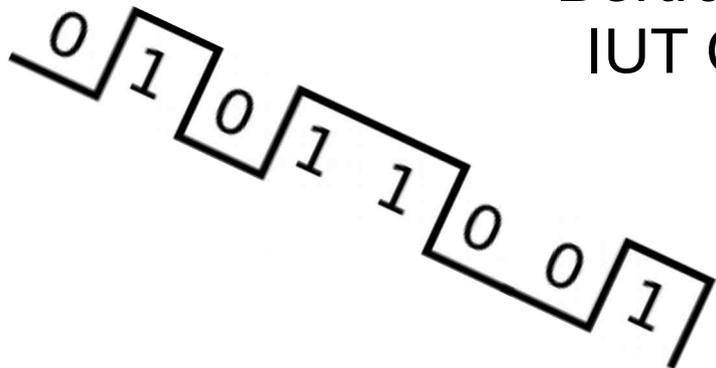


INITIATION A LA PROGRAMMATION ORIENTEE OBJET (POO)

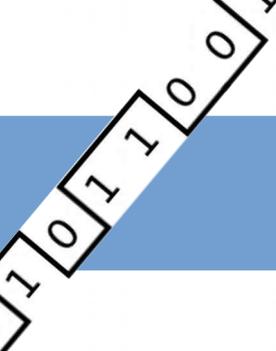
Bertrand Vandepoortaele
IUT GEII TOULOUSE
2019





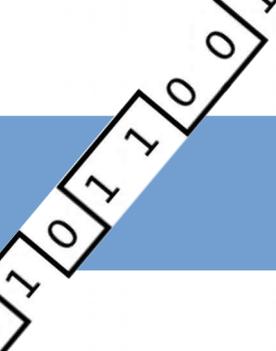
Objectifs

- Découvrir les **possibilités** offertes par la P.O.O.
- Comprendre les **concepts**
- Savoir **utiliser des objets déjà créés** (bibliothèques Arduino, QT...)
- Avoir un regard différent sur le développement d'une application



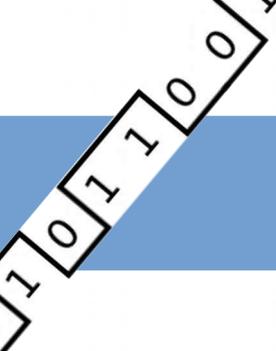
Déroulement

- 1 séances de cours (1x2h)
- 2 séances de TP (2x2h) : C++, QT Creator
- Manipulation en TP avec l'Arduino



Plan

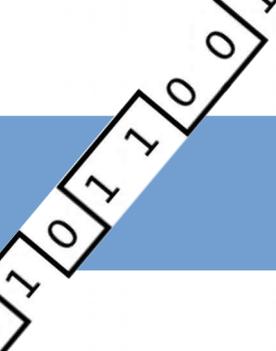
- Introduction
- Présentation des concepts
- QT
- Mise œuvre et comparaison



Introduction

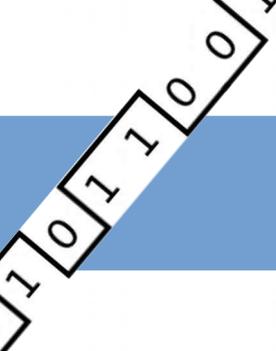
- Dans un premier temps, l'objectif n'est pas que vous sachiez **créer** des objets mais que vous sachiez les **utiliser**
- “La P.O.O. n'est pas faite pour faciliter la vie du **développeur** des classes, mais est faite pour faciliter la vie de **l'utilisateur** des classes”
- Ce cours suppose une connaissance de base de la programmation impérative en C, il y fait référence via des “équivalent à”
- Ce cours utilise la syntaxe du langage C++ mais la plupart des concepts s'appliquent à d'autres langages orientés objet
- Ce cours ne prétend pas être exhaustif !

- Plusieurs langages utilisent ces concepts :
 - C++, Python, Ruby, Java, Visual Basic.....



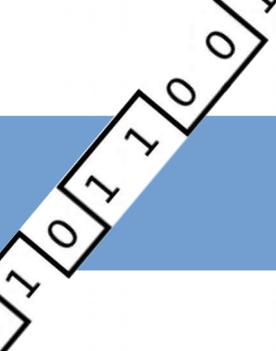
La P.O.O., c'est quoi?

- Un paradigme de programmation informatique.
- Définition et Interaction de briques logicielles appelées objets.
- Un **objet** représente un **concept** et possède:
 - Structure interne (Données)
 - Comportement (Code)
 - Capacité d'interaction avec d'autres objets (Interface)
- Décomposition des problèmes différente de l'approche purement impérative.



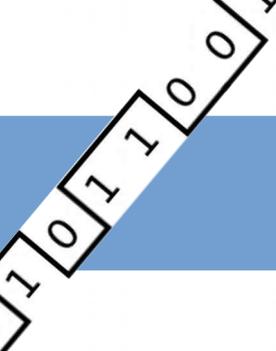
La P.O.O., c'est quoi?

- Avantages:
 - Meilleure structuration des programmes
 - Ré-utilisabilité accrue du code
 - Utilisation plus facile
 - Hiérarchisation
 - Protection de la cohérence des données
- N'est pas forcément plus puissant ou mieux que la programmation impérative, mais est utilisée plus largement pour des gros projets



Mais un Objet, c'est quoi?

- Un **état** défini par une structure de données constituée de ce qu'on appelle **les attributs**
- Un **comportement** défini par un ensemble d'actions possibles qu'on appelle **les méthodes**
- Une **Classe** est une **structure** informatique en POO:
 - Variables = **instances** (exemplaire) d'un type: `int i;`
 - Objets = **instances** d'une classe: `Chat minouche;`
- Les attributs et méthodes sont les **membres** de la classe
- Par rapport aux structures que vous avez manipulées en langage C, on peut dire de manière simplificatrice:
 - Les champs d'une structure “équivalent aux” attributs d'une classe
 - Fonctions manipulant les variables de type structure “équivalent aux” méthodes de la classe

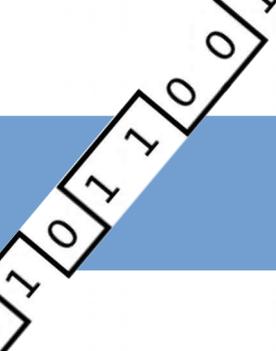


Exemple d'objets : **des chats**

- Pour manipuler des chats (à ne pas faire à la maison... l'auteur décline toute responsabilité en cas de mauvais usage de ce cours...)

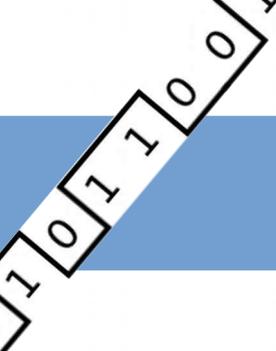


- Création d'une classe Chat



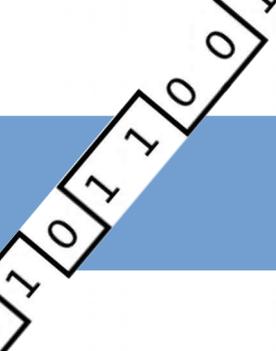
Attribut

- Encode une partie de l'**état** de l'objet
- Typé:
 - Type standard: char, int, tableaux, float....
 - Type énumération
 - Type structure
 - Type objet
- Attribut **d'instance**: propre à chaque instance, équivalent à un champ d'une structure
- Attribut **statique**: partagé entre toutes les instances de la classe, équivalent d'une variable globale mais accessible uniquement via la classe. Utile par exemple pour réaliser un compteur d'un type d'objet



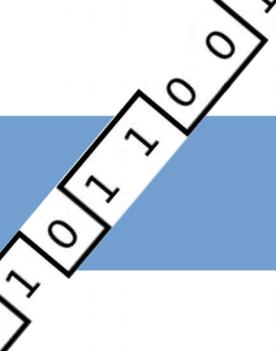
Exemple d'objet : la classe Chat

- Attributs d'instance:
 - Bool *vivant* (*pas adaptée au chat de schrödinger*)
 - *Emplacement* de type lieu avec
enum : lieu{terre,paradis,enfer}
 - *Couleur* de type couleur_chat avec
enum couleur_chat {NOIRETBLANC,BLEU,BLANC,ROUGE};
 - Char nom[100]
 - Int nombreDePattes ;
- Attributs statique:
 - Int *nombreVivants* ; incrémenté chaque fois qu'un chat est créé et vivant dans le programme et décrémenté chaque fois qu'il est éliminé.



Méthode

- Permet d'**agir** sur l'objet
- Une méthode peut être:
 - une méthode d'instance, n'agissant que sur un seul objet (instance de la classe) à la fois
 - une méthode statique (ou méthode de classe) , indépendant de toute instance de la classe (objet).
- La plupart des méthodes peuvent avoir un type de retour comme les fonctions
- Invocation d'une méthode “équivalent à” appel d'une fonction



Méthode (exemple Arduino)

- Exemple de méthode d'instance :

- mySerial1 et mySerial2 sont des instances

```
//Création d'un port Série émulé nommé mySerial1 avec RX=pin 5, TX=pin 6
```

```
SoftwareSerial mySerial1(5, 6);
```

```
//Création d'un port Série émulé nommé mySerial2 avec RX=pin 7, TX=pin 8
```

```
SoftwareSerial mySerial2(7, 8);
```

```
// configuration du baudrate à 4800 bauds pour port série mySerial1
```

```
mySerial1.begin(4800);
```

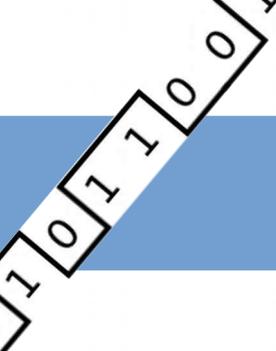
```
// configuration du baudrate à 9600 bauds pour port série mySerial1
```

```
mySerial2.begin(9600);
```

- Exemple de méthode statique :

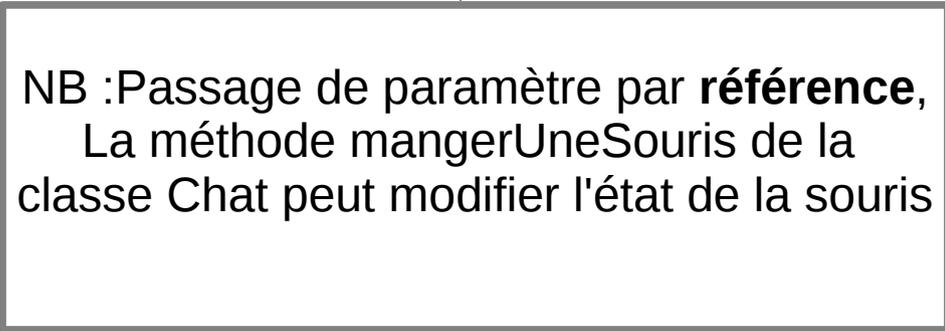
- Serial est une classe

```
Serial.Begin(9600) ;
```

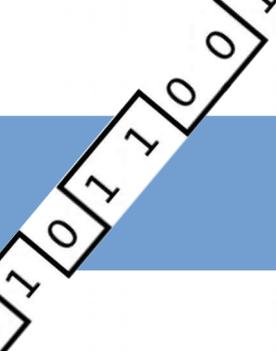


Exemple d'objet : la classe Chat

- Méthodes d'instance:
 - *miauler()*
 - *mangerUneSouris(souris & lapauvresouris)*
 - *deplacer();*
 - ...
- Méthodes statique:
 - *eradiquerTousLesChats()*



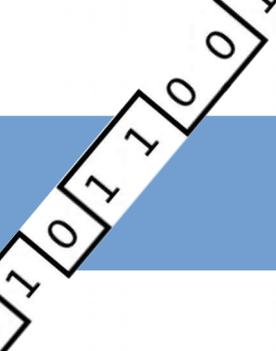
NB :Passage de paramètre par **référence**,
La méthode mangerUneSouris de la
classe Chat peut modifier l'état de la souris



Différentes méthodes

- Constructeur: appelé à la création de l'objet, porte le nom de la classe, paramétrable
 - Permet d'initialiser les attributs de l'objet, de réserver de la mémoire...
- Destructeur: appelé à la destruction de l'objet, porte le nom de la classe précédé de ~
 - Permet de libérer de la mémoire...
- Accesseurs (à préfixer par get): accès à des valeurs d'attributs **privés**
- Manipulateurs (à préfixer par set): modification des valeurs d'attributs **privés**
- Redéfinition d'opérateur (surcharger un opérateur), ex: +, -, *, /, >>, << etc...
- Méthodes abstraites: ne contiennent aucun code. Le code sera défini dans les classes **filles**

- Plusieurs méthodes avec le même nom mais des types et nombre de paramètres différents :
 - Surcharge de fonctions/méthodes et d'opérateur
 - Attention : les types et le nombre des paramètres déterminent quelle version est utilisée, caster si besoin
- Possibilité d'utiliser des valeurs de paramètres par défaut



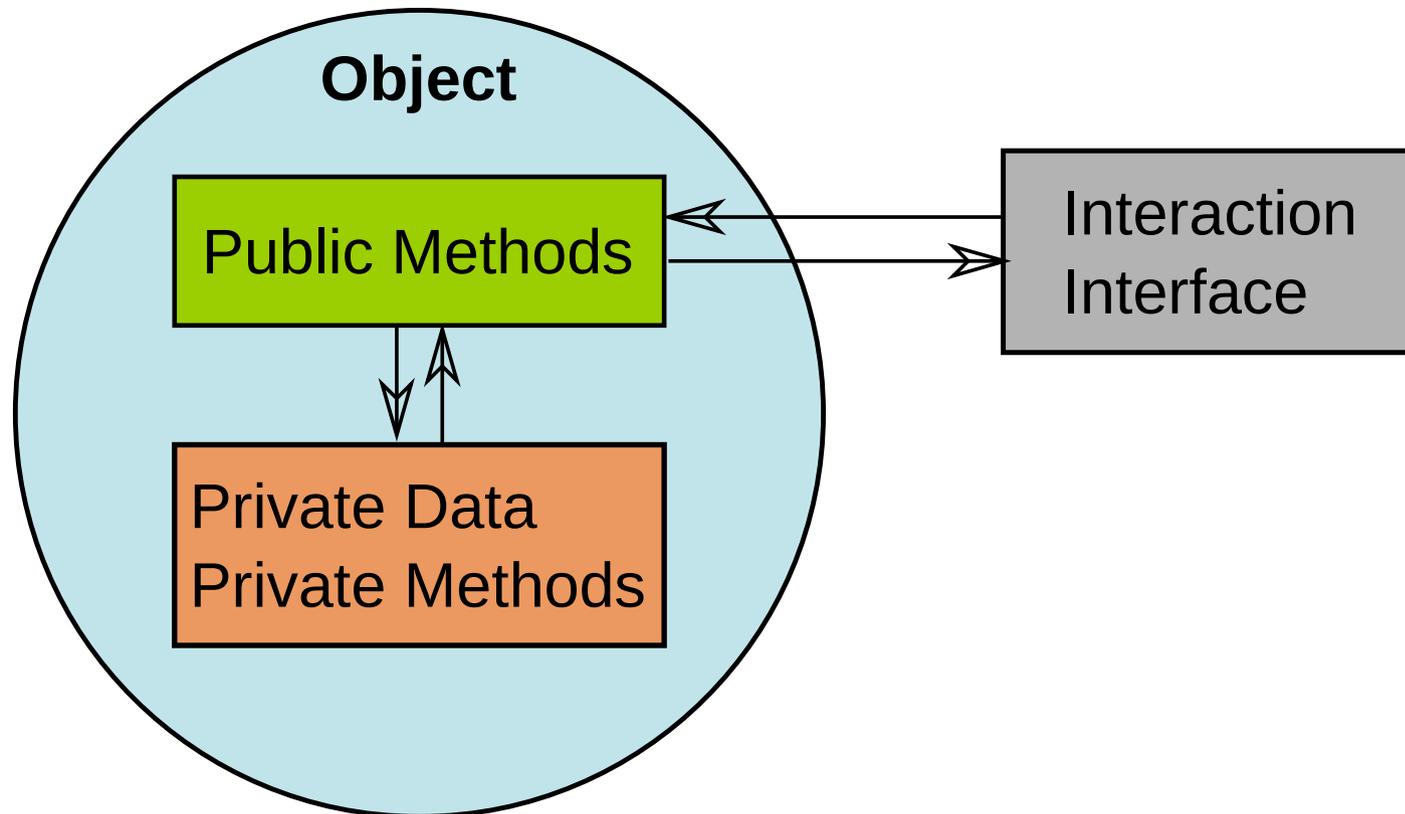
Modificateur de membre

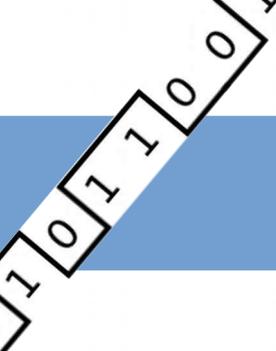
- Un membre (attribut ou méthode) peut être :
 - **private** : accessible uniquement à l'intérieur de la classe qui le contient
 - **protected** : accessible dans toutes les classes **amies**
 - **public** : accessible de partout

 - **static** : commun à toutes les instances d'une classe
- **private/protected** permet de protéger plus ou moins les membres

Principe de l'encapsulation

- Protège l'information contenue dans un objet
- Propose des méthodes pour agir sur l'objet





Exemple d'objet : la classe Chat

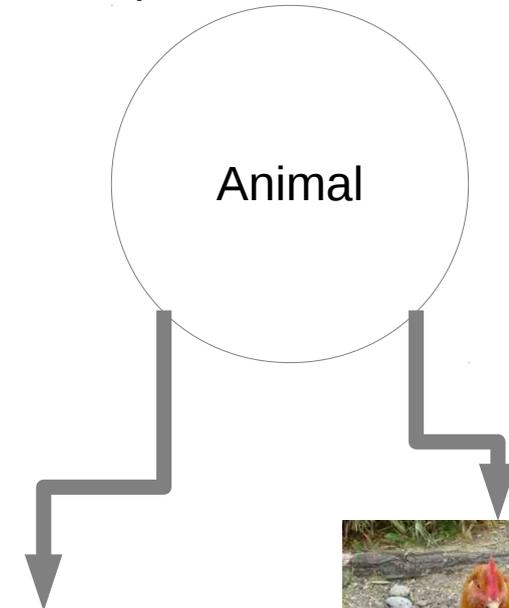
- On veut empêcher qu'un chat *vivant* se retrouve au *paradis* ou en *enfer*... ce serait **incohérent**
- Les attributs *vivant* et *emplacement* sont rendus **private**
- Le constructeur de chat initialise le chat sur *terre* et *vivant*
- On utilise un manipulateur pour changer l'*emplacement*
- Chat ::setEmplacement(lieu nouvelEmplacement)

```
{
  emplacement=nouvelEmplacement ;
  //pourrait aussi s'écrire : this->emplacement=nouvelEmplacement ;
  if (nouvelEmplacement==terre)
    vivant=true ;    //éventuellement, on ressuscite le chat
  else //enfer ou paradis...
    vivant=false ;   //éventuellement, on zigouille le chat
}
```

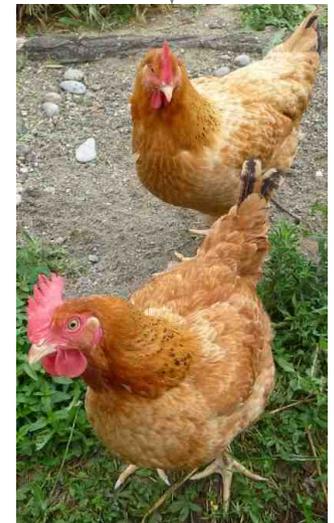
Hiérarchie de classes

- Notion d'héritage
- Classe mère et classe fille
- Différents niveaux
- Permet de mutualiser
 - code et données
- Diagramme de classe
- Documentation + simple

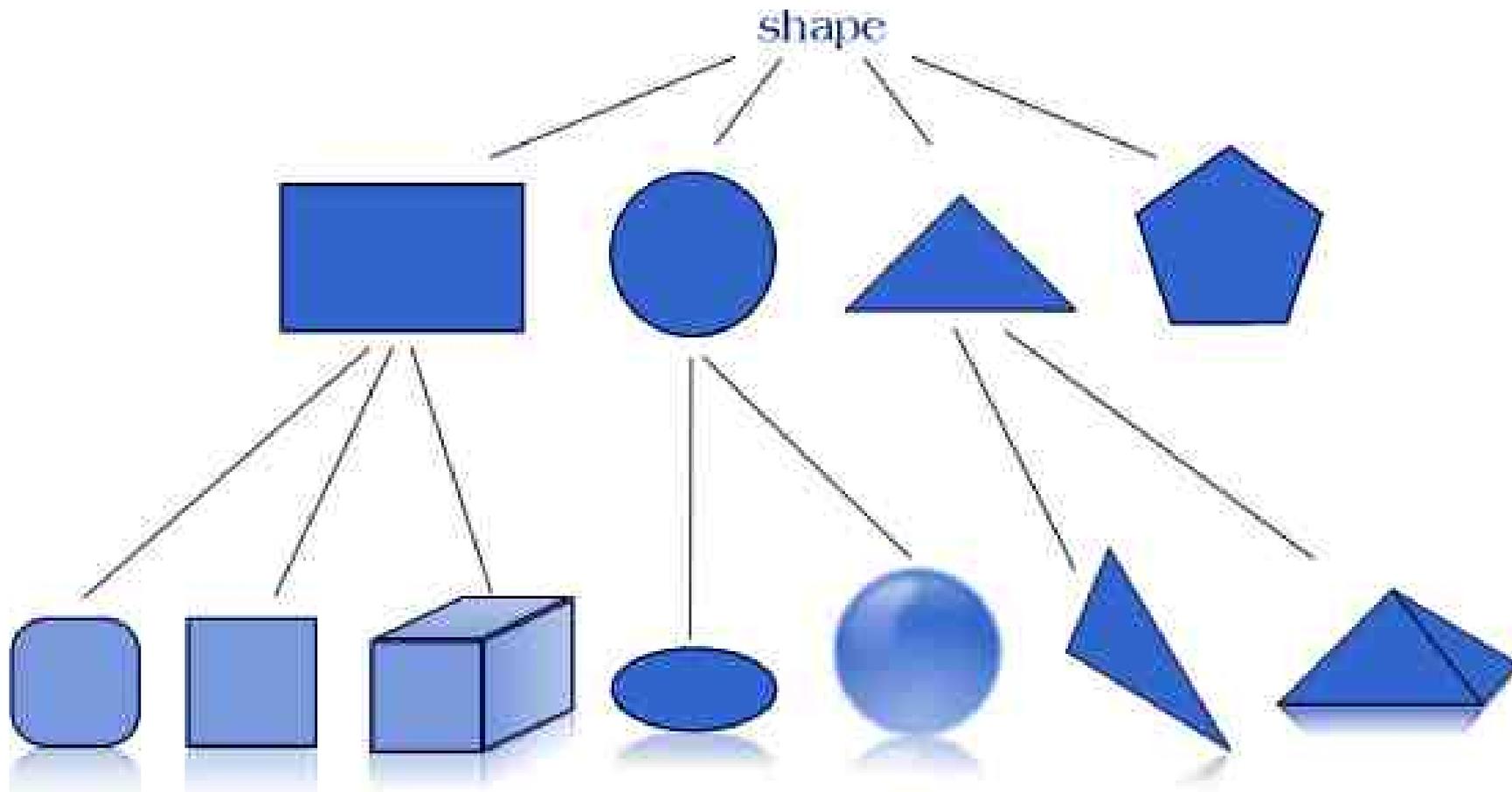
Classe mère :Définit tout ce qui est commun



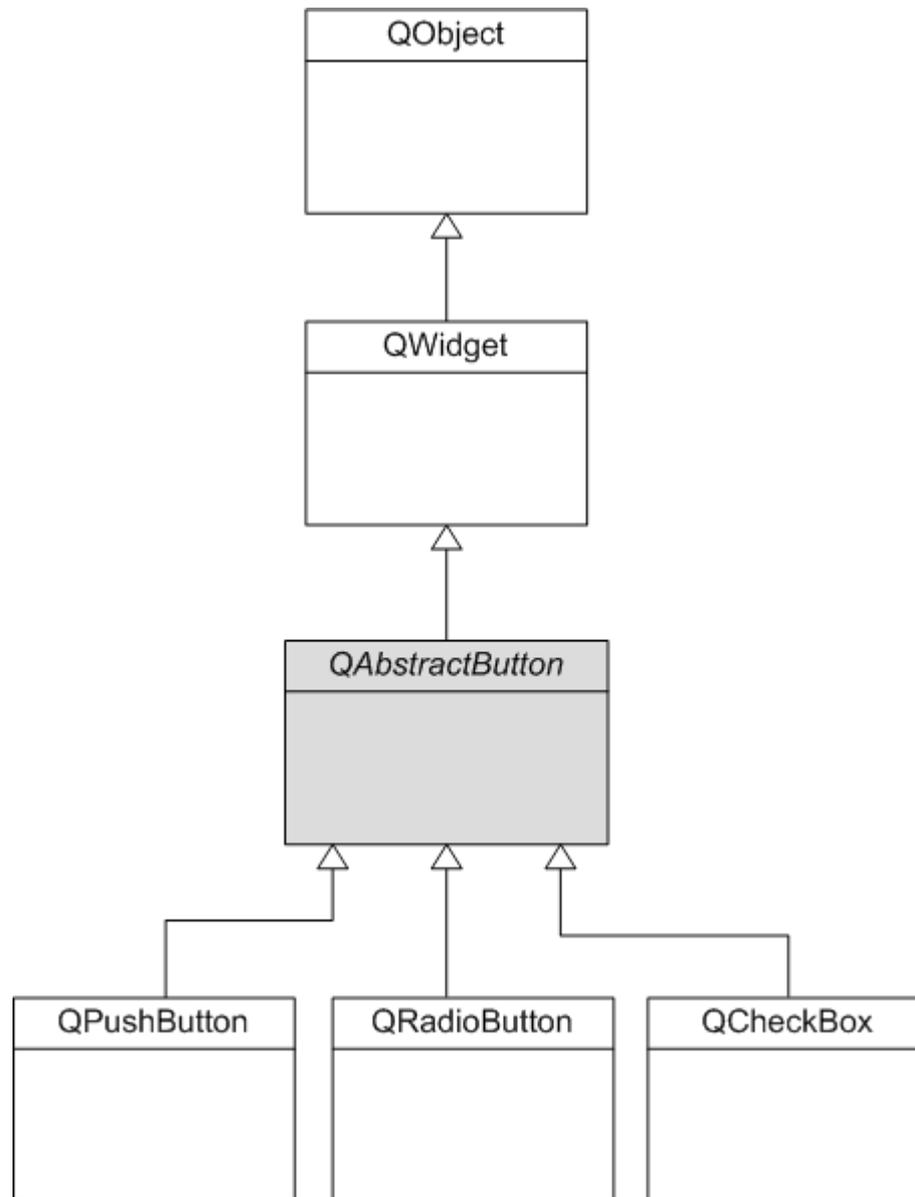
Classe fille :(Re)définit tout ce qui est particulier



Hiérarchie de classes



Hiérarchie de classes



Polymorphisme

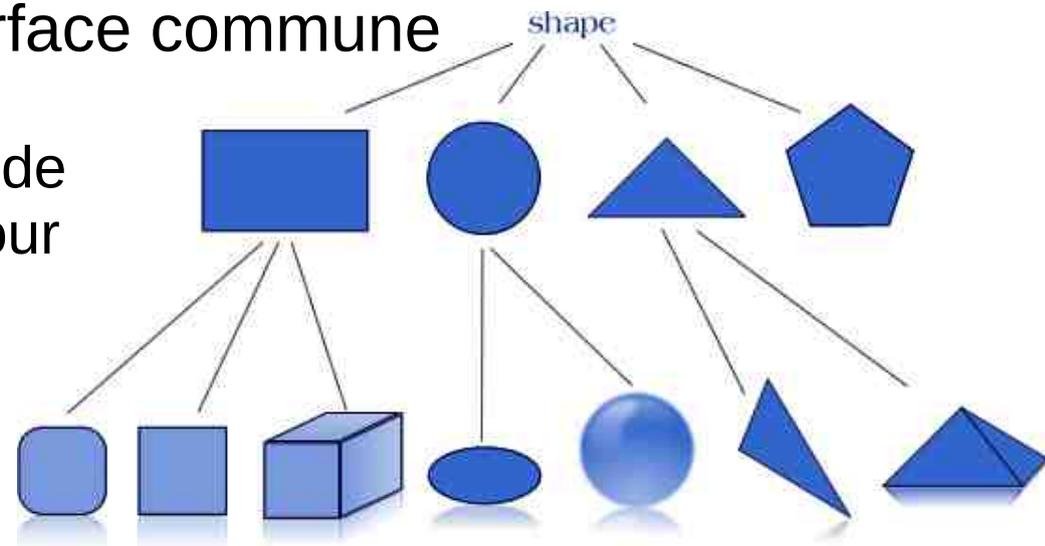
- Permet d'utiliser le même code pour manipuler des objets différents
- Plus simple pour l'utilisateur, qui n'a pas à se poser la question du type d'objet précis qu'il manipule
- **Classe abstraite** : classe qui ne sera pas directement instanciée mais qui sera utilisée comme classe mère
 - Permet de définir une interface commune

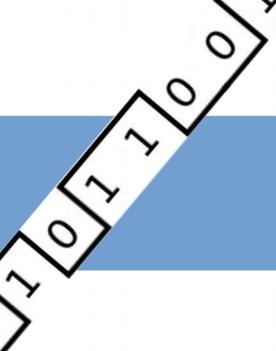
- Exemple :

- Implémentation de la méthode de calcul de surface différentes pour chacune des classes :

- Cercle ::calculeSurface() ;
- Cube ::calculeSurface() ;
- ...

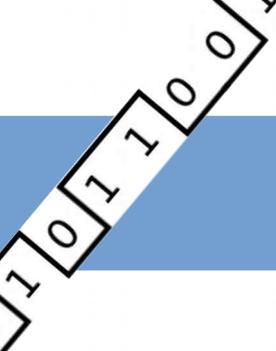
- Mais **appel identique**
objet.calculeSurface() ;





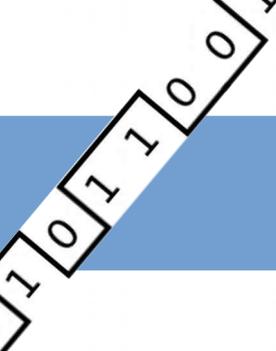
Espaces de noms (namespace)

- Permet de « ranger » des termes (variables, constantes, fonctions....)
- Permet à plusieurs variables ou fonctions d'avoir des mêmes noms, dans des espaces de noms différents
- Pratique lorsque l'on utilise des librairies :
 - Par exemple 2 librairies qui définiraient 2 fonctions ou variables portant le même nom : librairie standard std et une variable cout
 - Spécification de l'espace de noms lors de l'utilisation :
 - `Std::cout << "hello";`
 - `LibrairieCompta::cout= 42 ;`
 - Spécification de l'espace de noms jusqu'à nouvel ordre :
 - `using namespace std;`
 - `cout << "hello";`



Manipulation

- Instanciation comme une variable
 - NomClasse nomObjet(paramètres effectifs du constructeur)
 - Accès aux méthodes et attributs avec l'opérateur .
 - nomObjet.methode(paramètres effectifs de la méthode) ;
 - nomObjet.attribut=33 ;
- Instanciation via un pointeur
 - NomClasse * ptrNomObjet ;
 - Allocation/construction avec opérateur new
 - ptrNomObjet =new NomClasse(paramètres effectifs du constructeur)
 - Accès aux méthodes et attributs avec l'opérateur ->
 - ptrNomObjet->methode(paramètres effectifs de la méthode) ;
 - ptrNomObjet->attribut=33 ;
 - Libération/destruction avec opérateur delete
 - delete(ptrNomObjet)

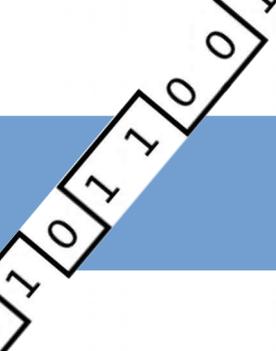


Manipulation

- Implémentation des méthodes de la classe

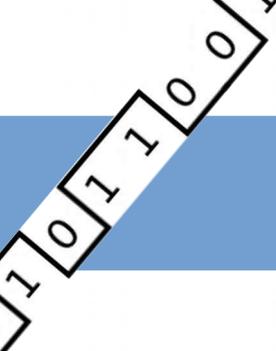
```
type_retour NomClasse::nomMethode(paramètres)
{
    du super code...
}
```

- A l'intérieur des méthodes de l'objet :
 - On peut manipuler les méthodes et attributs de la classe directement
 - this désigne un pointeur sur l'objet :
 - this-> permet d'expliciter le fait qu'on manipule les méthodes et attributs de l'objet en cours



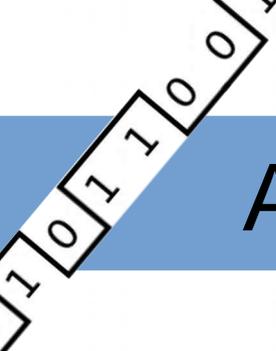
Allocation statique

- Jusqu'ici, vous n'avez vu que les variables décrites de manière statique, elles sont alors construites par le compilateur, par exemple :
 - Un tableau de caractères pour ranger une chaîne : char **chaîne**[1000]
 - Souvent, on ne connaît pas la taille réellement nécessaire pour le tableau à la compilation, il faut dimensionner au pire cas (et encore on n'est même pas sûr..)
 - La mémoire occupée par **chaîne** n'est pas disponible pour le reste du programme (à l'intérieur de la fonction où chaîne est définie)
- Que se passe-t-il si en fait si :
 - On a besoin d'une chaîne plus grande ?
 - On a besoin de la mémoire pour autre chose ?
 - ...



Allocation/libération dynamique

- L'allocation dynamique permet de « construire » et « détruire » les variables en fonction des besoins.
 - Définition d'une variable pointeur dans le programme, il pointe initialement sur rien de particulier
 - Demande d'une zone mémoire libre pour créer l'objet
 - Le pointeur est mis à jour pour indiquer cette zone allouée
 - Appel de la méthode constructeur de la classe
- Il faut gérer proprement la libération de la mémoire par le programme
 - Demande de libération de la zone mémoire à partir du pointeur
 - Appel de la méthode destructeur de la classe
 - Pour les fainéants, éventuellement ramasse miette (garbage collector) en langage JAVA, mécanisme proche avec les smart-pointers en C++11 ?



Allocation/libération dynamique C/C++

- En C : fonctions malloc / free

```
type * ptr_var;  
ptr_var=(type*)malloc(sizeof(type)*nbelements);  
free(ptr_var);
```

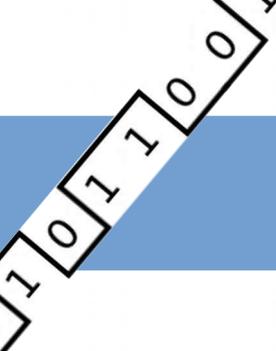
- En C++ : new /delete

```
classe * ptr_var;  
ptr_var=new classe( paramètres éventuels du constructeur)  
delete(ptr_var);
```

- Appel automatique des constructeurs et destructeurs de la classe
- Pour des tableaux :

```
int * tableau = new int[ 10] ; // alloue un tableau de 10 entiers  
delete [] tableau; // ATTENTION : ne pas oublier les crochets []
```

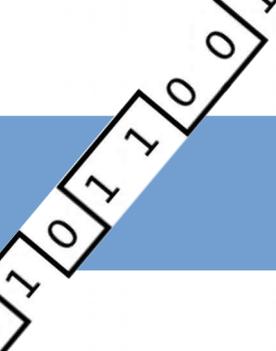
- Mais on peut plutôt utiliser la classe std::vector



Gestion des exceptions

- Permet de gérer plus facilement les cas problématiques
 - Exemple : allocation mémoire
 - En C : malloc renvoie NULL en cas de problème
 - Il faut tester la valeur de retour et programmer ce qui doit se passer pour chaque cas.
 - En C++ : new **lève une exception** en cas de problème
 - C'est équivalent à exécuter de manière automatique la méthode `std::bad_alloc` (un peu comme une interruption en réponse à un problème)
 - Exemple pour gérer la division par 0 :

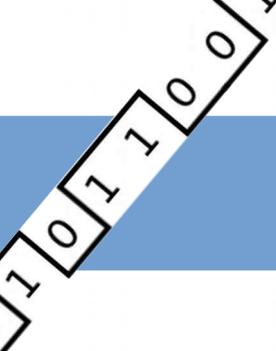
```
int division(int a,int b){
    try {
        if(b == 0)
            throw string("Division par zéro !");
        else
            return a/b;
    } catch(string const& chaine) {
        std ::cerr << chaine << endl;
    }
}
```



Gestion de flux

- Opérateurs << et >>
- Rappels **cin** et **cout**: <http://openclassrooms.com/courses/du-c-au-c/premier-programme-c-avec-cout-et-cin>
- Exemple si on a défini les opérateurs << et >> sur la classe *chat* pour manipuler l'attribut *nom*:

```
Chat minouche ;  
cout << "quel est le nom du chat ? "  
cin >> minouche ;  
cout << "le plus beau chat est " << minouche << " << endl;
```
- Lecture/écriture sur : la console, des fichiers, des interfaces...



Généricité

- Classes et fonctions **template** (modèle)
- Exemple de définition d'une fonction template qui renvoie le plus grand entre A et B (de type quelconque du moment que l'on peut les comparer avec l'opérateur >)

```
template<typename T>  
const T & Max( const T & A, const T & B )  
{ return A > B ? A : B;}
```

- Exemple d'utilisation :

```
int a=3,b=5;    float c=3.142,d=1.618;  
cout << "le plus grand entier entre " << a<< " et " << b<< " est " << Max( a,b ) << endl;  
cout << "le plus grand flottant entre " << c<< " et " << d<< " est " << Max( c,d ) << endl;
```



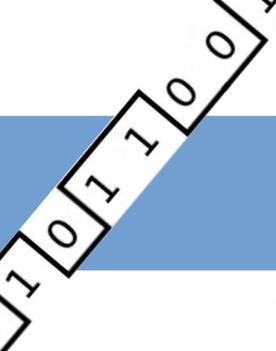
Organisation

- L'organisation du code d'une classe est normalisée pour permettre son intégration facile en tant que librairie
- Fichier h ou hpp
 - Contient la définition de la classe :
 - Nom, héritage, méthode, attributs, méthodes inlineés (ex : templates)
 - Contient les macros
 - #define ...
- Fichier cpp
 - Inclus le .h : #include « librairie.h »
 - Contient l'implémentation des méthodes de la classe (le code)



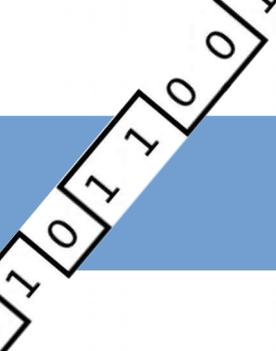
QT et Qtcreator

- Ensemble de bibliothèques (gestion fenêtres, réseau, temps, E/S, ...)
- Outil de développement : Qtcreator
 - Éditeur
 - Débogueur
 - Outils de dessin d'interface graphique
 - Gestion de version
- Multi plate-forme
- Gratuit
- Simple à installer
- Simple pour gérer les projets (utilise qmake)



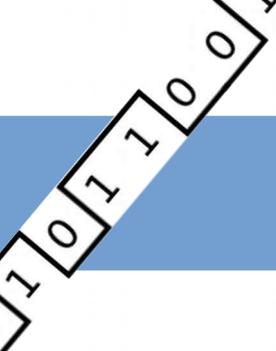
QT : Programmation par événements

- Mécanisme proche des interruptions matérielles
- C'est un logiciel qui se charge d'appeler les bonnes méthodes en fonction des événements :
 - Ex : slots QT, méthode d'un bouton appelée automatiquement quand on clique dessus
- Gestionnaire de signaux déjà implémenté par la librairie
- Ex : pour créer un programme avec une fenêtre :
 - Créer une classe de fenêtre personnalisée qui hérite d'une fenêtre générique déjà décrite par la librairie.



Mise en oeuvre

- Mise en oeuvre des concepts sur un exemple
 - Implémentation d'un compteur générique



Mise en oeuvre

- Mise en oeuvre des concepts sur un exemple
 - Implémentation d'une FIFO :
 - Approche purement impérative avec variables globales
 - Approche impérative utilisant une structure pour la gestion d'une FIFO
 - Approche objet