertoires du dossier `libraries`. Tout cela est bien beau, mais voyons d'abord comment une bibliothèque Arduino fonctionne et ce que nous pouvons faire avec.

Qu'est-ce qu'une bibliothèque exactement ?

Avant de passer à un exemple concret, vous devez savoir ce qu'est une bibliothèque. J'ai dit déjà qu'elle servait quasiment à empaqueter et réunir des tâches de programmation plus ou moins complexes en un paquet de programme. La figure 9-2 montre la coopération entre une bibliothèque Arduino et l'API Arduino.

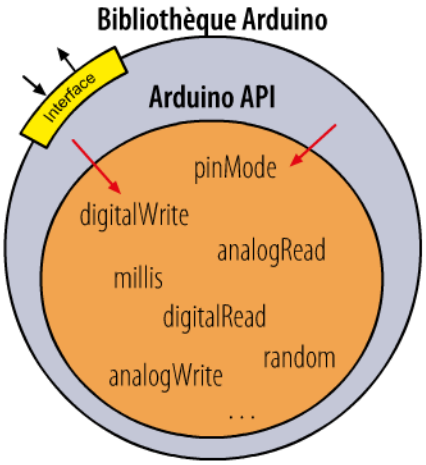


Figure 9-2
Comme fonctionne une bibliothèque Arduino ?

Nous avons affaire à deux couches de programme interdépendantes. Je procède de l'intérieur vers l'extérieur. J'ai appelé la couche interne API Arduino. (API est l'abréviation d'*Application Programming Interface* et une interface vers toutes les instructions Arduino disponibles.) Je n'en ai sélectionné que très peu par manque de place. La couche externe est constituée par la bibliothèque Arduino, qui enveloppe la couche interne. Elle est de ce fait appelée *wrapper* (enveloppe) et se sert de l'API Arduino. Pour pouvoir accéder à la couche wrapper, une interface doit être mise en œuvre car vous entendez bien sûr exploiter la fonctionnalité d'une bibliothèque. Une interface est un portail d'accès à l'intérieur de la bibliothèque, qui est en soi une unité fermée. Le terme technique est encapsulation. Vous allez bientôt voir en détail de quoi il s'agit et en quoi cela concerne le langage de programmation C++.

En quoi les bibliothèques sont-elles utiles ?

Question idiote à laquelle j'ai déjà répondu plusieurs fois. Aussi me contenterai-je ici de vous en rappeler les avantages.

- Pour ne pas avoir à « réinventer la roue » chaque fois, les développeurs ont trouvé un moyen de stocker le code de programme dans une bibliothèque. Beaucoup de programmeurs dans le monde profitent de ces structures logicielles, qu'ils peuvent utiliser sans problème dans leurs propres projets. Le mot-clé est ici réutilisation.
- Une fois testée et débarrassée de ces erreurs, une bibliothèque peut être utilisée sans en connaître les déroulements internes. Sa

fonctionnalité est encapsulée et cachée du monde extérieur. Il suffit au programmeur de savoir utiliser son interface.

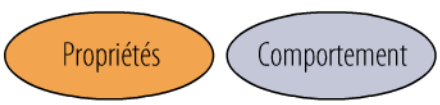
- Son code propre en est d'autant plus clair et plus stable.

Que signifie programmation orientée objet ?

La programmation orientée objet (ou POO) est du chinois pour la plupart des débutants, et peut même réserver maux de tête et nuits blanches à certains. Mais ce n'est pas obligé et j'espère pouvoir y contribuer, je veux dire à votre compréhension et non à vos maux de tête ! Dans le langage de programmation C++, tout est considéré comme objet et ce style – ou paradigme – de programmation s'oriente vers la réalité qui nous entoure. Nous sommes cernés d'innombrables objets qui sont plus ou moins réels et que nous pouvons toucher et observer. Si vous regardez un objet banal de plus près, vous pourrez constater certaines propriétés. Prenons par exemple un dé pour ne pas sortir du sujet. Vous savez déjà comment programmer et construire un dé électronique. Vous avez sûrement, dans l'un de vos jeux de société, un dé quelconque que vous pouvez regarder de plus près. Que pourriez-vous en dire, si vous deviez le décrire le plus précisément possible à un extra-terrestre ?

- À quoi ressemble-t-il ?
- Quelle taille a-t-il ?
- Est-il léger ou lourd ?
- De quelle couleur est-il ?
- A-t-il des points ou des symboles ?
- Quel nombre ou symbole est sorti ?
- Que peut-on faire avec ? (question idiote, non ?)

Les éléments de cette liste peuvent être répartis en deux catégories.



Mais quel élément fait partie de quelle catégorie ? Jetons un coup d'œil au tableau 9-1 puisqu'il s'agit de courant et de tension.

Propriétés	Comportement
Taille	Lancer le dé
Poids	
Couleurs	
Points ou symboles	
Nombre ou symbole sorti	

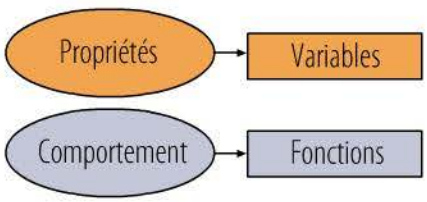
Tableau 9-1
Distinction entre propriétés et comportement

Seuls deux éléments de la liste sont pertinents pour la programmation prévue. Les autres sont certes intéressants, mais sans objet pour un dé électronique. Ces deux éléments sont :

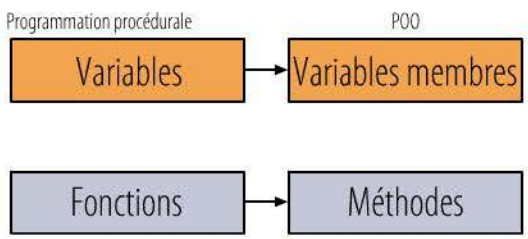
- le nombre de points sorti (état) ;
- lancer le dé (action).

Je n'ai pas la moindre idée de la manière dont on charge des propriétés ou un comportement dans un sketch. Comment fait-on ?

Ça ne pose aucun problème Ardus ! Voyez plutôt sur la figure suivante.



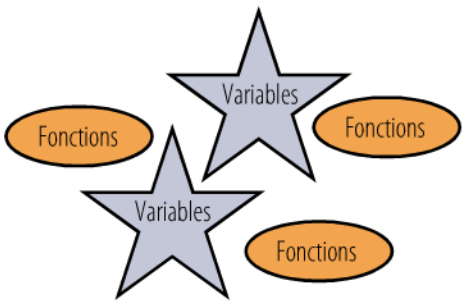
Les propriétés sont consignées dans des variables et le comportement est géré par des fonctions. Mais dans le contexte de la programmation orientée objet, variables et fonctions ont une autre désignation. Pas de quoi paniquer cependant puisque c'est en définitive la même chose.



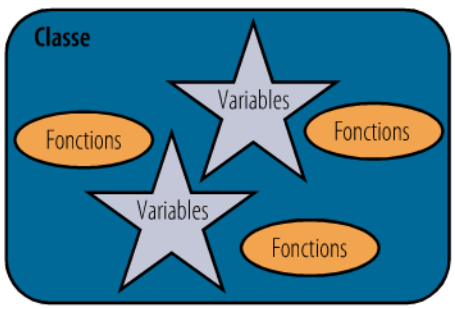
Les variables deviennent des variables membres (en anglais, *fields*) et les fonctions des méthodes (en anglais, *methods*).

Quelle avancée formidable ! Il suffit de rebaptiser deux éléments d'un programme pour avoir un nouveau – comme vous dites – paradigme de programmation. Le progrès tient à peu de choses, non ?

Allons Ardus, ne soyez pas sarcastique. C'est que je n'ai pas fini. Dans la programmation procédurale que nous connaissons à travers les langages C ou Pascal, des instructions ayant un rapport logique, qui sont nécessaires pour résoudre un problème, sont rassemblées dans ce qu'on appelle des procédures semblables à nos fonctions. Les fonctions opèrent en principe au mieux avec les variables qui leur ont été transmises comme arguments ou, dans le cas défavorable, avec des variables globales qui ont été déclarées au début d'un programme. Celles-ci sont visibles dans tout le programme et chacun peut les modifier à sa convenance. Toutefois, cela comporte certains risques et c'est actuellement, tout bien pesé, la plus mauvaise variante pour traiter des variables ou des données. Variables et fonctions ne forment aucune unité logique et vivent quasiment les unes à côté des autres dans le code, sans avoir aucun rapport direct entre elles.



Venons-en maintenant à la programmation orientée objet. Elle comporte une structure appelée classe. On peut dire pour simplifier que les servent de containers pour des variables membres ou des méthodes.



La classe enveloppe ses membres, appelés *members* dans la POO, à la manière d'un grand manteau. On ne peut en principe accéder aux membres qu'en passant par la classe.

Construction d'une classe

Mais qu'est-ce donc qu'une classe ? Si vous n'avez jamais eu affaire aux langages de programmation C++, Java et même C# pour ne citer que ceux-là, le terme ne vous en dira pas plus qu'un caractère chinois pour moi. Mais la chose est en fait assez facile à comprendre. Si vous regardez encore une fois le dernier graphique, vous verrez qu'une classe a vocation d'entourer et ressemble en quelque sorte à un container. Une classe est définie par le mot-clé `class`, suivi du nom qu'on lui a donné. Suit une paire d'accolades, que vous avez pu voir dans d'autres structures comme une boucle `for` et qui amène la formation d'un bloc. L'accolade finale est suivie d'un point-virgule.

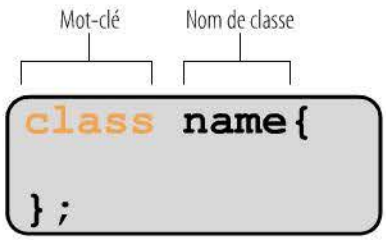


Figure 9-3
Définition générale d'une classe

Comme je vous l'ai déjà dit, la classe est composée de différents membres sous forme de variables membres et de méthodes, qui se fondent, selon la définition de cette classe, en une unité. La POO offre diverses possibilités de réglementer l'accès aux membres.

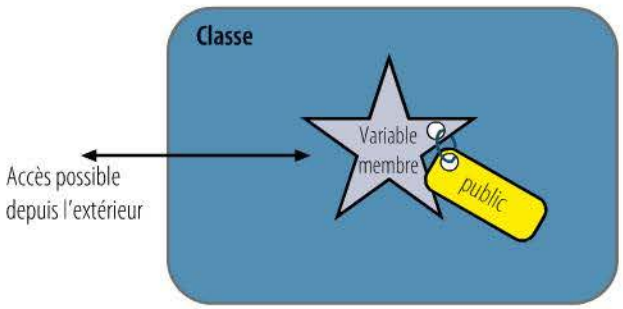
Oui, mais à quoi bon cette réglementation ? Quand je définis une variable ou plutôt une variable membre dans une classe, je veux pouvoir y accéder n'importe quand. À quoi sert cela sert-il si je ne peux plus ensuite accéder à la classe ? Ou peut-être ai-je mal compris le principe ?

Vous avez bien compris le principe, appelé d'ailleurs encapsulation. On peut protéger certains membres contre le monde extérieur, de telle sorte qu'ils ne soient pas directement accessibles depuis l'extérieur de la classe. Le mot directement est ici important. Il existe bien sûr des possibilités d'y accéder. Ce sont les méthodes qui, par exemple, s'en chargent. Mais vous devez vous demander quel est le sens de tout cela.

Exact ! On peut donc toujours influer directement sur les variables membres, n'est-ce pas ?

Bon ! Je pense que les figures suivantes vous permettront de mieux comprendre le principe.

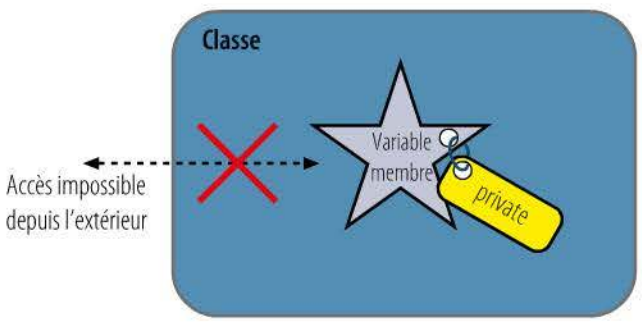
Figure 9-4 ▶
Accès à une variable membre de la classe



L'accès à la variable membre de la classe depuis l'extérieur est ici autorisé, car elle a reçu une certaine étiquette appelée modificateur d'accès. Elle a pour nom ici `public` et signifie à peu près ceci : l'accès est autorisé au public et tout un chacun peut s'en servir à sa guise.

Imaginez maintenant le scénario suivant : une variable membre doit piloter un moteur pas-à-pas, la valeur indiquant l'angle. Seuls des angles compris entre 0° et 359° sont cependant admis. Toute valeur inférieure ou supérieure peut compromettre l'exécution du sketch, si bien que le servo n'est plus commandé correctement. Quand vous donnez libre accès à une variable membre au moyen du modificateur `public`, aucune validation ne peut avoir lieu. Ce qui a été enregistré une fois produit immanquablement une réaction qui n'est pas forcément correcte. La solution du problème consiste à isoler les variables membres grâce à un modificateur d'accès `private` (privé). C'est le principe de l'encapsulation déjà évoqué qui est utilisé ici.

Figure 9-5 ▶
Pas d'accès à une variable membre de la classe



C'est bien beau tout ça !
Mais comment fait-on pour accéder à la variable membre ?

On y accède avec une méthode qui contient également un modificateur d'accès. Il doit cependant être `public` pour que l'accès fonctionne depuis l'extérieur. Le tout se présente comme sur la figure 9-6.

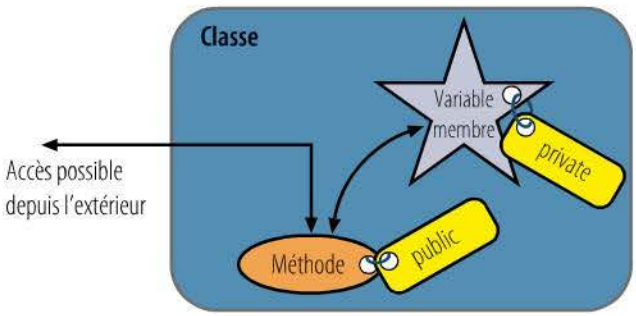


Figure 9-6
Accès à une variable membre de la classe par la méthode

On voit clairement que l'accès à la variable membre passe par la méthode, ceci étant un avantage et non pas un inconvénient. Vous pouvez maintenant procéder à la validation dans la méthode, seules des valeurs admises étant alors communiquées à la variable membre.

Mais pourquoi la méthode a-t-elle accès à la variable membre privée ? Je croyais que c'était impossible.

Le modificateur d'accès `private` signifie que l'accès depuis l'extérieur de la classe est impossible. Mais des membres de la classe comme les méthodes peuvent accéder à des membres déclarés `private`. Ils appartiennent tous à la classe et sont donc librement accessibles au sein de celle-ci. Pour faire court, les modificateurs d'accès gèrent l'accès aux membres de la classe.

Tableau 9-2
Modificateurs d'accès et leur signification

Modificateur d'accès	Description
`public`	L'accès aux variables membres et aux méthodes est possible depuis n'importe où dans le sketch. De tels membres constituent une interface publique de la classe.
`private`	L'accès aux variables membres et aux méthodes est réservé aux membres de la même classe.

Si vous voulez ajouter une classe à votre projet Arduino, mieux vaut créer un nouveau fichier se terminant par `.cpp` pour y stocker la définition de la classe. Vous verrez bientôt comment dans l'exemple concret de la bibliothèque-dé. Encore un peu de patience.

Une classe a besoin d'aide

Nous avons vu ce qu'une classe réalise et comment la créer en bonne et due forme. Mais je ne vous ai pas encore dit que la classe avait besoin d'un autre fichier très important. Celui-ci est appelé fichier d'en-tête et contient les déclarations (informations initiales ou préalables) pour la classe à concevoir. Si vous créez des variables membres

ou des méthodes en C++, vous devez impérativement les faire connaître auprès du compilateur avant de les utiliser. C'est chose faite en définissant les variables et les prototypes de fonction ou de méthode. Le fichier en question renferme également les consignes relatives aux modificateurs d'accès `public` et `private`.

La construction formelle du fichier d'en-tête ressemble à celle de la définition de classe, à ceci près qu'il ne contient pas de formulation de code. Autrement dit, seules les signatures des méthodes sont mentionnées. Une signature se compose uniquement des informations initiales avec le nom de la méthode, le type d'objet renvoyé et la liste des paramètres. La construction générale est la suivante :

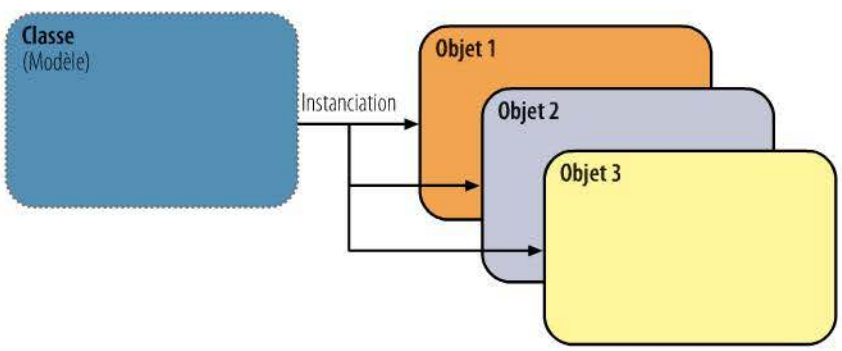
```
class Nom{
public:
   //Membre public
private:
  //Membre privé
};
```

La zone définissant le membre public vient après le mot-clé `public` suivi d'un deux-points. La zone définissant le membre privé vient après le mot-clé `private`, qui est suivi lui aussi d'un deux-points. Le fichier d'en-tête reçoit l'extension de nom `.h`.

Une classe devient un objet

Une fois créée par sa définition, une classe peut servir, comme lors de la déclaration d'une variable, de nouveau type de donnée. Ce procédé est appelé instanciation. Du point de vue du logiciel, la définition d'une classe ne veut pas dire qu'on a créé réellement un objet. Elle n'est qu'une sorte de modèle ou plan de construction qu'on peut utiliser pour concevoir un ou plusieurs objets.

Figure 9-7 ▶
De la classe à l'objet



L'instanciation se fait de la manière suivante :

```
Nomclasse Nomobjet();
```

Holà, il y a quelque chose qui ne va pas. Vous avez dit que l'instanciation d'un objet avait tout de la déclaration de variable ordinaire. Mais je vois encore une paire de parenthèses derrière le nom que vous avez donné à l'objet. Est-ce une double faute de frappe ? Sûrement pas. Qu'est-ce que c'est alors ?

Bien vu, Ardus ! Elle a naturellement son utilité. Une partie de ce projet va lui être consacrée car elle est extrêmement importante pour l'instanciation.

Initialiser un objet : qu'est-ce qu'un constructeur ?

Une définition de classe contient en principe quelques variables membres qui serviront après l'instanciation. Pour qu'un objet puisse présenter un état initial bien défini, il s'avère judicieux de l'initialiser en temps voulu. Quel meilleur moment pour cette initialisation que directement lors de l'instanciation ? Aucun risque ainsi qu'elle soit oubliée et pose plus tard problème lors de l'exécution du sketch. Mais comment faire pour initialiser un objet ? Le mieux est d'employer une méthode qui prend cette tâche en charge.

Il faut donc indiquer lors de l'instanciation une méthode à laquelle on donne certaines valeurs comme arguments. Mais comment savoir quelle méthode prendre ?

Exact, Ardus ! Il faut appeler une méthode et lui donner le cas échéant quelques valeurs en passant. Mais quel nom lui donner ? La solution est à la fois très simple et géniale. La méthode pour initialiser un objet porte le même nom que la classe. Cette méthode étant très spéciale, elle porte aussi un nom à elle. On l'appelle constructeur. Comme son nom l'indique, elle construit en quelque sorte l'objet. Mais puisqu'il n'est pas impérativement nécessaire d'initialiser dès le début un objet avec certaines valeurs, elle n'a pas forcément de liste de paramètres. Elle se comporte alors comme une méthode à laquelle aucun paramètre n'est donné et qui n'a que la paire de parenthèses vide. Ceci répond à votre question concernant la paire de parenthèses que vous avez vue dans l'instanciation.

Vous ne devez en aucun cas l'omettre ou l'oublier. Il me faut maintenant être un peu plus concret pour vous en montrer la syntaxe. Voici le contenu du fichier d'en-tête de notre bibliothèque-dé :

```
class Dice{
public:
    Dice();  //Constructeur
    // …
private:
    // …
};
```

Sous le modificateur d'accès `public` se trouve le constructeur qui porte le même nom que la classe. Il présente une paire de parenthèses vide, d'où son nom de *constructeur standard*.

Ne venez-vous pas de dire qu'on peut donner des arguments à un constructeur tout comme à une méthode pour initialiser l'objet ? La paire de parenthèses vide indique pourtant que le constructeur ne peut recevoir aucune valeur. Comment est-ce possible ? La deuxième chose que j'ai remarquée concerne le type présumé d'objet retourné par une méthode. Vous ne l'avez pas indiqué pour le constructeur. Pourquoi ?

Vous avez tout à fait raison Ardus quand vous dites que le constructeur ne peut accueillir aucune valeur sous cette forme. C'est une bonne introduction au prochain thème. Mais je vais d'abord répondre à votre question sur le type d'objet renvoyé manquant. Si une méthode renvoie une valeur à son appelant, le type de donnée en question doit naturellement être indiqué. Si aucun renvoi n'est prévu, le mot-clé `void` est utilisé. Revenons maintenant à notre constructeur. Il est appelé, non pas explicitement par une ligne d'instruction, mais implicitement par l'instanciation d'un objet. C'est pour cette raison que rien ne peut être retourné à un appelant et que le constructeur n'a pas même le type de renvoi `void`.

La surcharge

Ce que je vais vous dire là peut sembler déroutant à première vue : on peut définir un constructeur et bien entendu également des méthodes plusieurs fois avec le même nom.

Je vous avoue que j'ai du mal à le croire. C'est pourtant contraire au principe de clarté. Si par exemple une méthode apparaît deux fois avec le même nom dans un sketch, comment le compilateur peut-il savoir laquelle des deux est appelée ?

C'est vrai Ardus. Mais il n'y a pas que le nom qui soit déterminant, il y a aussi la fameuse *signature* dont je vous ai parlé plus haut. L'exemple suivant montre deux constructeurs acceptables qui portent le même nom mais dont les signatures diffèrent :

```
Dice();
Dice(int, int, int, int);
```

Le premier constructeur représente le constructeur standard et sa paire de parenthèses vide, qui ne peut accueillir aucun argument. Le deuxième porte une toute autre signature car il peut recevoir quatre valeurs du type `int`. Vous pouvez alors choisir entre deux variantes pour instancier un objet `Dice` :

```
Dice myDice();
```

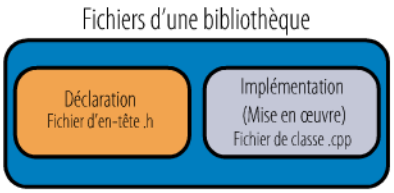
ou :

```
Dice myDice(8, 9, 10, 11);
```

Le compilateur est assez intelligent pour savoir quel constructeur il doit appeler.

La bibliothèque-dé

Toute cette introduction était nécessaire pour bien vous faire comprendre la création d'une bibliothèque Arduino. Le deuxième projet de dé servira de base pour constituer une bibliothèque. Il s'agit d'une variante améliorée avec commande des groupes de LED. Deux fichiers sont donc nécessaires pour réaliser la bibliothèque.



Le fichier d'en-tête

Commençons par le fichier d'en-tête, qui ne contient que les informations de prototype et ne présente aucune information de code explicite. Occupons-nous d'abord des membres de la classe qui sont nécessaires. Pour piloter les groupes de LED, il faut quatre broches numériques commandées par les variables membres :

- pinGroupA ;
- pinGroupB ;
- pinGroupC ;
- pinGroupD.

Ces informations seront transmises au moment de l'instanciation au constructeur, qui possède quatre paramètres du type `int`. Les variables membres sont déclarées privées (`private`) car elles ne sont traitées qu'en interne par une méthode appelée `roll`, qui n'a aucun argument et qui ne retourne rien. La classe reçoit le nom évocateur de `Dice` (dé).

```
#ifndef Dice_h
#define Dice_h

#if ARDUINO < 100
#include <WProgram.h>
#else
#include <Arduino.h>
#endif

class Dice{
public:
    Dice(int, int, int, int);  //Constructeur
    void roll();    //Méthode pour lancer le dé
private:
    int GroupA;   //Variable membre pour groupe de LED A
    int GroupB;   //Variable membre pour groupe de LED B
    int GroupC;   //Variable membre pour groupe de LED C
    int GroupD;   //Variable membre pour groupe de LED D
};
#endif
```

Quelques informations supplémentaires méritant une explication ont été ajoutées à la définition de la classe. La classe tout entière a été enveloppée dans la structure suivante :

```
#ifndef Dice_h
#define Dice_h
        ...
#endif
```

Des inclusions multiples étant possibles quand il y a du code imbriqué, un moyen a été trouvé pour les empêcher et éviter une double compilation. Cette précaution a pour but de garantir une inclusion unique du fichier d'en-tête. Les instructions `#ifndef`, `#define` et `#endif` sont des instructions de prétraitement. `#ifndef`, qui introduit une

compilation conditionnelle, est la forme abrégée de *if not defined* qui signifie « si non défini ». Si le terme `Dice_h` (nom du fichier d'en-tête avec un tiret bas) – appelé macro – n'a pas encore été défini, faites-le maintenant et exécutez les instructions dans le fichier d'en-tête. Si ce dernier était appelé une deuxième fois, la macro serait placée sous le nom et cette partie de la compilation serait rejetée. Les instructions `include` sont nécessaires pour faire connaître à la bibliothèque les types de données ou constantes propres à Arduino (par exemple : `HIGH`, `LOW`, `INPUT` ou `OUTPUT`).

```
#if ARDUINO < 100
#include <Wprogram.h>
#else
#include <Arduino.h>
#endif
```

Il y a ici un truc : toutes les versions Arduino antérieures à la version 1.0 nécessitent un fichier d'en-tête nommé `Wprogram.h` pour utiliser par exemple lesdites constantes. Il sert à bien autre chose encore, mais restons-en là pour le moment. Le numéro de version d'Arduino figure dans la définition du terme ARDUINO et peut donc être lu pour l'environnement de développement actuel. C'est ce que nous faisons dans notre cas. Si le numéro de version est <100 (soit la version 1.00), l'ancien fichier d'en-tête `Wprogram.h` doit être intégré. Sinon, le nouveau fichier d'en-tête `Arduino.h` sera utilisé. Cette modification des fichiers d'en-têtes fait souvent râler et je dois dire que cette adaptation ne m'enchante pas non plus.

Pouvez-vous me dire pourquoi le constructeur n'indique que le type de donnée pour les paramètres et pourquoi le nom de la variable correspondante est absent ?

Cela tient au fait que nous n'avons besoin ici que des informations de prototype. Le code en question apparaît plus tard avec une extension `.cpp` dans le fichier de classe.

Le fichier de classe

La véritable mise en œuvre du code est effectuée au moyen du fichier de classe présentant l'extension `.cpp` :

```
#if ARDUINO < 100
#include <WProgram.h>
#else
#include <Arduino.h>
#endif
```

```
#include "Dice.h"
#define WAITTIME 20

//Constructeur paramétré
Dice::Dice(int A, int B, int C, int D){
   GroupA = A;
   GroupB = B;
   GroupC = C;
   GroupD = D;
   pinMode(GroupA, OUTPUT);
   pinMode(GroupB, OUTPUT);
   pinMode(GroupC, OUTPUT);
   pinMode(GroupD, OUTPUT);
}
//Méthode pour lancer le dé
void Dice::roll(){
   int number = random(1, 7);
   digitalWrite(GroupA, number%2!= 0?HIGH:LOW);
   digitalWrite(GroupB, number>1?HIGH:LOW);
   digitalWrite(GroueC, number>3?HIGH:LOW);
   digitalWrite(GroupD, number==6?HIGH:LOW);
   delay(WAITTIME);  //Ajouter une courte pause
}
```

Pour que la liaison vers le fichier d'en-tête précédemment créé soit possible, référence est faite à ce dernier au moyen de l'instruction `include` :

```
#include "Dice.h"
```

Son intégration intervient lors de la compilation. `include` est ici également nécessaire pour pouvoir utiliser ce qu'on appelle les éléments de langage Arduino.

```
#if ARDUINO < 100
#include <Wprogram.h>
#else
#include <Arduino.h>
#endif
```

Passons maintenant au code, qui contient la mise en œuvre proprement dite. Commençons par le constructeur :

```
Dice::Dice(int A, int B, int C, int D){
   GroupA = A;
   GroupB = B;
   GroupC = C;
   GroupD = D;
```

```
    pinMode(GroupA, OUTPUT);
    pinMode(GroupB, OUTPUT);
    pinMode(GroupC, OUTPUT);
    pinMode(GroupD, OUTPUT);
}
```

Vous avez sûrement remarqué que la méthode `roll` était légèrement différente de celle du montage n° 8.

> Oui, aucune LED n'a été éteinte avant de commander l'allumage des nouvelles. Je ne vois pas bien pourquoi !

C'est vrai, Ardus ! Et ce n'est pas grave puisque les différents groupes de LED sont commandés par l'opérateur conditionnel `?` que vous avez déjà rencontré dans le montage n° 7 sur la machine à états. Cet opérateur retourne soit `LOW` soit `HIGH` quand la condition a été évaluée, si bien que le groupe de LED correspondant a toujours le bon niveau et n'a pas besoin d'être remis auparavant sur `LOW`. Une autre chose susceptible de vous étonner est le préfixe `Dice::` qui précède aussi bien le nom de la structure que la méthode `roll`.

Il s'agit du nom de la classe, qui permet au compilateur de savoir à quelle classe la définition de la méthode appartient. Cette dernière est qualifiée par cette notation. L'objet dé que nous voulons créer devant commander quatre groupes de LED, il est préférable de transmettre ces informations au moment de l'instanciation. Ce serait bien entendu possible aussi après la génération de l'objet, en utilisant une méthode distincte que nous appellerions par exemple `Init`. Mais le risque est grand que cette étape soit oubliée. C'est pourquoi le constructeur a été inventé. Voyons maintenant le sketch qui utilise cette bibliothèque.

Création des fichiers nécessaires

Je vous propose de programmer les deux fichiers de la bibliothèque `.h` et `.cpp` indépendamment de l'environnement de développement Arduino. Il existe pour ce faire de nombreux éditeurs, par exemple Notepad++ ou Programmers Notepad. Les deux fichiers sont stockés dans un répertoire au nom évocateur, par exemple `Dice` (dé), qui est copié une fois prêt dans le dossier des bibliothèques Arduino.

```
…\arduino-1.x.y\libraries
```

L'environnement de développement Arduino est ensuite redémarré et la programmation du sketch peut commencer.

Mise en surbrillance de la syntaxe pour une nouvelle bibliothèque

Des types de données élémentaires (par exemple : `int`, `float` ou `char`) ou d'autres mots-clés (par exemple : `setup` ou `loop`) sont signalés en couleurs par l'environnement de développement. Il est possible, lors de la création de ses propres bibliothèques, de faire connaître à l'IDE des noms de classes ou de méthodes, pour qu'ils apparaissent alors aussi en couleurs. Un fichier nommé `keywords.txt`, ayant obligatoirement une syntaxe spéciale, doit être créé pour que cela fonctionne.

Commentaires

Des commentaires explicatifs sont introduits par le signe `#` (dièse) :

```
# Ceci est un commentaire
```

Types de données et classes (KEYWORD1)

Les types de données mais aussi les noms de classes figurent en orange et doivent être définis en respectant la syntaxe suivante :

```
Nomclasse KEYWORD1
```

Méthodes et fonction (KEYWORD2)

Méthodes et fonctions apparaissent en marron et doivent être définies en respectant la syntaxe suivante :

```
Méthode KEYWORD2
```

Constantes (LITERAL1)

Les constantes sont en bleu et doivent être définies en respectant la syntaxe suivante :

```
Constante LITERAL1
```

Voici maintenant le contenu du fichier `keywords.txt` de notre bibliothèque-dé :

```
#-------------------------------------------------------
# Affectation des couleurs pour la bibliothèque Dice
#-------------------------------------------------------

#-------------------------------------------------------
# KEYWORD1 pour types de données ou classes
#-------------------------------------------------------
```

```
Dice KEYWORD1

#-------------------------------------------------------
# KEYWORD2 pour méthodes et fonctions
#-------------------------------------------------------

roll KEYWORD2

#-------------------------------------------------------
# LITERAL1 pour constantes
#-------------------------------------------------------
```

Utilisation de la bibliothèque

Le fait que la bibliothèque-dé se trouve dans le répertoire indiqué précédemment vous permet de la retrouver dans la dernière entrée du menu (voir figure 9-8).

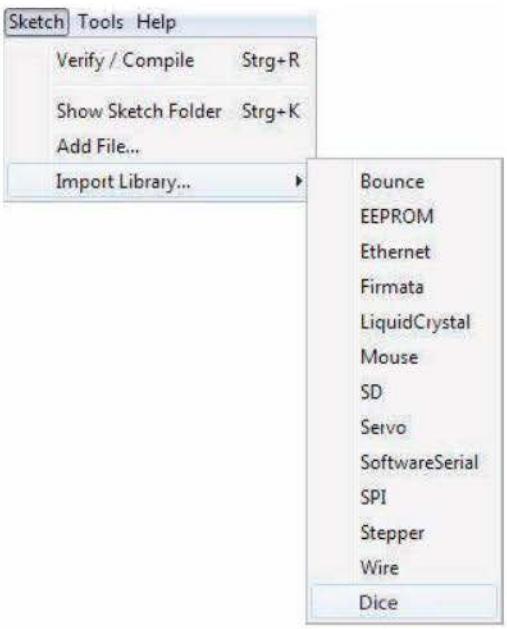


Figure 9-8
Importer la bibliothèque-dé

Le terme importer est quelque peu mal venu puisque absolument rien n'est importé à ce moment précis. Seule la ligne suivante est ajoutée dans votre fenêtre de sketch :

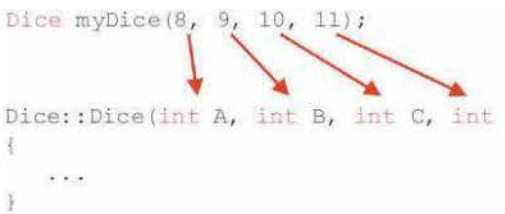
```
#include <Dice.h>
```

La ligne `include` est impérative pour pouvoir accéder à la fonctionnalité de la bibliothèque-dé. Comment le compilateur saurait-il sinon à

quelle bibliothèque il doit accéder ? Les différentes bibliothèques disponibles ne sont pas incluses par l'opération du Saint-Esprit ! Passons maintenant à l'instanciation, qui élève la définition de classe au statut d'objet réel. L'objet créé `myDice` est également appelé variable d'instance. Ce terme revient souvent dans la littérature.

```
Dice myDice(8, 9, 10, 11);
```

Les valeurs transmises `8`, `9`, `10` et `11` figurent les broches numériques auxquelles les groupes de LED sont reliés. Un objet dé a ainsi été initialisé de manière à pouvoir opérer en interne quand la méthode pour lancer le dé est appelée. Les arguments sont transmis dans l'ordre indiqué.



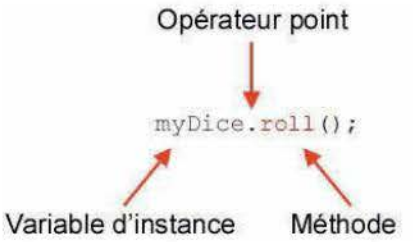
Ils sont stockés dans les variables locales `A`, `B`, `C` et `D`, qui sont à leur tour transmises aux variables membres `GroupA`, `GroupB`, `GroupC` et `GroupD`. Vient ensuite l'appel de la méthode à condition qu'un potentiel `HIGH` soit appliqué à l'entrée numérique, appel qui peut se faire par le bouton-poussoir raccordé.

```
void setup(){
   pinMode(13, INPUT); //Pas obligatoire - oui mais pourquoi ?
}
void loop(){
   if(digitalRead(13) == HIGH)
               myDice.roll( );
}
```

On voit ici aussi que la mise en surbrillance de la syntaxe fonctionne puisque le nom de classe et la méthode sont en couleurs. La méthode `roll` étant un membre de la définition de classe `Dice`, une liaison vers la classe doit être établie en cas d'appel de celle-ci. Un appel par :

```
roll();
```

provoquerait ici une erreur. La relation est établie par l'*opérateur point* inséré entre classe et méthode et faisant office de lien.



Vous serez amené plus tard à programmer l'une ou l'autre bibliothèque qui pourra vous être utile à vous ou à d'autres programmeurs. Vous acquerrez alors un peu de pratique dans la manipulation du langage C++ et la programmation orientée objet.

Pour aller plus loin

Pour compléter ce chapitre, vous pouvez effectuer une recherche sur Internet sur les mots-clés :

- programmation orientée objet ;
- POO ;
- Arduino Library.

Qu'avez-vous appris ?

- Je reconnais que les choses sont devenues un peu plus difficiles dans ce projet, mais le jeu en vaut la chandelle. Vous en savez maintenant plus sur le paradigme de programmation orientée objet.
- Vous savez bien faire la différence entre une classe et un objet.
- Dans la POO, méthode remplace fonction et variable membre remplace variable.
- Le constructeur est une méthode avec une tâche particulière. Il initialise l'objet de manière à obtenir un état de départ bien défini.
- Vous connaissez les informations de code qu'un fichier d'en-tête ou `.cpp` doit contenir. Les différents modificateurs d'accès `public` ou `private` réglementent l'accès aux membres de l'objet, `private` assurant l'encapsulation des membres.
- L'objet instancié à partir d'une classe est également appelé variable d'instance.
- Un opérateur point, ajouté après le nom de la variable d'instance et servant quasiment de lien entre les deux, est utilisé pour accéder aux variables membres et aux méthodes.

- Vous savez comment créer une bibliothèque Arduino et à quel endroit copier celle-ci dans le système de fichiers pour pouvoir y accéder à tout moment.
- Vous avez enfin appris à configurer certaines méthodes en tant que mots-clés avec un marquage couleur.