Programmation Orientée Objet
Java

Nicolas Zinovieff

15 janvier 2007

Table des matières

1 Vers la programmation objet

1.1 Rappels

1.1.1 Fonctions

Une fonction est une succession d'instructions effectuées dans l'ordre et toujours de la même façon.
Une fonction informatique peut être représentée par une fonction mathématique, ou une succession de boites noires. Une fonction simple s'exprime sous la forme
$resultat = fonction(param1, param2, etc...)$.
Une fonction a donc un domaine d'application, et un domaine de résultats.

Une fonction plus compliquée peut modifier ses paramètres à l'aide des pointeurs (passage par adresse), mais il s'agit d'une façon détournée de renvoyer plusieurs informations sans avoir à créer de structure dédiée, et sans copier les valeurs plusieurs fois.

1.1.2 Programmation Séquentielle

La programmation séquentielle fait appel aux fonctions, et dénote un besoin d'ordre : les fonctions sont des séquences d'instructions (élémentaires, ou appels de fonction) qui se font toujours selon le même enchaînement. En plus d'avoir un domaine d'application et de résultats, le but de la programmation séquentielle est généralement de garantir la reproductibilité des executions. Etant donné un état et des paramètres de départ, le résultat ne doit pas varier.
On dit généralement que la programmation séquentielle traite des problèmes opérationnels.

1.1.3 Modules

Il est naturel de regrouper les fonctions au regard de leurs domaines d'applications et d'en faire des modules ou des librairies. Par exemple, ce qui concerne le son ou l'image sera groupé ensemble, et aura des paramètres similaires.

Typiquement, nos fonctions liées au son auront une signature de la forme :
$SON * nouveauSon()$;
$int\ chargerSon(SON *\ s,\ char *\ chemin)$;
$int\ sonPlusAigu(SON *\ s,\ float\ amplitude)$;
$int\ sonPlusRapide(SON *\ s,\ float\ dilatation)$;

etc...
Ce regroupement "naturel" se fait au sein de bibliothèques de fonctions, et garde une certaine cohérence, de façon a être plus utile et plus réutilisable.

1.1.4 Programmation évenementielle

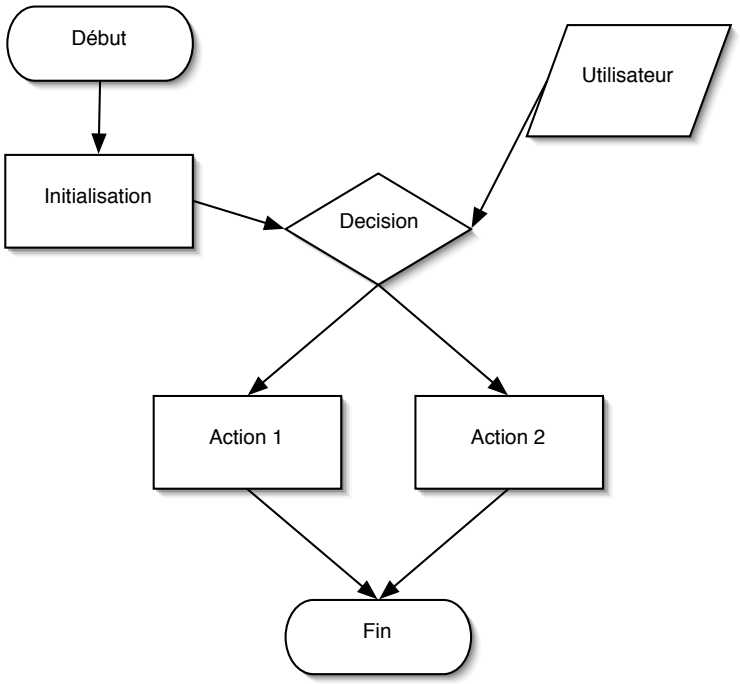
Il existe des situations dans lesquelles le déroulement du programme peut être variable. On pense notamment aux jeux ou tout autre programme dans lequel l'intervention d'un utilisateur est non-seulement souhaitée, mais indispensable.
Dès lors, l'exécution n'est plus linéaire. Elle dépend d'un certain nombre d'actions dont l'ordre ou les paramètres sont inconnus au moment du lancement.
On parle alors de programmation événementielle :
Le déroulement du programme suit alors un schéma de décision, avec des points d'attente. A chaque point d'attente, le programme décide, en fonction de ce qui lui est transmis, de basculer dans telle ou telle branche (généralement par appel de fonction).
On se retrouve alors dans un modèle dit d'automate décisionnel.

1.1.5 Automates et noeuds

Avec un automate, on a deux grands types d'éléments :

- les noeuds, qui représentent des points d'arrêt, des moments où le programme doit "décider" de suivre telle ou telle branche.

– les branches, qui représentent des actions ininterrompues. On les appelle également transitions.

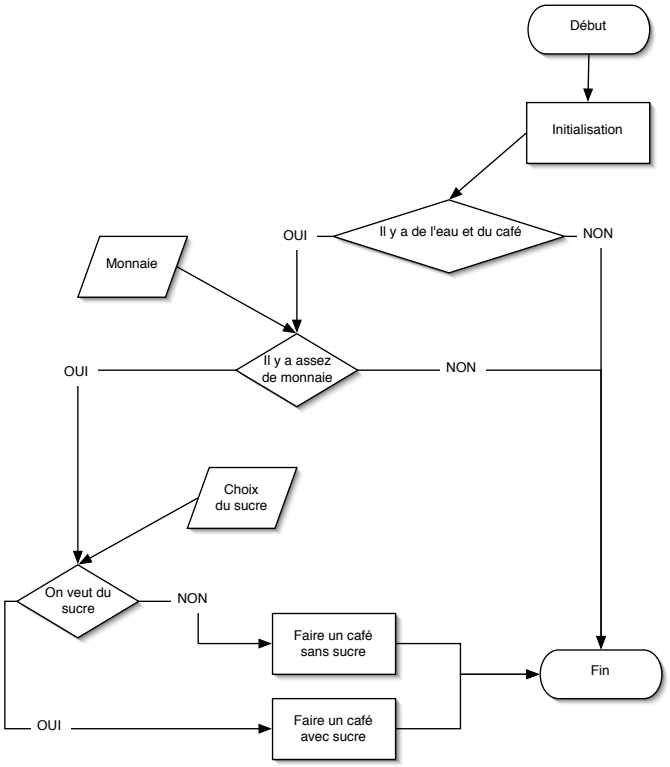


Si on change de perspective, ces noeuds sont en réalité indépendants les uns des autres : s'ils ne sont jamais atteints, cela ne les empêche pas d'exister.
Ces éléments peuvent être considérés sous un autre angle : et s'ils étaient des modules, avec des entrées et des sorties ? Les branches seraient alors des appels de fonctions.

On pourrait alors imaginer ces noeuds comme des acteurs, des "objets", qui se parlent les uns aux autres et gèrent localement les interactions avec les autres noeuds présents dans le système.

1.1.6 Example : la machine à café

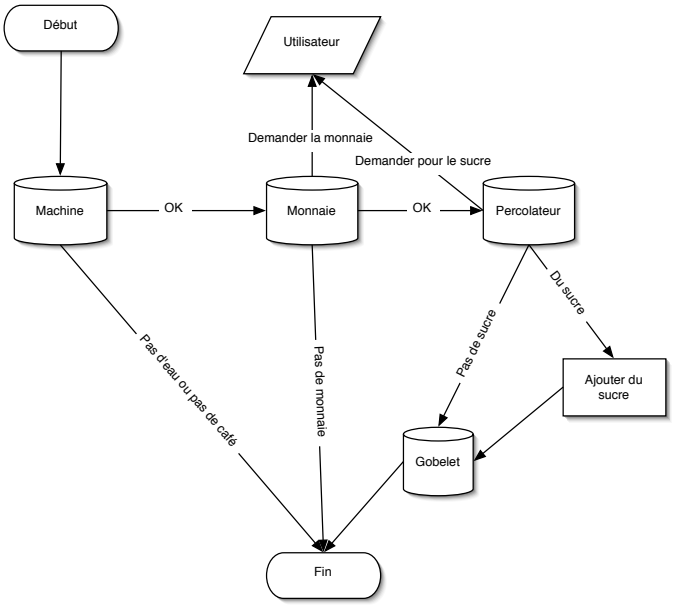
Soit une machine à café. Elle a un automate décisionnel simple et se base sur quelques interactions fondamentales avec l'utilisateur : la monnaie, la sélection du sucre, et la sortie/gobelet, ainsi que les différents tests internes.



Quels sont les acteurs ici ?

– L'utilisateur, évidemment
– Le système de sécurité de la machine à café
– L'automate à pièces
– Le "fabricant" de café

Le schéma peut alors aussi s'écrire sous la forme suivante



1.2 Vers un nouveau paradigme

On a donc vu qu'en partant de la programmation séquentielle, pour certains problèmes, on pouvait en arriver au point où il était plus simple de considérer des interactions entre acteurs. Ces acteurs, ces "objets", communiquent entre eux au travers de fonctions, ou d'appels.
La programmation objet est née de ce besoin d'exprimer des relations, des interactions, entre des éléments qui existent indépendamment les uns des autres. Le problème n'est alors plus lié à un ordre, à un souci opérationnel, mais à un problème de système, de données.

La question devient donc "comment lie t'on les données, comment les articule t'on ?"

1.2.1 Objets

On s'est aperçu que ces acteurs représentaient plus où moins des éléments "physiques" du système considéré. La modélisation en objets ressemble donc fortement à une modélisation physique, dans lequel on identifie les objets et les lois qui s'y appliquent.
Pour mettre au point ce type de programmation, on a donc essayé de revenir a une approche plus "naturelle" du programme, pour que cela ressemble à ce qu'on côtoie tous les jours.

Lorsqu'on pense au monde physique, on pense en objets. Un objet a des propriétés de famille, et des propriétés intrinsèques. Le fait qu'une porte soit en bois, en plastique, transparente ou peinte, ne nous empêche pas de reconnaître une porte. C'est donc que toutes les portes partagent des caractéristiques communes, même si elles sont différentes les unes des autres. De même, tous les objets de l'univers (planètes, vélos, percolateurs, ordinateurs, personnes) sont "classables" en familles.

Vocabulaire
Une famille d'objets est appelée une classe d'objets. Elle énumère les caractéristiques propres à tous les membres de celle-ci.
Un objet en particulier est appelé une instance. Il a des propriétés qui lui sont propres, et qui ne sont pas partagées par les autres objets de la même classe.

Si on prend comme exemple le vélo, le fait qu'il ait deux roues et des pédales sont des propriétés de classe (tous les vélos dont faits comme ça, sinon ca ne serait pas des vélos).
Par contre, sa position dans l'espace, et la présence de petites roulettes sont des propriétés de l'instance (heureusement que tous les vélos de l'univers ne sont pas au même endroit, et qu'on peut abandonner les roulettes après avoir appris, sinon le vélo ne servirait à rien).

1.2.2 Hiérarchie

On peut identifier des sous groupes dans une famille. Par exemple, un 4x4 reste une voiture, et un Range Rover reste un 4x4.
On doit hiérarchiser les objets, de façon a identifier les propriétés communes et les spécificités de chacun.

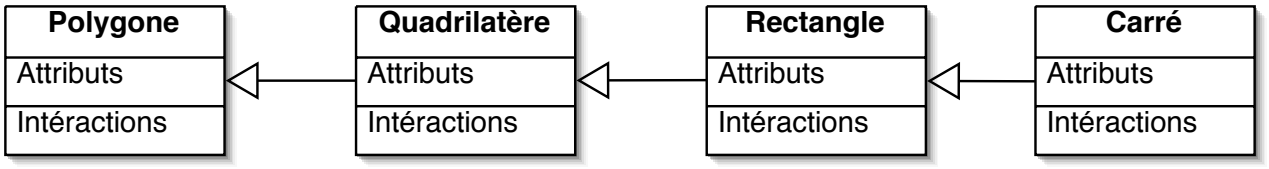
On peut imaginer plein de hiérarchies comme celle-ci :
- un carre est un rectangle, qui est un quadrilatere, qui est un polygone
- un chien est un mammifere, qui est un vertebre, qui est un animal
- etc...

Toutes les propriétés sont partagées, "héritées". Par exemple, un polygone a plusieurs côtés, donc un quadrilatère a également plusieurs côtés (4 pour être précis), donc un rectangle a aussi plusieurs côtés (4, deux à deux égaux), donc un carré a enfin plusieurs côtés (4, tous égaux).
On dit que le carré hérite, ou spécialise le rectangle.
La classe "carré" est donc la classe fille de la classe "rectangle".

1.2.3 Parenthèse : UML

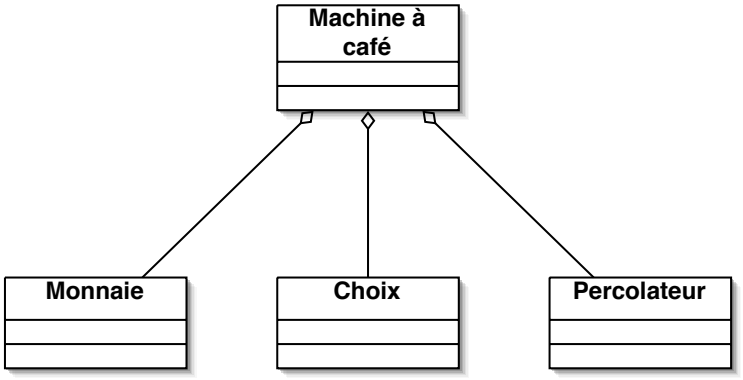
Plutôt que de décrire systématiquement les classes, on a mis en place une façon de les dessiner UML (Unified Modeling Language).

Les rectangles représentent des classes, et sont divisés en deux : les propriétés et les actions (*cf* Méthodes et intéractions).
L'héritage, quant à lui, est représenté par des flèches.

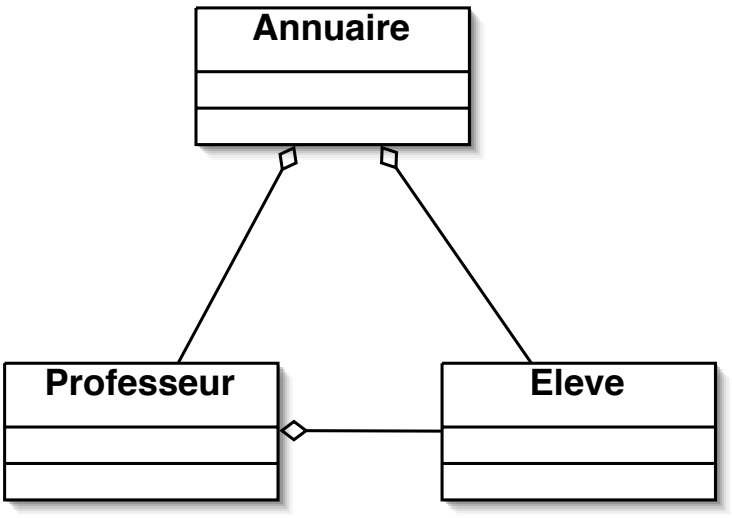


1.2.4 Agrégation

On appelle agrégation le fait de faire référence à d'autres objets (dans un système, on parlerait de "contenance"). Une structure en C est une agrégation d'éléments, et le même principe est applicable à l'univers des objets.
Un exemple simple de relation de contenance peut être tiré de l'exemple de la machine à café : la machine contient l'unité de monnaie, celle des boutons de choix du sucre, et le percolateur.



Attention, cette agrégation peut aussi être plus abstraite. Un annuaire d'école *contient* les noms des professeurs et des élèves, mais le professeur *connaît* ses étudiants.



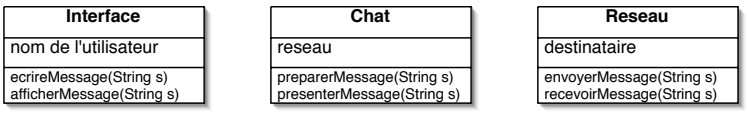
1.2.5 Interactions

Les objets dans notre univers interagissent entre eux. Dans le cas de la machine à café, celle-ci se repose sur les différents objets qui la composent pour réagir.
Dans la pratique, la machine "passe la main" au premier de ses composants, lui laissant le choix de se comporter comme bon lui semble, avant de passer à son tour le flambeau.
Ce passage de message, cette interaction, est en fait une fonction. La machine appelle la fonction "demande de l'argent" à son monnayeur.

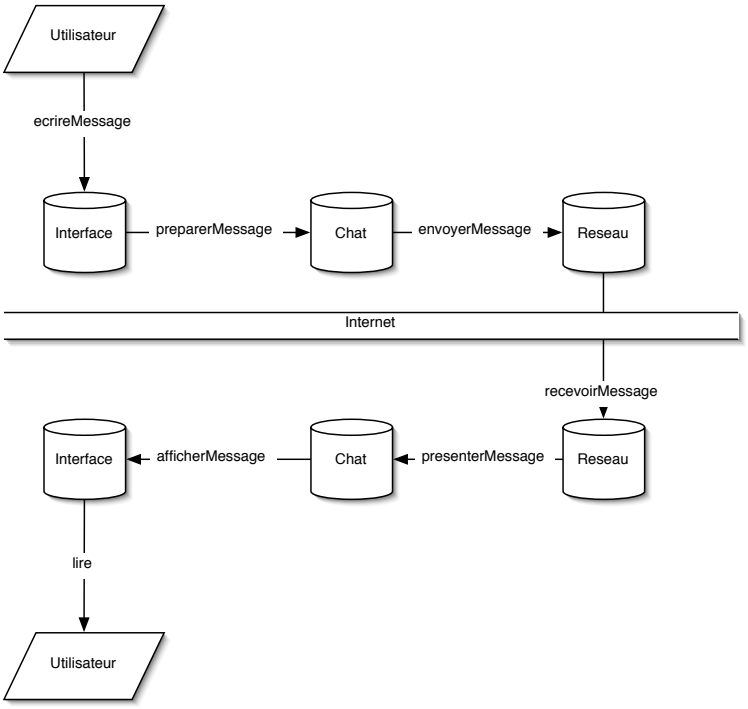
Imaginons un programme dont la tâche est de faire communiquer deux personnes au travers d'internet. Identifions les acteurs :

- les participants, bien sûr
- le module de chat
- le réseau

Quand on envoit un message, que se passe t'il ?
L'utilisateur tape son message. Celui ci arrive dans le module de chat, par l'intermédiaire d'une interface graphique quelconque. Le module de chat prépare les données, et les envoit au réseau. Le réseau transmet ses données au module de chat en face, qui les présente a son utilisateur.
A chaque action correspond une fonction de l'objet **cible**. En effet, chaque élément reçoit des messages et agit en conséquence, il ne les réclame pas. Il s'agit en quelque sorte d'évènements.
On peut modéliser nos classes comme suit :



Si on regarde l'enchaînement des actions effectuées, cela donne quelque chose comme cela :

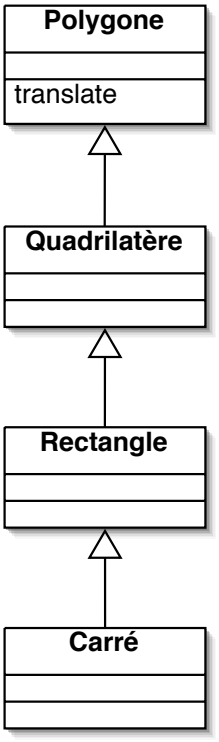


1.2.6 Hiérarchie d'intéractions

Puisqu'on peut hériter des caractéristiques d'une classe d'objets, on peut aussi hériter de leur comportement. Reprenons l'exemple du carré :

- on peut appliquer une translation à n'importe quel polygone (appelons cette fonction de façon très originale *translate*)
- **donc** un quadrilatère peut faire également l'objet d'une translation. Quelle est la différence entre la translation d'un polygone et celle d'un quadrilatère ?
  Il n'y en a *aucune* ! Par conséquent, la fonction *translate* reste inchangée, elle est héritée du polygone
- ...
- Pour finir, on translate le carré comme le rectangle, lui même se comportant comme un quadrilatère, lui même se comportant comme un polygone.

En pratique, rien ne change dans la façon de translater un carré, par rapport à un polygone. L'héritage est direct et inchangé.



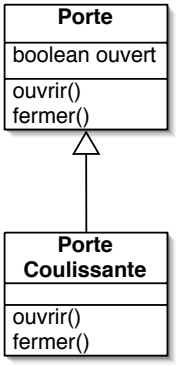
On notera que les fonctions héritées sont omises chez les descendants. En effet, TOUTES les fonctions qui existent dans les classes antérieures sont utilisables dans les classes filles

1.2.7 Héritage et surcharge

En pratique, tout se passe comme si les fonctions (aussi appelées méthodes) étaient "collées" dans les classes filles. Dans le monde des objets, une instance se comporte par défaut comme l'indique sa classe, ainsi que toutes ses classes ancêtres.
Dans le cas des translations vues précédemment, lorsque je veux translater un carré, j'appelle la méthode *translate* du carré. Celui ci n'a pas de méthode avec ce nom, donc il demande a sa classe mère, et ainsi de suite jusqu'à ce qu'une définition soit trouvée.

Reprenons un instant l'exemple de la porte, vu au tout début. La fonction première d'une porte est de s'ouvrir (et de se fermer). Oui, mais une porte coulissante ne s'ouvre pas du tout de la même manière qu'une porte-fenêtre, ou qu'une porte "classique".
Le phénomène d'ouverture est bien une méthode de la classe **porte**. Toutes les portes s'ouvrent, c'est bien connu. Comment intègre t'on donc une différence aussi minime de comportement ?
Si l'on regarde comment le carré se translate, on s'aperçoit qu'on remonte la hiérarchie *jusqu'à ce qu'on trouve une définition*. La solution consiste donc à écrire une **nouvelle** définition, pour la même fonction. Cette opération s'appelle la surcharge.

Pour en revenir à nos portes, l'ouverture "classique" est une rotation de la porte autour de ses gonds. La classe **porte coulissante**, qui hérite de la classe **porte**, va surcharger la méthode ouvrir, en précisant cette fois ci que le mouvement est remplacé par une translation. Dans un diagramme UML, cela se représente tout simplement comme suit :



On notera que le paramètre "ouvert", qui est hérité, n'est pas remis, puisqu'on ne le change pas.

1.2.8 Partage

Il arrive quelques fois que deux objets qui n'ont aucun lien d'héritage partagent des caractéristiques communes. Par exemple, une batte de baseball et un marteau sont tous les deux considérés par la justice comme des "objets contondants". Ou bien un bateau et un hydravion sont tous deux capables de naviguer.
Le partage de caractéristiques en dehors de tout héritage existe, et doit donc être modélisable.

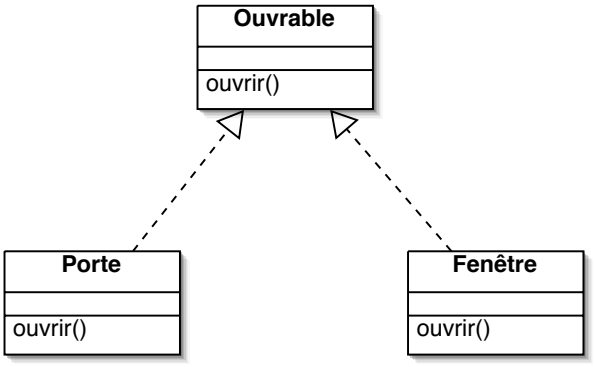
Spécifications En électronique, on appelle ces caractéristiques des spécifications. Généralement, il s'agit de préciser des fonctions "obligatoires". En programmation objet, ces spécifications peuvent prendre plusieurs formes, en fonction de la façon dont elles sont réalisées.

Quel est l'intérêt ?
Il arrive parfois qu'on doive grouper ensemble des éléments *a priori* différents, mais qui doivent effectuer les mêmes actions au sein d'un ensemble. Si on parle par exemple de météorologie, on a des thermomètres, des anémomètres, des baromètres, etc...
Tous ces instruments ne mesurent pas la même chose, ne fonctionnent pas de la même façon, etc... Mais mis ensemble dans une station météo, tous doivent transmettre leurs informations d'une manière homogène : les fabricants ne veulent pas avoir a gérer des moyens de connexion tous différents. Donc ils se plient a une interface commune, un port compatible... Cette notion de port ou d'interface est la face "réelle" de la spécification : un point de passage commun entre des objets qui fonctionnent de façon différente.

Catégorie - Interface - Classe Abstraite Prenons un exemple : la porte et la fenêtre. Ce sont deux objets qui n'ont aucun lien de parenté, puisqu'une porte n'est pas une fenêtre, et vice versa.
Pourtant, toutes deux sont capables de s'ouvrir. Quelle est la spécification de cette ouverture ?
L'ouverture prend comme paramètre une autorisation (clef, verrou, code, peu importe), et le résultat est une transformation spatiale de l'objet.

On peut modéliser cela de plusieurs façons :

Classe Abstraite On pourrait imaginer une "fausse" classe, mère de porte et fenêtre, qui comporterait une méthode *ouvrir*. Etant donné que l'ouverture d'une porte et d'une fenêtre n'a pas le même résultat, ni le même mécanisme, cette fonction d'ouverture est un simple prototype (comme en C). Le corps de la fonction est à remplir par les classes filles.
On a donc un diagramme qui ressemble à ça :



Vocabulaire
Une classe abstraite qui ne contient que des prototypes, aucune définition, est appelée *Interface*.

Petite parenthèse UML :
L'implémentation d'une interface est souvent représentée par une flèche d'héritage en pointillés (non standard)

Catégories On a parlé de comportements communs, dont l'implémentation est laissée à la classe fille. Il existe aussi des phénomènes dont l'implémentation est connue, et identique quelle que soit la classe.
Par exemple, une porte coulissante et une fenêtre coulissante s'ouvrent *de la même manière*. C'est pour ça qu'on dit qu'elles sont coulissantes. Lorsque le "comment" est connu, et ne change pas, on parle de catégories, plutôt que d'interfaces.
Les catégories sont des entités à part et très peu utilisées. Nous ne les aborderons donc pas.

Quelques grands types d'interfaces
- **Les interfaces "utilitaires"**
  Peut se rapporter à un comportement utilisable : ouvrable, mangeable, executable, etc...
  Ces interfaces comportent au moins une méthode, et servent généralement à forcer la standardisation.
  Quelques exemples pratiques : les drivers, les ports série, etc...
- **Les interfaces "d'identification"**
  Ne servent généralement qu'à mettre dans un même sac des objets disparates. On s'en sert pour dire "ces objets appartiennent à un même groupe, mais ne partagent rien d'autre".
  On pourra dire par exemple que la nationalité s'applique aussi bien à des personnes, que des biens, ou des bâtiments, ou des véhicules...

2 Les différents langages objet

2.1 Rappels

Compiler un programme revient à "traduire" un langage dans un autre (généralement d'un langage "humainement compréhensible" dans un langage machine). Cette compilation intervient pour des raisons diverses, mais le but est toujours de transformer des intentions en executions.
Tous les langages dits "Turing complets" (qui peuvent exprimer toutes les fonctions mathématiques, en gros) sont inter-compilables. Derrière ce mot barbare se cache la notion de compatibilité ou de traduction.
On peut tout simplement dire que tout programme écrit dans un langage peut également s'écrire dans un autre.
La première remarque à ce sujet, c'est que les langages séquentiels (comme le C), et objets (comme le Java) peuvent produire les mêmes programmes. Seules la lisibilité, la complexité, et la vitesse d'exécution seront différentes d'un programme écrit en C à un programme écrit en Java.

2.2 Panorama

On distingue plusieurs grandes familles de langages objets :
- Les langages dits "compilables" : C++ / Java / Objective C /...
- Les langages dits "à bytecode" ou "interprétés" : Java / C# / perl / python / ruby / PHP /...
- Les curiosités, les inclassables : ocaml / NSOF / ...

2.3 Lignes Générales

Pour comparer ces différents langages, nous allons examiner différents critères :
- Gestion mémoire (garbage collector, explicite, ou mixte)
- Indirections (structures, interprétation dynamique par un runtime,...)
- Typage ("Always trust the programmer", typage "strict")

Gestion mémoire En C, la gestion mémoire est explicite : chaque élément doit être déclaré et instancié explicitement ($char * \ c = malloc(1)$), mais aussi et surtout libéré explicitement ($free(c)$).
Quel que soit le langage, la déclaration et la création doivent être explicites, mais la libération peut être prise en charge automatiquement, lorsqu'on n'a plus besoin de l'objet en question, par un *Garbage Collector*.

Indirections En C, l'indirection est minimale : un appel de fonction consiste (en assembleur) à sauter a l'adresse du code et à continuer l'execution.
Dans les langages objets, comme on l'a rapidement vu au sujet de l'héritage, il peut arriver qu'on doive "chercher" la fonction avant de l'executer. Les méthodes de recherche de fonctions, et surtout le nombre d'opérations à effectuer avant de la trouver sont appelées indirections.
Dans les langages proches du C, l'indirection est minimale, elle consiste généralement à chercher dans une table l'adresse du code à executer mais dans d'autres langages, il se peut que cette fonction doive ensuite être "traduite" à la volée, augmentant le nombre d'indirections à l'execution.

Typage Le typage est depuis longtemps un point de friction entre les différents programmeurs. En effet, si une variable n'est qu'une case mémoire, on peut traiter son contenu comme bon nous semble. Qu'est-ce qui nous empêche de traiter un *char* comme un *int* ?
Le problème avec les objets est que non seulement ce sont des cases mémoire, mais qu'en plus ils font référence à des fonctions. Que se passe t'il si on appelle la fonction "ouvrir" sur un marteau ? Que devrait il se passer ?
Les partisans du typage fort soutiennent que ces actions sont illégales dans le langage, et doivent être fortement vérifiées pour éviter des problèmes qui de toute façon ne devraient même pas arriver. Si on appelle la méthode "ouvrir" sur un objet, celui-ci *doit* faire partie d'une classe qui a la méthode "ouvrir".
Les partisans du typage laxiste suivent la règle "always trust the programmer". Si le développeur décide de faire quelque chose qui n'a aucun sens *a priori*, c'est son problème, il doit bien y avoir une bonne raison.

2.4 C++

En C++, la syntaxe et le fonctionnement sont très proches du C. Et pour cause, un compilateur C++ est tout à fait capable de compiler du C. On dit que C++ est un *superset* de C.
Par conséquent, Tout fonctionne comme en C :

- les objets sont des structures C un peu plus complexes
- la gestion de la mémoire est complètement explicite : malloc et free sont "remplacées" par *new* et *delete*, pour les objets
- le niveau d'indirection est relativement petit, dans la mesure où le parcours de structures est rapide
- le typage est extrêmement laxiste, comme en C. Rien n'empêche, à la compilation, de faire un *cast* de Marteau vers Porte. Par contre, ça plantera à l'execution.
- l'héritage est multiple, aidé en plus par un système de templates (catégories)

Le C++ est un des tous premiers langages objets utilisés. Les premiers compilateurs C++ faisaient une traduction en C, avant de faire appel a des compilateurs "normaux".
Il a ses adeptes et ses détracteurs, son avantage principal étant le relatif manque de rigueur, qui permet à des programmeurs aguerris de faire toutes les cabrioles possibles. L'inconvénient majeur de ce langage est que le manque de rigueur conduit souvent à des erreurs incompréhensibles pour des débutants ou des programmeurs plus habitués à l'aide précieuse fournie par les erreurs de compilation.

2.5 Objective C

L'Objective-C est également un superset du C. Le fonctionnement des objets est toutefois complètement différent : il emprunte sa syntaxe et son mode de fonctionnement a *Smalltalk*, un langage assez peu utilisé, aux concepts novateurs pour l'époque : les messages.

- les objets sont des structures C très complexes, voire complètement opaques.
- la gestion de la mémoire est mixte. En réalité, elle dépend le la plateforme utilisée. Il existe des implémentations a gestion explicite (GNUStep), et d'autres avec un Garbage Collector (Cocoa)
- le niveau d'indirection est relativement élevé : chaque objet est une table, chaque appel de méthode requiert une réécriture.
- le typage est extrêmement laxiste. Contrairement au C, il ne s'agit pas d'une absence de contrôle, mais d'une volonté explicite d'accepter tous les appels, quitte à ne rien faire si cela ne correspond à aucune méthode.
- l'héritage est mono-parental, et le langage comporte toutes les variantes de spécifications : classes abstraites, interfaces (appelées protocoles), catégories, etc...

L'Objective-C est un langage volontairement "différent". Si le compilateur ObjC est également capable de compiler du C, les structures et fonctions ne participent que de loin aux mécanismes objet.
Le fondement de ce système est le *message*. Un objet reçoit des messages, et agit en fonction. S'il en est incapable, tant pis, le message est ignoré. Ce côté dynamique permet des acrobaties programmatiques intéressantes tout en conservant les bases "normales" de la programmation.
Par exemple, envoyer "ouvrir" à un marteau ne fera rien, même pas d'erreur, tandis que le même message envoyé à une porte conduira à l'execution de la méthode éponyme.

2.6 Langages "fonctionnels"

Les langages fonctionnels sont une famille à part entière, différents des langages procéduraux (c'est comme ça qu'on appelle les langages tels que Pascal, C, Java,...) tant du point de vue philosophie, que fonctionnement. Dans un langage fonctionnel, on manipule des fonctions. Les variables n'existent qu'au travers de la "fonction" qui les crée, et on peut passer des fonctions en paramètres d'autres fonctions. D'où le nom.
Ces langages sont utilisés principalement dans les milieux académiques. Ils forment la base de la matérialisation des théories informatiques. D'un point de vue programmatique, ils apportent une vision extrêmement différente des problèmes à régler, et sont généralement boudés par les "vrais" développeurs.

Pour prendre un exemple, on peut citer CaML, développé par l'INRIA. CaML est un langage fonctionnel de type "lambda calcul", et comporte une couche objet, baptisée OCaML.
La plupart des langages fonctionnels sont interprétés et ont donc à la fois un niveau d'indirections élevé et un garbage collector efficace.
Le typage est nécessairement très fort, étant donné le domaine d'application scientifique.

2.7 Langages par prototype

Les langages par prototype sont une toute petite famille de langages objets. Ils sont principalement utilisés dans les systèmes embarqués, à cause de leur spécificité : il n'existe qu'une seule instance pour chaque classe.
Grosso modo, vous avez un seul vélo dans l'univers, mais vous pouvez en "fabriquer" un deuxième, en vous

basant sur le modèle déjà existant. Si le vélo universel vous convient, dans ce cas, vous êtes libre d'en faire ce que vous voulez.
Le gros avantage de cette philosophie est que l'empreinte mémoire est extrêmement faible, et surtout globalement constante. Ces langages sont donc bien adaptés aux systèmes qui n'ont pas beaucoup de mémoire.

2.8 Java

Java est un langage assez "ancien", un des premiers conçus exclusivement dans le but d'être objet. Depuis sa toute première mouture, les concepts objet sont présents et respectés, avec quelques particularités notables.

- les objets sont des pointeurs, certes, mais l'accès direct à la mémoire n'existe pas. Même si ils étaient représentés en mémoire par des structures, il n'y aurait aucun moyen d'accéder à leurs composantes.
- la gestion de la mémoire est implicite. Si la création d'un objet se fait avec *new* (comme en C++), il n'existe aucun moyen de détruire un objet explicitement. Cette tâche est échue à un garbage collector, qui libère la mémoire allouée *si et seulement si* la variable considérée n'est plus référencée où que ce soit.
- le niveau d'indirection est relativement élevé : la machine virtuelle étant une application, un appel de fonction se fait dans $l'espace$ Java, qui le traduit ensuite en code machine. $De\ facto$, Java est un langage semi-interprété : un programme est compilé en *bytecode*, sorte de langage machine universel, qui ensuite est traduit à la volée en langage machine.
- le typage est strict. Les concepteurs de Java ont tenu le raisonnement suivant :
  Puisque le même programme Java (sans recompilation) *doit* tourner sur toutes les plateformes, un maximum d'erreurs doivent être évitées à la compilation. Pas moyen, donc, de faire un *cast* de Marteau en Porte...
- l'héritage est mono-parental, et le langage comporte les deux systèmes de partage de base : classes abstraites et interfaces.

Java est le langage retenu pour cet enseignement en programmation objet. Les raisons sont évoquées dans le comparatif : étant donné que la compilation se charge de signaler un maximum d'erreurs, l'apprentissage s'en trouve simplifié. De plus, Java suit la philosophie WORA($Write\ Once\ Run\ Anywhere$) ce qui rend la dépendance à une plateforme donnée obsolète.
Cette même philosophie tend également à supprimer les dépendances à des librairies externes : de fait, le langage Java contient déjà la plupart des outils sympathiques aux développeurs : interfaces graphiques, réseau, multithreading, calculs scientifiques, etc...

3 Java - Syntaxe

3.1 Variables

En Java, on distingue deux types de variables :

- les objets
- les "scalaires"

Ces deux types sont différents et non compatibles. On peut les comparer aux variables en C, entre les pointeurs et les autres types.

3.1.1 Scalaires

Les scalaires sont les types "de base". On les déclare comme en C, et on en compte un (relativement) petit nombre :

- byte
- char
- boolean
- int
- float
- double
- ...

Ces types sont convertibles les uns en autres, au travers de *cast*, exactement comme en C :
$int\ a\ =\ 4;$
$float\ b\ =\ 1.0;$
$double\ c\ =\ (double)a\ +\ (double)b;$

Cette conversion est "parfaite", à l'arrondi près (le cas échéant). Les scalaires *ressemblent* à leurs équivalents en C, mais Java est plus strict. Si vous manipulez des données de types différents, le *cast* est obligatoire. On ne peut pas additionner un float et un double sans prendre de précautions.

3.1.2 Objets

D'un point de vue syntaxique, les objets sont déclarés exactement comme les scalaires :
$String\ s;$
Faire quelque chose avec s

Là où les choses diffèrent sensiblement, c'est pour l'initialisation et l'utilisation.
Prenons la classe String, qui représente les chaînes de caractères.

Initialisation L'initialisation ressemble un peu au malloc du C. Elle est **obligatoire**.
$String\ s\ =\ new\ String("toto");$

Attention !
$String\ s;$ est juste une déclaration. Le contenu de la variable s n'est rien du tout, même pas $NULL$. Pour mettre un objet a $NULL$, on écrit :
$String\ s\ =\ null;$ (attention aux minuscules)
Si vous oubliez d'initialiser une variable, et que vous vous en servez, le compilateur émettra une erreur.

Utilisation L'élément essentiel du java est le point (".") . Ce caractère sert à accéder au contenu de l'objet, un peu comme dans les structures C.
Pour avoir la longueur de s, on a besoin d'utiliser la méthode $int\ length()$.
$int\ l\ =\ s.length();$

Pour appeler une méthode sur un objet, vous devez **impérativement** :

- avoir déclaré vos variables
- avoir initialisé vos objets
- avoir vérifié que l'objet avait bien cette méthode

3.2 Méthodes / Fonctions

Une méthode se déclare exactement comme en C. On donne le type de retour, le nom de la fonction, et les paramètres avec leurs types entre parenthèses :

```
int foo(String s, int l, ...) {
    // Corps de la fonction
}
```

La différence majeure tient au fait qu'on est dans un univers objet : les méthodes n'existent pas en dehors des classes. Par conséquent, vos définitions de fonctions doivent impérativement être dans des déclarations de classes.

3.3 Classes

Une classe est un *patron*, une forme commune à tous les objets de la famille.
Si on dit quelque chose comme "un carré a quatre sommets, et peut être translaté", cela veut dire que la classe *Carre* contiendra :

- 4 variables pour les sommets (de type *Point*)
- 1 méthode *translate* qui prendra deux arguments de type *float*, pour le X et le Y

Question bête : Que renvoit la méthode translate ?
Si j'ai un carré, et que je le translate, qu'est-ce qu'on crée ? **RIEN**
Donc la méthode renvoit *void*.

Insistance : Oui, mais mon carré est modifié, donc je détruis l'ancien, et j'en crée un nouveau, non ? **NON**. Le carré reste le même, il est juste déplacé. C'est au carré lui même de bouger, en fonction des instructions, tout comme la porte s'ouvre, etc...
La méthode translate se contente de modifier *son propre objet*.

Syntaxe Une classe se déclare toujours de la même manière :

- on dit que c'est une classe (avec le mot *class*)
- on donne son nom
- si besoin est, on dit de quelle classe elle hérite
- si besoin est, on dit quelles interfaces elle implémente

Au final, la forme de la classe est la suivante :

```
class Nom extends ClasseMere implements Interface1, Interface2, ... {
    // Variables et fonctions
}
```

Exemple : Carré On va juste considérer que le carré est un polygone, pour simplifier la déclaration.

```
class Carre extends Polygone {
    // variables
    Point sommet1;
    Point sommet2;
    Point sommet3;
    Point sommet4;

    // methodes
    void translate(float x, float y) {
        // on suppose que les sommets sont également translatables
        sommet1.translate(x,y);
        sommet2.translate(x,y);
        sommet3.translate(x,y);
        sommet4.translate(x,y);
    }
}
```

Exemple d'utilisation :

```
Carre c = new Carre();
c.translate(1.0,2.0);
```

Retour sur le . Le "." permet l'accès aux méthodes, comme on l'a vu plus haut. Mais il donne également accès aux variables, comme dans une structure C. "*c.sommet*1" renvoit le Point sommet1. Cela permet un certain nombre de choses, mais ouvre également la voie à des problèmes d'intégrité : si un autre bout de programme peut accéder aux données, c'est qu'il peut les modifier. S'il peut les modifier, c'est que, potentiellement, il peut modifier le comportement de l'objet...

3.4 Portée

Dans une classe, on distingue les éléments accessibles de l'extérieur de ceux qui restent "privés".
Imaginons un carré, avec ses 4 sommets. On ne veut pas que n'importe quel bout de programme vienne toucher à nos sommets, qui sont (faut-il vraiment le préciser) placés avec grand soin. Ces sommets devraient donc être privés, au contraire de la méthode *translate*, qui elle peut être appelée à tout moment.
Cette portée (appelée *scope* en anglais) peut être précisée à la déclaration de la classe. On distingue 4 portées différentes :

- *public* : tout le monde peut voir et appeler
- *protected* : tous les membres du package (collection de classes, voir plus bas), ainsi que toutes les classes filles peuvent voir et appeler
- *package* : tous les membres de la même collection de classe peuvent voir et appeler (voir les packages plus loin). C'est le mode "par défaut", lorsque rien n'est précisé
- *private* : visible seulement pour la classe. Ses descendantes et ses camarades de package ne peuvent ni voir ni accéder

Le scope se positionne juste devant la déclaration, et définit ainsi la portée de ce qui suit. Par exemple :
private int c;
Il s'applique à **tout** :

- classes
- variables
- méthodes

Ainsi, vous pourrez protéger vos données lorsque la représentation interne d'une classe doit conserver une certaine intégrité. Généralement, on considère que toutes les variables d'une classe doivent être *protected* ou *private*. L'appel direct aux variables internes d'une classe reste (et doit rester) marginal.

3.5 Accesseurs

Si on protège nos variables, mais qu'on souhaiterait laisser un accès en lecture ou en écriture, il faut écrire des méthodes dont le rôle est d'accéder aux variables d'instance.
Ces méthodes particulières sont appelées accesseurs.

Pour reprendre notre exemple du carré, nos méthodes de lecture (*getSommet*1() par exemple) doivent être publiques et ne font rien d'autre que fournir le sommet en retour :

```
public Point getSommet1() {
    return sommet1;
}
```

Par contre, pour ce qui concerne la *modification* de nos variables, il faut faire attention ! Un carré dont un sommet est déplacé ne fait généralement pas un carré... L'accesseur *setSommet*1(*Sommet s*) doit donc vérifier que le point est bien à un endroit "acceptable" :

```
public void setSommet1(Point s) {
    if(<<test que le point est bien>>) {
        sommet1 = s;
    } else {
        // ne rien faire, ou faire une erreur
    }
}
```

Les accesseurs sont donc les “gardiens” des variables. Ca n’est pas obligatoire, mais la plupart du temps, on veut être sûr que le programme ne fait pas n’importe quoi à nos données...

3.6 Constructeurs

On a vu comment *manipuler* nos objets, comment les *déclarer*, etc... mais il reste un point d’ombre : à quoi correspond le “new” ?
Dans ce qui précède, le *new* avait l’air de correspondre à peu près à un *malloc* en C. Ceci n’est pas remis en cause : le *new* crée effectivement l’espace mémoire nécessaire. Mais ça n’est pas tout !

3.6.1 Constructeur par défaut

Le constructeur par défaut existe d’office : il se charge de créer l’espace mémoire, et d’allouer les ressources nécéssaires pour les variables. Il ne fait rien d’autre, et surtout pas initialiser correctement vos variables.
La méthode de base est de faire un new, puis d’appeler les accesseurs :

```
Carre c = new Carre();
Point p1, p2, p3, p4;
// initialiser les points qui vont bien

c.setSommet1(p1);
c.setSommet2(p2);
c.setSommet3(p3);
c.setSommet4(p4);
```

Comme on peut le remarquer, cette méthode est efficace, bien que manquant cruellement d’élégance.

3.6.2 Surcharge du constructeur par défaut

La première étape est de dire que certaines variables ont *forcément* une certaine valeur à la création. Par exemple, quoi qu’il arrive, lorsqu’une voiture est fabriquée, elle n’a pas de passagers. Ou bien, lorsqu’on crée un stylo, celui ci est plein d’encre.
Prenons la classe Stylo :

```
class Stylo {
    private float encre;
}
```

Cette encre est un $float$, compris entre 1.0 (plein) et 0.0 (vide). Lorsque je fais un “*new Stylo*()”, je m’attends a ce que *encre* vale 1.0.
On doit donc indiquer à la classe comment se comporter lors d’un *new*.

Syntaxe La syntaxe d’un constructeur est la suivante :
$< porte > Nom(< params >)$, où Nom est le nom de la classe, et $< params >$ peut être vide.

Mon constructeur de Stylo

```
public Stylo() {
    encre = 1.0;
}
```

Ce constructeur est appelé lorsqu’on fait *new Stylo*(), sans paramètre.

3.6.3 Constructeurs génériques

Il arrive que l’on ait besoin de passer des paramètres à la création de l’objet. On l’a vu plus haut pour le carré, faire un *new* sans option nous donne un carré inutile, puisque non initialisé.
On peut donc également créer des constructeurs avec des paramètres. Ces paramètres définiront le comportement *initial* de notre objet. Rien n’empêche de tout changer à mesure que le programme avance.

Pour notre carré, on aimerait lui passer ses 4 points au moment du *new* :
$Carre\ c\ =\ new\ Carre(p1, p2, p3, p4)$;

On a donc besoin d'un constructeur qui prenne 4 Points en paramètres :

```
public Carre(Point s1, Point s2, Point s3, Point s4) {
    sommet1 = s1;
    sommet2 = s2;
    sommet3 = s3;
    sommet4 = s4;
}
```

Vous pouvez créer autant de constructeurs que vous voulez, avec toute la variété de types imaginables. Ici, il pourrait être intelligent d'avoir un constructeur avec les X et les Y de chaque point, par exemple.

3.6.4 Conventions

Généralement, pour chaque classe, on compte trois constructeurs "indispensables" :
- le constructeur sans paramètre, dit "par défaut", qui donne aux variables une valeur arbitraire, mais sensée. Par exemple, un point par défaut peut être en $(0; 0)$, un stylo a plein d'encre, etc...
- le constructeur avec tous les paramètres, dit "complet", qui demande la valeur de toutes ses variables. On l'a vu pour le carré, ça s'applique de la même manière à toutes les classes
- le constructeur "par copie". Ce constructeur "duplique" un objet existant. Il est très utile lorsque les données ne peuvent pas être partagées

3.7 Packages

Les packages correspondent aux modules, ou aux librairies. On verra plus loin la façon de les utiliser, mais l'esprit est le suivant :
Lors de la déclaration d'une classe, on "l'affecte" à un package.
Dans chaque package, le nom des classes doit être unique. Mais rien n'empêche d'avoir une classe *Point* dans le package *toto* **ET** dans le package *tata*
Par défaut, une classe appartient à un package "anonyme".

3.8 Quelques conventions

Le style en Java est important. Il est très respecté, en partie parce que l'ensemble des classes existantes le respecte déjà. Nous attendons de vous une certaine rigueur par rapport à ces conventions.

3.8.1 Fichiers et classes

Cette règle est "forcée" par le compilateur, vous devrez donc la respecter :
Chaque fichier source (*.java*) contient **une et une seule** classe. La classe *Point* est déclarée dans un fichier "Point.java", la classe *Carre* dans un fichier "Carre.java", etc...
A cause de cet état de fait, évitez de donner des noms compliqués ou accentués à vos classes : "Carré" devient Carre, "$\Omega\beta\kappa Truc$" devrait devenir quelque chose comme OBKTruc.

3.8.2 Noms

Les noms suivent quelques règles simples :
- pas d'espace, pas de _, pas de tirets. On colle tous les mots d'un nom ensemble.
- les mots commencent par une majuscule au sein d'un nom
- les variables commencent par une minuscule, les méthodes aussi, et les classes par une majuscule

Ainsi, on aura "*class MaSuperClasse*", "*int maSuperVariable*", "*monSuperObjet.maSuperMethode*()".

3.9 Autres mots clefs

3.9.1 static

C'est le mot clef "spécial" que vous rencontrerez le plus souvent en Java. Et pour cause, il est présent au moins une fois dans chaque programme... *forcément* !
On a vu dans ce qui précède que les variables et les méthodes n'existent que dans une classe. Aucune fonction ne peut être définie en dehors de cette structure. Toutefois, il existe deux types de méthodes :
- celles qui marchent de le même façon, quelle que soit l'instance qu'on regarde
- celles qui marchent avec une instance en particulier

Pour prendre un exemple simple, la translation ne peut s'appliquer qu'à un objet en particulier, voire à un groupe d'objets. En aucun cas ce n'est une fonction universelle.
Par contre, certaines fonctions ont une vocation à être utilisées en dehors de tout contexte. A quel objet demander le nombre de roues d'un vélo ? Chaque vélo en a forcément deux, ca n'est pas propre à un vélo en particulier. Pourquoi faire un *new* $Velo()$, juste pour lui demander combien de roues il a ?
Ces fonctions "générales" sont appelées méthodes de classe. Elles s'appellent en faisant $Velo.nombreDeRoues()$, si Velo est une classe.
Ces méthodes se déclarent en rajoutant *static* entre le scope et le type de retour. Ici :

```
class Velo {
    public static int nombreDeRoues() {
        return 2;
    }
}
```

3.9.2 final

Ce mot placé avant le type dans la déclaration d'une variable en fait une constante. Rien ne pourra modifier sa valeur.

```
final static int CONSTANTE = 0;
```

Si vous voulez empêcher la modification d'un paramètre dans une fonction, vous pouvez le "verrouiller" à l'aide de final.

```
public void foo(final Velo v);
```

Ce mot clef est très peu utilisé. Dans la plupart des cas, vous n'en n'aurez pas besoin.

3.9.3 abstract

Lorsqu'on déclare une classe, on peut si on le désire laisser la définition d'une fonction "vide". Contrairement au C/C++, il n'y a pas de headers en Java. Si une méthode n'est pas définie complètement dans la classe, elle ne pourra être définie que dans les classes filles.
Une classe qui contient au moins une méthode "vide" est dite "abstraite". Pourquoi ?
Une classe qui n'a pas toutes ses méthodes ne peut pas être initialisée : on ne peut pas faire un *new* avec. Elle représente donc quelque chose qui existe (puisqu'elle est déclarée), mais qui n'a aucune réalité "physique" (aucune instance). Par exemple :

```
abstract class ClasseAbstraite {
   public int foo1() {
       return 1;
   }

   public abstract int foo2();
}
```

Il est assez rare d'utiliser les classes abstraites en Java. On leur préfère les interfaces.

3.9.4 interface

Une interface est une classe complètement abstraite : aucune de ses méthodes n'est définie. Si vous n'avez pas retenu ce qu'était une interface, voyez la section "Classes abstraites" au début de ce cours.
En Java une interface se déclare comme une classe :

```
interface MonInterface {
    public int foo();
    public void bar();
}
```

Si une classe respecte une interface, on dit qu'elle l'implémente. Si une classe implémente une interface elle **doit** définir **toutes** les méthodes de l'interface.

```
public class MaClasse implements MonInterface {
    public int foo() {
        // implementer la méthode
    }

    public void bar() {
        // idem
    }
}
```

4 Point d'entrée

4.1 En assembleur

Généralement, le label d'entrée de programme en assembleur est _start
C'est la première instruction chargée, le premier pas.

4.2 En C

En C, la première fonction chargée est
*int main(void) int main(int argc, char** argv)*
Il ne peut y en avoir qu'une seule dans un programme, c'est la fonction qui prépare tout et commence la séquence d'instructions. Lors de la compilation, elle reçoit le label _start.

4.3 En Java

En Java, la méthode main existe, et est également le point d'entrée.
Toutefois, il y a un problème conceptuel : sachant qu'une fonction ne peut pas exister en dehors d'une classe, où la ranger ? Quelle classe peut l'héberger ? Quelle classe **doit** l'héberger ?

La réponse est simple : chaque classe peut contenir un point d'entrée. Et comme en C, la méthode main a une forme obligatoire.

4.3.1 Syntaxe

```
public static void main(String args[])
```

Les différents mots utilisés devraient être clairs, mais une petite explication ne fait jamais de mal :
- *public* : Accessible par tous (et surtout par la machine virtuelle)
- *static* : Indépendant de l'instance dans laquelle on se trouve (pas besoin de faire un new)
- *void* : Pas de return dans le main en Java. On utilise $System.exit(intstatus)$
- *String* : String est un objet qui a des méthodes, pas un tableau de caractères
- *[]* : Tableau de chaînes. Attention, le tableau est une entité ambigüe en Java

4.3.2 Classe Principale

Puisque chaque classe contient potentiellement un *main*, comment identifie-t'on la *bonne* ?
La réponse de Java à cette question est toute aussi simple : on décide lors de l'exécution laquelle parmi toutes les classes est la première. Par conséquent, on dit que c'est un problème de *runtime*. La machine virtuelle, une fois lancée, cherche dans ses paramètres quelle classe est considérée comme principale, avant de lancer le *main* de cette dernière.

Sous Linux, on tape
$ java *classePrincipale*

Dans Eclipse, comme dans la plupart des environnements de développement, on spécifie la classe principale manuellement, avant de lancer l'exécution.

5 Système d'exceptions

Que se passe t'il si on a une erreur ? Comment se comporter dans le cas où on est en dehors du domaine d'application (racine carrée d'un nombre négatif, par exemple) ? Comment faire si au cours de l'exécution, un élément crucial manque ?

En C, la méthode traditionnelle est de renvoyer un code d'erreur (un int, généralement). C'est le cas de la fonction *main*, par exemple, qui renvoit 0 si tout s'est bien passé, et n'importe quoi d'autre dans le cas contraire. En Java, même si on peut aussi avoir ce mécanisme, on lui préfère le système des exceptions.

5.1 Principe

Une fonction a normalement un domaine de départ et un domaine d'arrivée. Elle a également un contexte d'exécution. Mais elle a également des *types* de départ et d'arrivée.
Par exemple, une fonction mathématique comme racine carrée (*sqrt* en C) prend des doubles en paramètres. Un double va de quelques-milliards à moins-quelques-millards. Mais la fonction racine carrée ne marche qu'avec des nombres positifs.
Avec ce petit exemple, on remarque que "erreur" est un mot trop restrictif : faire une racine carrée d'un nombre négatif n'est pas une erreur... c'est juste *non-défini*.

Le principe est donc d'écrire une fonction qui fonctionne "normalement", et d'identifier les cas où la fonction ne s'appliquera pas.
On distingue alors deux grandes catégories d'exceptions :

- les "accidents" : divisions par zéro, index en dehors du tableau, pointeurs nuls, ...
- les "zones non-définies" : valeur en dehors d'un domaine, clic sur autre chose qu'un bouton, etc...

Dans le cas de *sqrt*, le pseudo-code associé sera donc :

```
float sqrt(float v) exception EnDehors {
    if(v < 0)
        lever exception EnDehors
    else
         return calcul_sqrt(v)
}
```

5.2 Vocabulaire

Les exceptions, quel que soit le langage utilisé, ont un vocabulaire particulier, et quelques particularités de base :
Tout d'abord, une exception est un objet. On est dans un univers objet, donc tout est objet. Les exceptions n'y font pas défaut. Une exception a donc des variables et des méthodes.
Dès lors, on peut fabriquer des exceptions correspondant à nos besoins, qui contiendront (par exemple), la raison de l'exception, le contexte, des valeurs arbitraires, etc...

Lorsqu'une erreur survient, ou qu'on est dans un cas exceptionnel, on dit qu'on **lève** une exception. En anglais, on appelle ça *throw* (on **lance** une exception).

Si on souhaite traiter une exception, on dit qu'on **l'attrape**, ou qu'on la **rattrape**. En anglais, *catch*.

Throw et *catch*, on utilise toujours des métaphores "physiques".

5.3 Pseudo-code et prémisses de syntaxe

Comment gère t'on les exceptions ?

5.3.1 Lever une exception

Lever une exception est simple, et ressemble tout naturellement à l'expression utilisée :
On crée une exception avec un *new*
On lève l'exception créee

```
Exception e = new Exception();
throw e;
```

5.3.2 Rattraper une exception

Pour rattraper une exception, on doit tout d'abord se poser deux questions fondamentales :

- Où peut-on avoir une exception ?
- Quelle exception peut-on avoir ?

Quelques exemples :

- J'ai une fonction mathématique qui n'est pas définie pour certaines valeurs, l'exception porte sur le domaine d'application
- Je joue avec un tableau, l'exception porte sur l'index recherché (Out of bounds)
- Je tente de lire dans un ficher, les exceptions sont variées : le fichier n'existe pas, je n'ai pas le droit de lire dedans, le contenu est pourri, ou ne correspond pas à ce que j'essaie de lire, etc...

On agit donc en deux temps :

```
essayer de faire {
    quelque chose qui pourrait faire une exception
} si on a une exception comme (UnTypeDException) {
    traiter l'erreur
} si on a une exception comme (UnAutreTypeDException) {
    traiter l'erreur
} etc...
```

5.4 Syntaxe

Comme on l'a vu dans le vocabulaire, on lance une exception. Cela se traduit de deux manières :

- le code de la méthode lève une exception
- le *prototype* de la méthode indique qu'on *peut* avoir une exception

Le deuxième point est crucial : il sert à indiquer aux développeurs, et au compilateur, quels sont les fonctions susceptibles de provoquer des exceptions. Il **force** le programmeur à agir en conséquence.

```
public int func(int a) throws DomainException {
    if(a < 0) {
        DomainException e = new DomainException();
        throw e;
        // fin de l'exécution, throw se comporte comme return
    } else {
        // faire quelque chose avec a
        int b = a;
        return b;
    }
}
```

Qu'apprend t'on ici ? que $func$ peut lever l'exception *DomainException*, et qu'il faut donc s'en occuper.

Pour rattraper une exception, on la *catch*. Comme on a vu précédemment, on doit d'abord identifier le bout de code qui peut avoir un comportement exceptionnel, avant de traiter les exceptions.

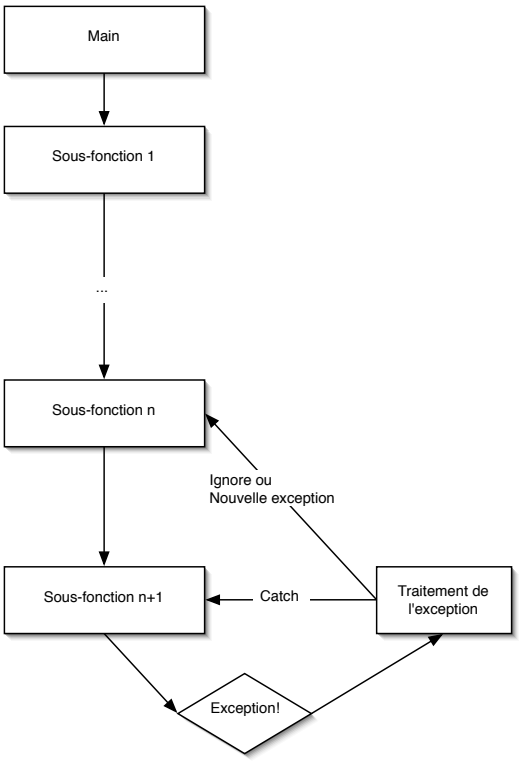
```
try {
    int c = func(10);
} catch (DomainException e) {
```

```
    // si on a une exception, on fait quelque chose avec e
    e.afficher();
}
```

5.5 Pile d'appel

La levée d'une exception *interromp* l'exécution (comme un return). Le reste n'est pas exécuté. La fonction appelante rattrape donc l'exception, et la traite de la façon qui lui convient :

- Elle traite l'exception (en changeant les paramètres, en changeant de branche, etc...). C'est le comportement typique.
- Elle lève une autre exception. Si on se trouve dans un contexte d'application un peu complexe, et que l'exception levée n'a pas d'intérêt pour les niveaux supérieurs en tant que telle, on peut choisir de créer une nouvelle exception et de la lancer à son tour.
  Attention, dans ce cas, le prototype de la méthode *doit* signaler la nouvelle exception !
- Elle ignore l'exception. Son exécution est alors interrompue, et on remonte d'un niveau dans la pile d'appel.
  Attention, dans ce cas, le prototype de la méthode *doit* signaler l'exception !



Si une exception remonte la pile d'appel jusqu'au *main* et qu'elle n'y est pas traitée, le programme sort.

5.6 Classes d'exceptions

La classe mère de toutes les exceptions s'appelle (tout naturellement) *Exception*. Cette classe contient toutes les informations "de base" : un message (avec la méthode $getMessage()$), la pile d'appel (avec la méthode $getStackTrace()$), etc...

Dès lors, fabriquer une nouvelle exception, qui contiendrait nos informations à nous, consiste tout simplement à sous-classer Exception.
Prenons un exemple simple : nous voulons une exception qui nous donne l'heure à laquelle l'exception a été levée.

```
public class DatedException extends Exception {
    private Date d;
```

```
    public DatedException(Date errorDate) {
        d = errorDate;
    }

    public getDate() {
        return d;
    }
}
```

Une autre manière de fabriquer une exception est d'implémenter l'interface *Throwable*. La syntaxe reste la même.

```
public class DatedException implements Throwable {
    private Date d;

    public DatedException(Date errorDate) {
        d = errorDate;
    }

    public getDate() {
        return d;
    }
}
```

5.7 Usages

Quel est l'intérêt de ce système d'exceptions ? Le code de retour marche très bien !
Oui, mais les informations recueillies par un système de code de retour sont assez limitées : même si on pouvait coder tout un tas d'informations dans un int, certaines choses comme la pile d'appel, ou le message d'erreur, seraient fortement tronqués.
L'exception est avant tout un *message* passé à la fonction appelante. La voir comme une simple erreur est une réduction : comme son nom l'indique, une exception dénote un comportement *exceptionnel*.
On peut voir une exception comme un évènement un peu particulier, qui aurait la capacité d'interrompre n'importe quelle exécution. Prenons un petit exemple simple, un jeu.

Imaginons un personnage qui se déplace dans l'espace. le joueur dit "aller à droite", le personnage avance vers la droite sans s'arrêter. Le comportement normal du personnage est de se déplacer, le cas exceptionnel est de rencontrer un obstacle. Est-ce une erreur ? Non, juste un évènement exceptionnel.

Souvent, les exceptions dénotent un problème (accès interdit, réseau déconnecté, domaine faux, etc...). Mais la capacité de ce système à véhiculer des messages de bas en haut de la pile d'appel en font un mécanisme puissant pour ce qui concerne la programmation événementielle.