Introduction à la Programmation Objet

Jusqu'à présent, vous avez appris à écrire
des programmes de **plus en plus complexes**.

Il faut donc maintenant des outils
pour ***organiser*** ces programmes de **façon plus *efficace***.

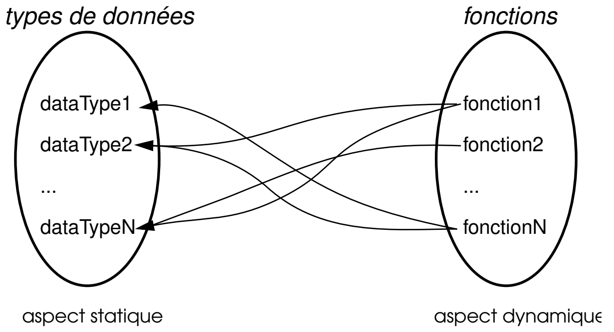
C'est l'un des objectifs principaux de la notion d'***objet***.

Les objets permettent de mettre en œuvre
dans les programme les notions:

- **d'*encapsulation* et d'*abstraction*;**
- **d'*héritage* et de *polymorphisme*.**

Encapsulation (1)

Dans les programmes tels qu'on les a écrits (et décrits) jusqu'à présent,
les notions de ***type de données*** et de ***fonctions*** sont **séparées**:



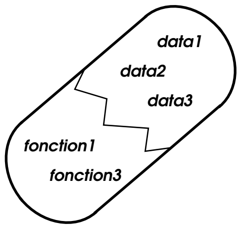
Où les «dataType$_i$» sont des types de données (`int`, `char`, mais aussi ceux définis par `typedef`), et les «fonctions$_j$» sont des fonctions utilisant ces types, par exemple:

```
dataType2 fonction2(const dataType1 x, dataType3 y)
```


Encapsulation (2)

Le **principe d'*encapsulation*** consiste à **regrouper**,
dans un même élément informatique, les aspects *statiques* et *dynamiques*
(i.e. les données et les fonctions) spécifiques à une entité.

Cet élément informatique est appelé: «**objet**»



- les [structures de] données définies dans un objet sont appelées **les *attributs*** de l'objet;
- les fonctions [de manipulations] définies dans un objet sont appelées **les *méthodes*** de l'objet.

On a donc la relation fondamentale:

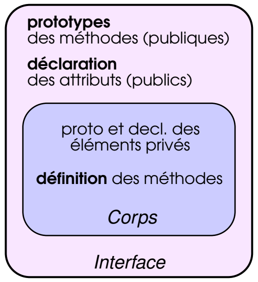
OBJET = attributs + méthodes

Encapsulation (3)

En plus du regroupement des éléments statiques et dynamiques [d'une entité],
l'*encapsulation* permet de définir **deux *niveaux de perception***:

- le **niveau *externe***: perception de l'objet depuis **l'extérieur**
- le **niveau *interne***: perception de l'objet depuis **l'intérieur**

Le **niveau *externe***, correspond à la partie **visible** de l'objet;
il est constitué des spécifications des élements [de l'objet] visibles de l'extérieur (appelés «***éléments publics***»[1]), à savoir les prototypes et les déclarations de ses méthodes et attributs *publics*.
Ce niveau représente donc ***l'interface* de l'objet avec l'extérieur**.



Le **niveau *interne*** correspond à l'**implémentation** de l'objet;
il est constitué des éléments [de l'objet] visibles uniquement de l'intérieur de cet objet (appelés «***éléments privés***»); outre le prototypage et la déclaration des méthodes et attributs *privés*, le niveau interne regroupe également les définitions de l'ensemble des méthodes de l'objet.
Ce niveau représente donc **le *corps* de l'objet**.

1. Les concepts relatifs à la visibilité des éléments (publiques, privés, etc) seront traités en détails plus tard dans le cours.

Abstraction (1)

Pour être véritablement intéressante,
la notion d'objet doit permettre un certain **degré d'*abstraction***.

Si l'on considère par exemple les figures géométriques,
la notion d'objet «rectangle» ne sera intéressante que si l'on peut
lui associer des propriétés générales, vraies pour l'ensemble des rectangles,
et non pas uniquement pour un rectangle particulier.

Le processus d'abstraction consistera donc à *identifier*,
pour un ensemble donné d'éléments:

1) des caractérisations valides pour la totalité de ces éléments;
2) des mécanismes communs à la totalité de ces éléments.

Ces caractérisations et mécanismes constituent donc une
description générique des éléments de l'ensemble considéré.

Par exemple, les notions de «largeur» et «hauteur»
sont des propriétés communes à l'ensemble des rectangles,
c'est à dire au *concept* ***abstrait*** «rectangle», et il en va de même pour la relation
qui existe entre la «surface» d'un rectangle et ses dimensions (surface = largeur x hauteur)

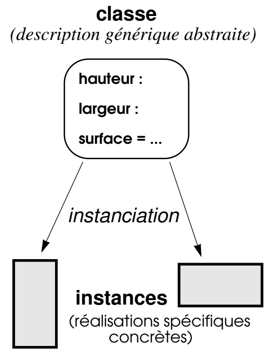
Abstraction (2)

En programmation objet,
nous appelerons ***classe*** le résultat du
processus d'abstraction précédemment décrit.

Une «***classe***» **factorise donc les caractérisations communes** des éléments qu'elle permet de décrire, et de ce fait peut aussi être vue comme un moule (patron) pour la création des objets (que l'on appelle alors des «***instances***» de la *classe*).

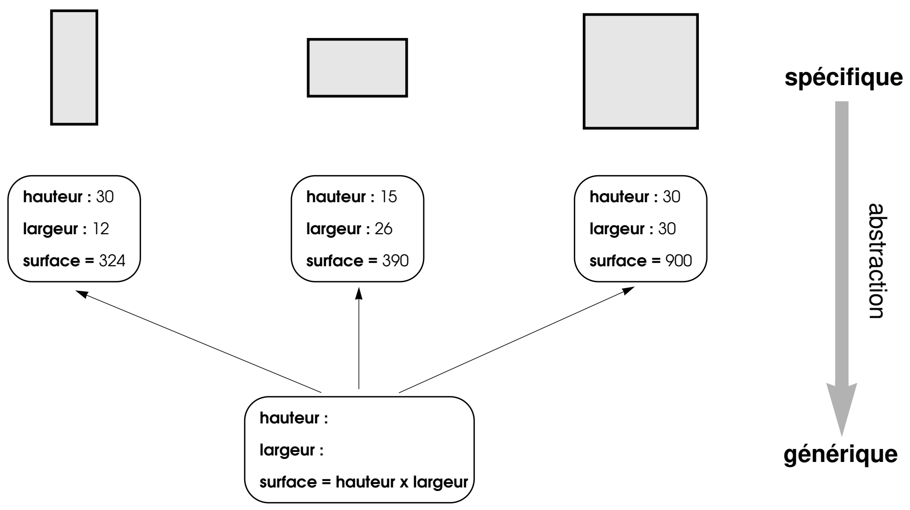
Une classe correspond à un type[2]
et la «création» d'une instance pour une classe donnée
se fait par le biais de la déclaration d'une variable
du type représenté par la classe.

Cette déclaration est aussi appelée ***instanciation***
et l'*instance* produite est donc une
réalisation particulière de la classe.



2. Un type complexe, car composite (plusieurs attributs) et pour lequel on définit de plus un ensemble de fonctions, les «méthodes».

Abstraction (3)



Exemples d'objets

Quelques exemples de représentation d'objets:

- le livre «Conception et programmation orientées objet» de la bibliothèque du DI ;
- le velo de votre petit frère ;
- votre petit frère ;
- le stylo que vous tenez dans la main ;
- ...
- la chaîne de caractères qui représente votre nom ;
- la fenêtre dans laquelle netscape s'affiche ;
- ...

Et leur **abstraction** en terme de classes:

- l'ensemble des livres ;
- l'ensemble des vélos ;
- l'ensemble des personnes ;
- l'ensemble des stylos ;
- ...
- le type «chaînes de caractères» (string) ;
- les fenêtres d'une interface graphique (CDE) ;
- ...

Classes et instances (1)

Il est important de bien saisir la différence
entre les notions de *classe* et d'*instance*:

- **classe** = ***attributs* + *méthodes* + *mécanisme d'instanciation***
  On désigne souvent les *attributs* et *méthodes* d'une classe comme ses ***propriétés***.
  L'*instanciation* est le mécanisme qui permet de créer des instances dont les traits sont ceux décrits par la classe;
  l'ensemble des instances d'une classe constitue l'***extension*** de la classe.

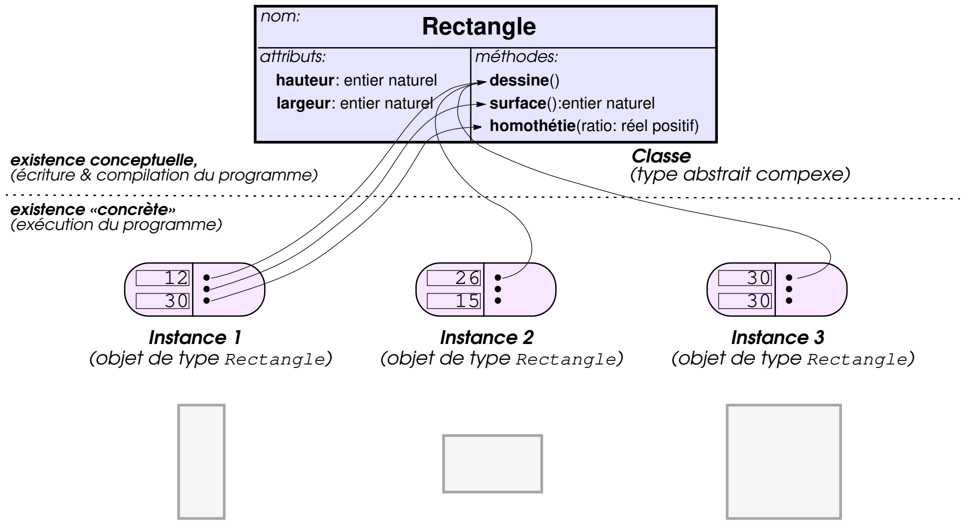
- **instance** = **valeurs d'*attributs* + accès aux *méthodes***

Dans le jargon de la programmation objet,
les termes *instance* et *objet* sont pratiquement synonymes[3],
d'où la reformulation de l'équation fondamentale de départ:

OBJET = [valeurs d']attributs + [accès aux] méthodes

3. La seule différence que l'on peut relever est qu'une *instance* désigne **toujours** une réalisation d'un type «classe», tandis que le terme *objet* est parfois utilisé dans son sens commun («Toute chose ayant **unité** et **indépendance** et qui affecte les sens»; «Ce qui est donné par l'expérience, et **existe indépendamment de l'esprit**») et désigne alors des réalisations de type quelconques.

Classes et instances (2)



Intérêt de l'encapsulation

L'intérêt de la séparation entre niveau interne et externe
est de donner un cadre plus rigoureux à l'utilisation des objets:

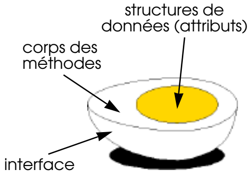
> Les objets ne peuvent être utilisés
> qu'au travers de leur interface (niveau externe)

de ce fait, les éventuelles modifications apportées à leur structure interne
restent invisibles à l'extérieur.[4]

Cette manière de procéder peut se traduire
par les deux recommandations suivantes:

> les variables globales (existant hors de tout objet)
> sont, dans la mesure du possible, à proscrire

> les attributs des objets ne devraient en général pas
> être directement accessibles depuis l'extérieur,
> mais uniquement au travers de méthodes[5]



4. Cette volonté de séparer la caractérisation des fonctionalités disponibles de leur définition effective est récurrente en informatique.
5. Sans quoi, ils peuvent être apparentés à (utilisés comme) des variables globales.

Syntaxe: forme générale

Une syntaxe possible pour le prototypage d'une classe est:

```
class <nom de classe>
{
  public:
  // déclaration des attributs
  ...
  // prototype/définition des méthodes
  ...
};
```

La classe est définie à l'aide d'un bloc et se termine par un point-virgule,
comme s'il s'agissait d'une instruction ordinaire.

Tout ce qui suit le mot-clef **`public`** constitue l'*interface de la classe*
et est donc visible et utilisable depuis l'extérieur.

l'ordre de déclaration des méthodes et des attributs est sans importance,
mais il est d'usage de les regrouper, et d'indiquer en premier les attributs publics.

Syntaxe: déclaration des attributs

La syntaxe de déclaration des attributs est
la syntaxe normale de déclaration des variables:[6]

```
<type> <identificateur d'attribut>;
```

Exemple:

Les attributs **hauteur** et **largeur**, de type entier, de la classe **Rectangle**, pourront être déclarés par:

```
unsigned int hauteur;
unsigned int largeur;
```

unsigned int est un type de base représentant les entiers (**int**) sans signe (**unsigned**), donc positifs.

Contrairement aux variables simples, les attributs (variables ou constants) ne peuvent être directement initialisés lors de leur déclaration. Pour réaliser ces initialisations, on définira des méthodes particulières, appelées ***constructeurs***, et dont le rôle est précisément d'attribuer une valeur initiale aux attributs. Ce point sera examiné plus en détail dans la suite du cours.

6. Il existe toutefois une différence, qui réside dans la possibilité de déclarer des variables «mutables» (c.f. suite du cours).

Syntaxe: définition des méthodes (1)

La syntaxe de définition des méthodes
est globalement la même que celle des fonctions:

```
<type> <identificateur>(<arguments>)
{
   <corps de la fonction>
   [return <valeur>]
}
```

Exemple: La méthode **dessine** de la classe **Rectangle**:

```
void dessine()
{
   // primitives graphiques
}
```


Syntaxe: définition des méthodes (2)

Les attributs comme les méthodes «appartiennent» aux objets;
en particulier, ils sont pleinement accessibles (et donc utilisables)
dans toutes les méthodes de l'objet
(on peut les considérer comme des variables globales de l'objet).

Il n'est **pas nécessaire** de passer les attributs
en arguments des méthodes.

Exemple: Les attributs `hauteur` et `largeur` des objets issus de la classe `Rectangle` sont «connus» par chacune des méthodes de cette classe;
la méthode `surface` peut donc être définie ainsi:

```
unsigned int surface()
{
   return (hauteur * largeur);
}
```


Syntaxe: masquage des attributs

Si, dans une méthode, un attribut est **masqué** (c'est-à-dire par exemple qu'il existe dans la méthode une variable de même identificateur que l'attribut, il peut tout de même être référencé, à l'aide du mot-clef **this**:

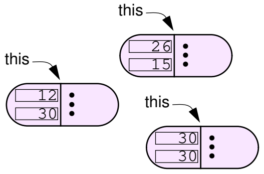
La syntaxe est alors la suivante:

this-><*nom de l'attribut*>

Exemple:

Une méthode permettant de modifier la hauteur d'un rectangle tout en assurant que la nouvelle valeur soit supérieure à zéro:

```
void changeHauteur(unsigned int hauteur)
{
   if (hauteur > 0)
   {
      this->hauteur = hauteur;
   }
}
```



this peut être interprété comme *argument caché* dans chaque méthode. Il fait toujours référence à **l'instance pour laquelle la méthode est invoquée** (i.e. l'objet dont on a invoqué une méthode).

Syntaxe: portée des attributs

La portée des attributs dans les définitions
des méthodes est résumée par le schéma suivant:

```
class C
{
   int x;
   int y;
...
   void uneMethode(int x)
   {
   ...
   ... y ...
   ... x ...
   ... this->x ...
   }
};
```

attributs

variables locales

```
class C
{
   int x;
   int y;
   ...
   void uneMethode(int x)
   {
   ...
   ... y ...
   ... x ...
   ... this->x ...
   }
};
```


Exemple: la classe `Rectangle`

```
class Rectangle
{
   public:

   // déclaration des attributs ...........
   unsigned int hauteur;
   unsigned int largeur;

   // prototypage et définition des méthodes ....................
   unsigned int surface()
   {
      return (hauteur * largeur);
   }

   void changeHauteur(const unsigned int hauteur)
   {
      if (hauteur > 0)
         this->hauteur = hauteur;
   }
};
```


Syntaxe: instanciation d'une classe

En programmation objet, les classes sont
considérées commes des types à part entière.

Instancier une classe revient donc simplement à déclarer
une variable ou une constante, dont le type est l'identificateur de la classe.

La syntaxe est donc:

$\underbrace{\textit{<identificateur de classe>}}_{\text{type}}\ \underbrace{\textit{<identificateur d'instance>}}_{\text{variable}}$ `;`

Exemples:

`Rectangle rect1;`
déclare une instance, nommée **`rect1`**, de la classe **`Rectangle`**.

`string maChaine;`
déclare une instance, nommé **`maChaine`**, de la classe **`string`**.

Accès aux contenu d'une instance (1)

L'accès aux attributs et méthodes d'une instance se fait en ajoutant,
après l'identificateur d'instance, un point («.») suivi
par l'identificateur de l'attribut ou de la méthode en question:

Accès aux attributs: *<id d'instance>*`.`*<id d'atribut>*

Exemple:

l'accès à l'attribut `hauteur` de l'instance `rect1` de la classe `Rectangle`
se fera au moyen d'une expression telle que «`rect1.hauteur`»

Par exemple: `rect1.hauteur = 18;`

Accès aux méthodes: *<id d'instance>*`.`*<id de méthode>*`(`*<arg_1>*`,` *<arg_2>*`, ...)`

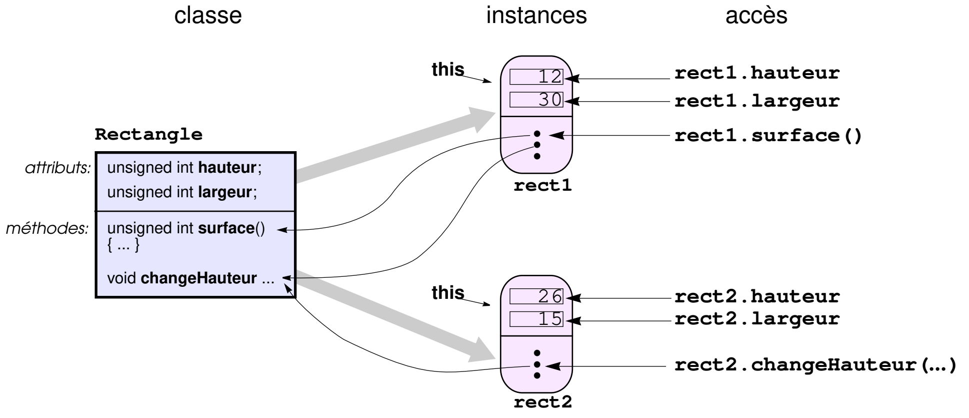
Exemple:

l'invocation (appel) de la méthode `surface` de l'instance `rect1` se fera au
moyen d'une expression telle que «`rect1.surface()`»

Par exemple: `cout << "Surface = " << rect1.surface();`

Accès aux propriétés d'une instance (2)

Les accès aux attributs et méthodes d'une instance
sont résumés dans le schéma suivant:



Utilisation de la classe Rectangle

```
#include <iostream>

// définition de la classe Rectangle comme précedemment.

void main()
{
   Rectangle rect;

   cout << "Entrez la hauteur: ";
   unsigned int h;
   cin >> h;
   rect.changeHauteur(h);

   cout << "Entrez la largeur: ";
   cin >> rect.largeur;

   cout << "Surface du rectangle = " << rect.surface() << endl;
}
```


Actions et prédicats

C++ permet de distinguer les méthodes qui modifient
l'état de l'instance (les ***actions***) de celles qui ne
le font pas (les ***prédicats***):

Cette distinction est indiquée dans le prototype de la méthode:

```
<type> <identificateur>(<arguments>) [const]
{ ... }
```

La clause optionnelle **const**, située après la liste des arguments
(et faisant partie de la signature) désigne une méthode de type «prédicat»[7],
pour laquelle le compilateur interdira toute modification d'attributs[8]:

Exemple:

```
unsigned int surface()
{
   hauteur = largeur = 0;
   return (hauteur * largeur);
}
```

```
unsigned int surface() const
{
   hauteur = largeur = 0;
   return (hauteur * largeur);
}
```

7. C'est-à-dire une méthode dont on peut être certain que l'invocation n'aura pas d'incidence sur l'instance utilisée..
8. A l'exception de ceux déclarés «mutables» (c.f. suite du cours)

Surcharge de méthodes et arg. par défaut

Comme pour les fonctions, il est possible, d'une part, de surcharger
les méthodes (plusieurs méthodes ayant un même identificateur) et, d'autre part,
de définir des méthodes admettant des arguments avec valeur par défaut.

Les règles à respecter sont les mêmes
que dans le cas de fonctions ordinaires:

- ⇨ dans le cas de la surcharge, les signatures des fonctions doivent différer; il faut donc que le nombre ou le type des arguments soient différents, ou, dans le cas des méthodes, le droit accordé ou non de modifier l'état de l'instance (présence/absence de «const»).
- ⇨ dans le cas des arguments avec valeurs par défaut, ces arguments doivent apparaître en dernier dans la liste des arguments de la méthodes.

Exemple:

La méthode `changeHauteur` pourrait être **redéfinie** avec une valeur par défaut:[9]

```
void changeHauteur(const unsigned int hauteur = 5) {...}
```

L'invocation «`changeHauteur()`» est alors équivalente à «`changeHauteur(5)`».

9. Mais il ne serait pas possible de surcharger la méthode précedemment définie; par contre, les deux versions de la méthode `surface` présentées pour illustrer les méthodes de type «prédicat» peuvent très bien cohexister au sein d'une même classe.