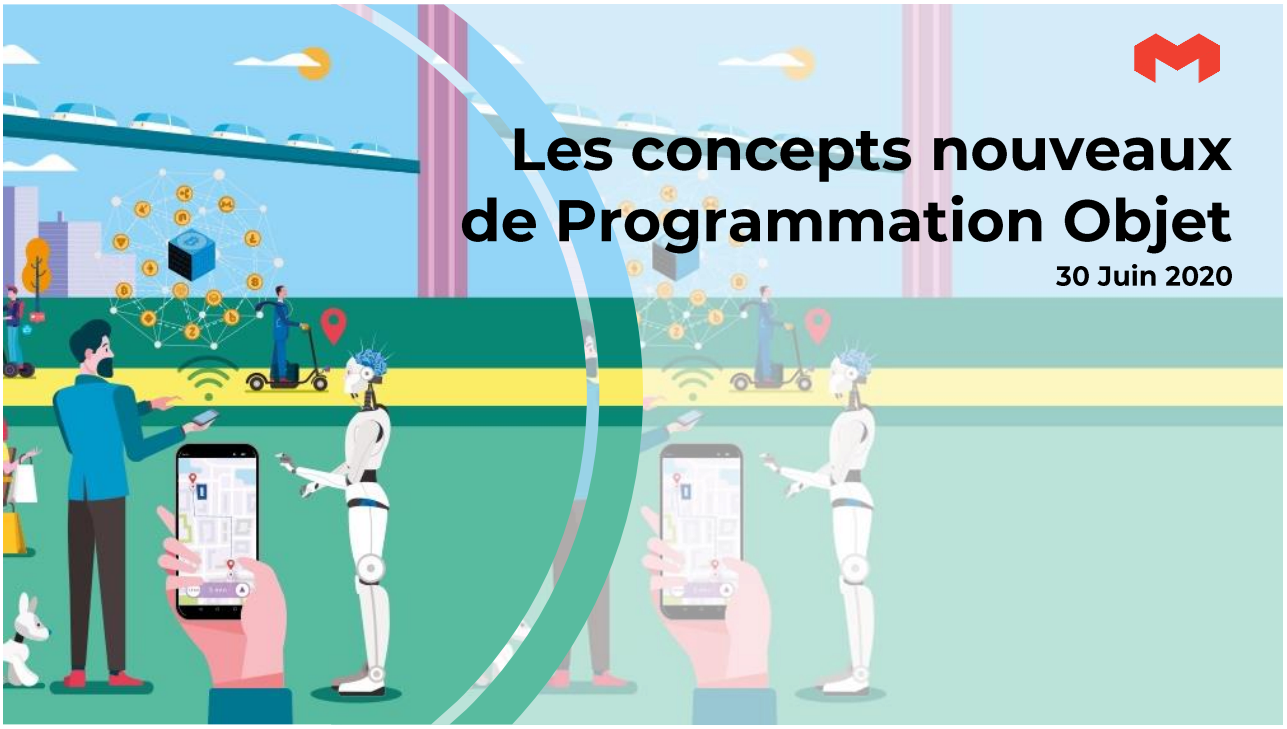
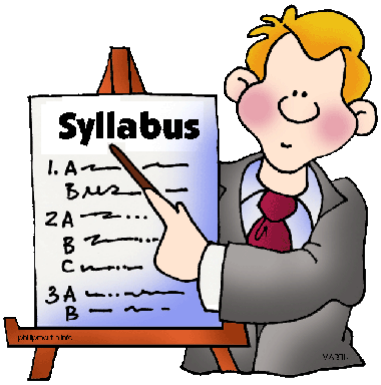


Les concepts nouveaux de Programmation Objet

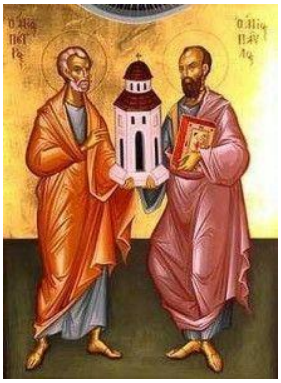
30 Juin 2020

Sommaire

Les concepts nouveaux de programmation objet

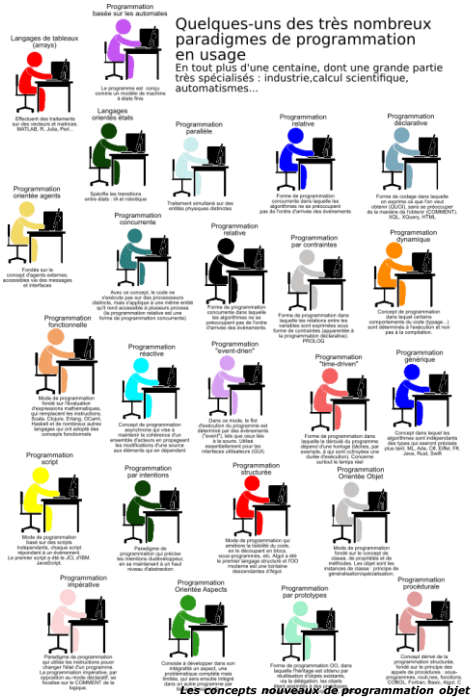


Le grand schisme de la programmation



- Désormais cinq familles de développeurs (on peut en imaginer d'autres)
  - Langages proches des machines : assembleurs, C…
  - Langages script et déclaratifs (SQL, HTML)
  - Langages objet et fonctionnels
  - Langages "low code"
  - Langages proches du métier (Cobol, L4G)
- Ce ne sont pas mêmes populations
- L'avenir va conforter la séparation entre des développeurs très professionnels et des développeurs occasionnels
- Séparation entre les prestataires et les usagers : quel intérêt y aura-t-il à conserver des développeurs chez soi, autres que pour l'UI ?

- La conception d'un compilateur est une science, appliquer sa syntaxe est une technique.
- Les bonnes pratiques d'un langage se réfèrent à l'expérience. On sait ou on ne sait pas.
- Le risque est grand de se réfugier derrière l'expérience d'un langage et de la confondre avec la connaissance.
- Il est beaucoup plus facile de se spécialiser sur un langage que d'essayer de prendre du recul (et de la hauteur) et de le repositionner dans son contexte
- Sujet très sensible, qui divise la communauté…



Les principaux paradigmes de programmation

La programmation réactive

Un exemple de ce qui va nous arriver

- Il y a deux natures de systèmes, selon que les éléments qui les constituent sont indépendants ou non.
- Ex d'une affectation a = b + c.
- **Les fondements**
  - Ils s'intéressent aux variables évolutives dans le temps. A tout moment, elles doivent être justes et refléter la réalité du processus simulé.
  - Quand une variable change de valeur, il faut qu'elle émette un message ou génère un évènement et doit informer les variables dépendantes pour les réactualiser.
  - Un système réactif suit les dépendances entre les variables, directes ou indirectes et si une nouvelle dépendance apparaît, il doit l'intégrer dans le processus de mise à jour automatique. Il construira pour cela un graphe des dépendances.
  - Il doit être capable de propager les dépendances. Avec essentiellement deux méthodes, soit c'est la variable qui a été modifiée, qui avertit les autres de son changement (« push »), soit chacune des variables dépendantes se charge d'interroger la variable origine, pour savoir si elle a changé ou non (pull). Il peut aussi y avoir un mode mixte, qui va exploiter les deux pratiques.
- Ex : Red, Emfrp, Rcket…

C'est quoi un langage objet ?

- Nombreuses définitions, mais il ne faut pas faire preuve d'ayatolisme…
- On peut simplifier…
- La POO est un formalisme basé sur le concept de classe, qui correspond à un traitement délimité, une description statique de ce traitement, instanciée à l'exécution sous forme d'objets
- On lui associe généralement trois caractéristiques essentielles :
- **L'encapsulage**: on communique avec les objets exclusivement par des méthodes, publiques, privées (entre autres), sans que l'on ait besoin de connaître la cinématique interne des traitements
- **L'héritage** : une classe peut hériter d'une autre et récupère par ce biais, les méthodes et propriétés de la classe mère, point sur lequel les théoriciens s'accrochent depuis 50 ans… sans jamais se mettre d'accord
- **Le polymorphisme** : correspond au fait qu'une classe peut être conçue de telle manière qu'elle "prend vie" différemment selon les contextes d'usage, on parle ici de "comportement générique" ou d'interfaces
- Historiquement Simula est considéré comme le premier langage OO
- On ne perdra rien à se souvenir de la différence entre agrégation et composition :
  - La composition est une relation de type **"fait partie de"** : si un objet B fait partie d'un objet A alors B ne peut pas exister sans A (appartenance forte : l'agrégé ne peut participer qu'à un seul agrégat).
  - Losange plein côté agrégé).
  - L'agrégation est une relation de type **"a un"** : si un objet A a un objet B alors B peut vivre sans A (appartenance faible : l'agrégé peut participer à d'autres agrégats). Losange vide côté agrégat.

Des concepts pas toujours bien compris

Formation objet : on commet souvent l'erreur d'apprendre UN langage, alors qu'il faudrait passer par une approche générique, dans laquelle on expliquera les concepts…

Les grandes évolutions de l'objet

Un bouillonnement permanent

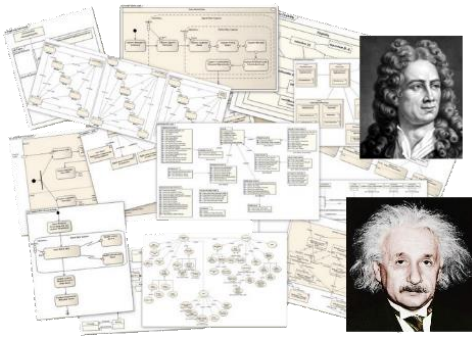
- Bouillonnement permanent : mais pas de remise en cause fondamentale, sauf l'apport du fonctionnel, mais des évolutions structurelles, sémantiques et d'usage
- Toujours la même découpe entre impératif et objet
- Mais aussi entre PHP ou JavaScript et l'objet…
- Une tendance forte, le transfert de responsabilité du langage vers son environnement (API et serveurs d'applications)
- L'apport du fonctionnel : composabilité, expressivité (fonctions d'ordre supérieur), fonctions pures (sans effet de bord), curryfication, transparence référentielle, immutabilité
- Très gros progrès dans le traitement de la concurrence
- Un objectif commun : la simplicité d'écriture et la compréhension
- Eloignement vis-à-vis des contraintes matérielles et logicielles des plates-formes
- Il devient difficile de distinguer l'appartenance des langages à une "chapelle"
- Extension très importante des structures de données
- Catalogue de bonnes techniques plus ou moins appliquées dans les langages : Swift, Go…
- Ce que l'on trouve dans l'un a des chances d'apparaître dans les autres : il suffit d'être patient
- La spécialisation des développeurs est la règle, qui ont transféré leurs préoccupations du langage vers les API… ce qui se comprend

L'esthétisme du code

- Une interrogation qui prend de plus en plus d'importance : sujet très vaste
- Un code "beau à regarder" est plus facile à comprendre et fonctionnera mieux en production et nécessitera moins d'efforts correctifs
- Fait partie de la qualité : le code doit être facile à lire, comprendre et modifier
- La simplicité algorithmique et d'écriture sont un complément à l'esthétisme du code
- Eternelle opposition entre les tenants de la clarté du codage et sa complexité qui rassure…
- L'indentation avec différentes méthodes suggérées ou imposées (Python)
  - Méthode RT (créateurs d'Unix) : l'accolade d'une fonction sur la ligne suivante
  - GNU et de nombreuses autres : il faut faire un choix
- Quelques principes d'écriture claire
  - Principe de responsabilité unique : un bloc de code ne fait qu'une seule chose (et il la fait bien) : penser le code en termes de responsabilités
  - Séparation des "commandes" (effectuent une action) et "queries" (répondent à une question), évite les effets de bords
- Penser à la cohésion des éléments d'un programme
  - Plutôt que répéter le code
  - Attention à la dépendance

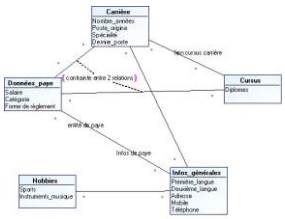
"Comment convaincre les gens que la simplicité et la clarté du code, ce que les mathématiciens appellent l'élégance, ne sont pas un luxe, mais une nécessité cruciale qui fait la différence entre succès et dysfonctionnements".
Edsger W. Dijnstra

La modélisation conceptuelle des données



"Ce qui se conçoit bien s'énonce clairement, et les mots pour le dire arrivent aisément"
(Boileau)

"If you can't explain it simple, you don't understand it well enough"
(Albert Einstein)

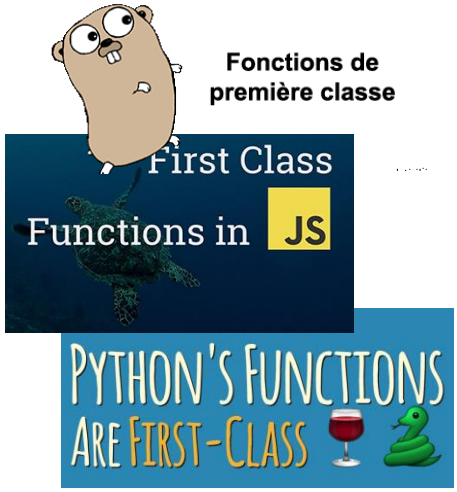


- Une modélisation bien conçue est souvent la meilleure manière d'aborder une problématique de données
- Les modeleurs graphiques traitent deux types de besoins :
  - La modélisation des données proprement dites : le modèle conceptuel
  - La modélisation des bases de données, qui doivent être générées automatiquement à partir de scripts créés par l'outil de modélisation
- L'idée de base est que toute modification sur les données doit entraîner un nouveau modèle conceptuel, qui générera un nouveau schéma logique lié à une base de données déterminée et les requêtes SQL de création de tables correspondantes
- Dans la pratique il en est très autrement…
  - Les concepteurs « ne font pas l'effort de conceptualisation », ce qui se traduit par des incohérences de données

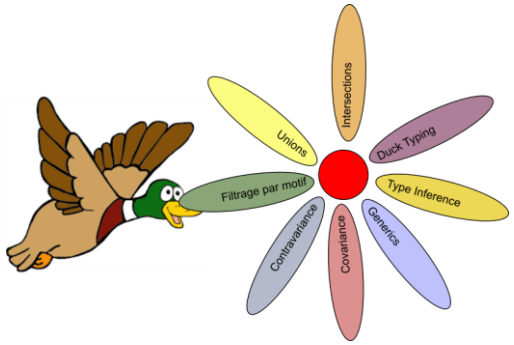
Les grandes évolutions : données et codage

Les fonctions de première classe

- Ex de la programmation fonctionnelle
- Une fonction est de première classe, quand on peut la traiter comme n'importe quelle variable : un entier, une chaîne de caractères… On dit qu'elle a le même statut.
- Elle peut être transmise en tant qu'argument à d'autres fonctions, peut être retournée par une autre fonction et affectée en tant que valeur à une variable.
- On peut donc agir dessus de la même manière que sur une variable classique :
  - On peut la nommer, la typer et l'affecter :
    - x := sin;
  - On peut la définir et la créer à la demande :
    - x := (function x-> x+1);
  - On peut la passer en argument à une autre fonction :
    - f(sin);
  - Elle peut être le résultat d'une fonction :
    - (f(5))(7);
  - Elle peut être stockée dans une structure de données quelconque :
    - array := {log, sin, cos, tan};

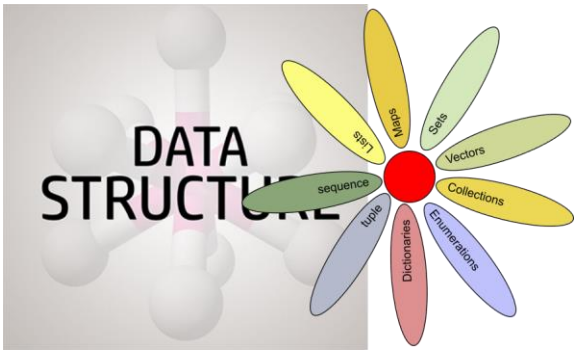


Le typage des données, fonctions et méthodes



- **Le typage des données**
- En matière de typage, le mode statique est considéré comme la meilleure protection sécuritaire, mais on peut être agacé par l'étroitesse du concept et le transtypage explicite, qui lui est généralement associé.
- On peut le traiter avec plus de hauteur, voire le déléguer au compilateur ou le considérer comme un paramètre. Avec l'avantage de proposer un code plus aéré, plus « esthétique », mais aussi moins sûr (peut-être…)
- **Union et intersection de types** : dans Ceylon, on trouve plusieurs dispositions qui touchent également au typage des données, les unions et intersections de types.
- L'union permet de manipuler une donnée en sachant seulement que son type fait partie d'une liste, mais sans savoir exactement duquel il s'agit.
- L'intersection est plus restrictive, car la référence doit satisfaire à tous les types mentionnés dans la liste. C'est le ET contre le OU.
- On pourrait ajouter aux spécificités de Ceylon, les types covariance et contravariance, qui ne seront pas sans rappeler quelques souvenirs aux mathématiciens qui ont fait un peu de calcul tensoriel…

Les structures de données



- Il existe différentes manières de classer ces structures, selon qu'elles sont finies, indexées (tableaux, vecteurs) ou récursives : un élément au moins de la structure comporte un pointeur vers une autre structure du même type (listes, graphes)
- On peut aussi les distinguer selon qu'elles sont séquentielles, les éléments étant rangés d'une certaine manière, comme les listes et les tableaux, qu'il s'agit de tables de symboles, qui sont en fait des tableaux associatifs (maps), pour ranger les contenus en fonction d'une clé (table de hashage…), voire d'autres formes de collections, comme les ensembles, les bags ou les graphes.

Dictionnaires Swift et Python
Un dictionnaire au sens Python et Swift, est une association d'éléments de type clés : valeurs, où la clé sert d'index pour retrouver une valeur. L'avantage étant que l'on peut retrouver très rapidement une correspondance, même (et surtout) avec des dictionnaires volumineux.
Enumérations Swift
Une énumération au sens Swift, mais aussi Ada, C#, C++ ou Java (entre autres), est une entité désignée par un nom unique, qui comporte un certain nombre de valeurs, qui selon les implémentations du concept, seront toutes du même type ou pas.

L'inférence de type

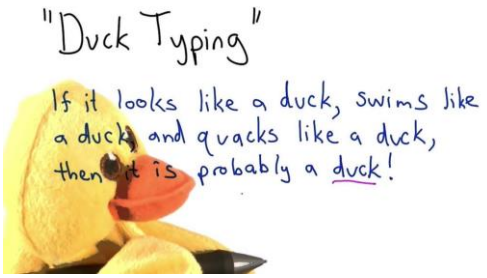
- L'inférence de type, qui existe depuis longtemps, est une disposition qui permet à un compilateur de déterminer le type d'une variable ou d'une expression, sans se référer à un typage statique. On dit qu'il le devine, qu'il « infère ».
- Il existe un grand nombre de techniques, selon les langages, qui aboutissent au même résultat : Prolog, C#, F#, Java, Swift, Go, Scala, Groovy, C++.
- L'inférence de type que l'on ne confondra pas avec le typage dynamique, qui lui se dénoue à l'exécution, est généralement très apprécié des développeurs qui voient en lui un très bon moyen pour rendre le code plus robuste.
- Exemple en C# :

```
var name = "Lance Armstrong" ;
var age = 39 ;
var isAided = True ;
Type nametype = name.GetType();
Type nametype = age.GetType();
Type nametype = isAided.GetType();
```

- Nulle-part, ce code n'intègre le type des variables. C'est le compilateur qui le déduit du contexte. Quand c'est possible…

Le "duck-typing"

- Le principe du « duck typing » (Ruby, Python, Java, JavaScript, C#…), fait référence à la phrase de James Riley : « si je vois un oiseau qui vole comme un canard, cancane comme un canard et nage comme un canard, j'appelle cet oiseau un canard ».
- Transposé en programmation, le « duck typing » s'applique à l'ensemble des méthodes et attributs qui caractérisent un type, le compilateur détectant le type à partir de ces informations, sans précisions explicites.
- Ainsi un objet issu d'une classe A pourra être considéré comme un objet issu d'une classe B, sans qu'il y ait de relation d'héritage entre les 2, à la condition que les attributs et méthodes de B soient aussi présents dans A.
- Le « duck typing » se rapproche du concept de typage structurel d'OCaml, Haskell ou ML. Dans lesquels 2 objets sont dits compatibles si leurs types ont une structure identique, constat qui ne se fonde pas sur des déclarations explicites.
- Le pseudo-code suivant illustre le principe.



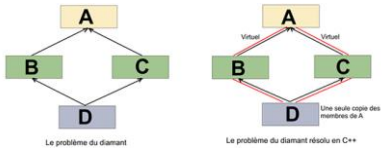

```
function compute(a, b, c) => return (a+b)*c
a = compute(1, 2, 3)
b = compute("apple", "peach", 3)
print to_string a
print to_string b
```

- A l'exécution on obtiendra :
- **9** et **"apple peach, apple peach, apple peach"**
- Le compilateur s'est simplement assuré que les objets a et b disposaient bien des méthodes + et *.

Héritage simple ou multiple

Une vielle affaire

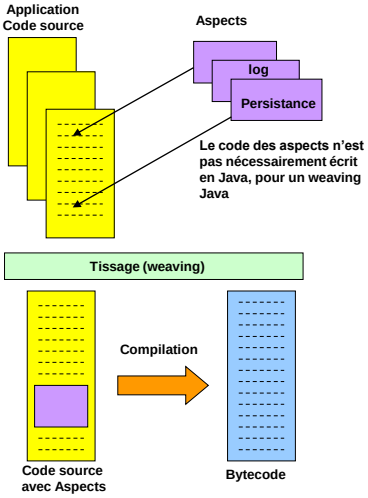
- Pour éviter le diamant de la mort
- **C++ : concept de classe virtuelle**
  - Opérateur de résolution de portée : lourd et peu pratique
  - Ex : D hérite de B et C, mais pas de A, si celle-ci est déclarée classe virtuelle
- **Java** : depuis le JDK 8, une interface peut hériter de plusieurs interfaces, dotée de mêmes méthodes "default" (concrètes). Il doit alors choisir explicitement l'interface dont il "récupèrera" la méthode "default". Si une classe hérite d'une interface et d'une classe, avec la même méthode, c'est la classe qui l'emporte.
- **Scala** : pas d'héritage multiple et pas d'interfaces. Mais une classe peut hériter d'une autre classe et d'autant de "traits" qu'on le veut (les traits peuvent comporter des propriétés et méthodes concrètes). En cas de conflit dans les "traits", il faut choisir avec "extends", les autres traits étant identifés par "with".
- **Golang** : pas un langage OO, mais des structures et des interfaces, les objets (pas au sens OO) étant étendus par composition et non pas par héritage (ça se ressemble). Une structure peut être obtenue par composition de plusieurs autres structures, d'où problème proche de celui du diamant.
- **Mixin** : sorte d'héritage multiple restreint, une classe pouvant hériter (implémenter) de plusieurs interfaces, à condition qu'il n'y ait pas d'ambiguïté sur les noms, un mixin que l'on retrouvera sous des formes approchées dans Dart, Ruby ou Ceylon...PHP et JavaScript.



Le diamant de la mort

La Programmation par Aspects

- AOP : Aspect Oriented Programming
- Principe qui permet de séparer le code d'une application d'un certain nombre de traitements, souvent techniques et transversaux, qui peuvent être programmés indépendamment et insérés dans le code de l'application
- Pour intégrer par tissage (weaving) des aspects entiers de programmation, qui auront été écrits une fois pour toutes : commit de base de données, gestion de la persistance objets…
- AspectJ a été le premier AOP (Java)
- Un aspect AspectJ est un module logiciel qui définit un comportement et se diffuse en différents points de l'application : il est composé de trois éléments :
  - Un nom unique d'identification
  - Une expression de coupe (« pointcut »), qui permet d'identifier les points d'applications (ou points de jointure, « join points ») de cet aspect dans les classes de l'application
  - Le point de jointure est un moment au cours de l'exécution de l'application où un certain nombre de propriétés sont vérifiées
  - Un ensemble de prescriptions (les « advices ») qui précisent ce qui doit être fait dans les différents points d'application de l'aspect



Un intercepteur en Java permet de définir des traitements, généralement transverses, qui seront exécutés lorsque certains événements vont survenir. Leur rôle est similaire aux fonctionnalités de base de l'AOP.

La généricité

- La programmation générique consiste à définir des algorithmes et des techniques susceptibles d'intervenir sur des données de types différents. Une définition à laquelle il faut ajouter d'autres critères, les fonctions variadiques, des surcharges diverses, l'idée étant que le code pour des questions de clarté, doit être le plus éloigné possible des conditions de mises en œuvre. On dit alors qu'il est générique.
- L'idée est d'augmenter le niveau d'abstraction du codage pour les langages statiques, en écrivant des programmes identiques pour traiter des données structurellement différentes.
  - Pour une classe, le type devient un paramètre et c'est le compilateur qui s'assure de la cohérence du type utilisé dans le générique (la classe).
  - Résultat : On ne code qu'une seule fois un algorithme et on n'a pas à supporter des transtypages explicites.
- C'est une forme de polymorphisme appliqué aux types.
- En C++, on pourrait par exemple créer une liste générique avec les templates :
  - **List<T>** ou <T> est un type.
  - A l'instanciation on pourra fabriquer une **List<Int>** ou une **List<String>**, le code traitant ensuite la liste, comme si son type avait été explicitement précisé.

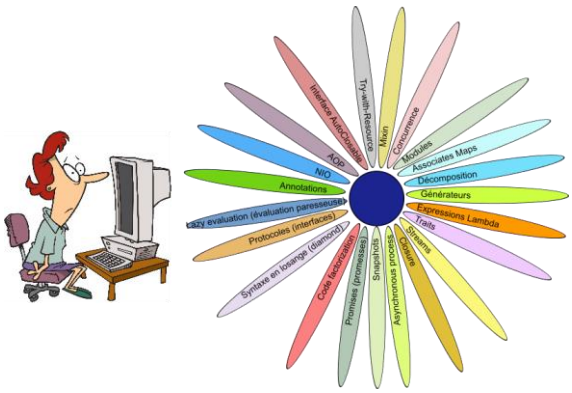
La généricité

- Il existe de nombreuses formes de programmation générique, que l'on trouve dans les langages objet, mais aussi impératifs, comme le C. Le codage générique n'est pas, limité à un langage type Ada, C++, C# ou Java.
- Au début des années 80, le concept est apparu sous la forme de « templates » (modèles) avec C++. Par la suite, la généricité s'est répandue avec .NET dans VB.NET et C#, puis dans Java 5, amélioré par la suite dans Java 7 avec la syntaxe en losange, Eiffel de Bertrand Meyer et OCaml.
- Ada 83 est considéré comme le premier à avoir implémenté un concept de programmation générique, avant les « templates » de C++. Il permettait d'écrire du code sans se préoccuper du type des données impactées.
- C'est C++ qui a poussé le plus loin l'idée, avec les « templates », des modèles génériques qui permettent de fabriquer automatiquement des fonctions ou des classes à partir d'un certain nombre de paramètres, ce que l'on appelle la "spécialisation". Et qui aboutissent donc à un code particulier adapté à un type donné, à partir du modèle générique.
- Aujourd'hui tous les langages disposent ou prévoient d'implémenter le concept (Go)

Les fonctions anonymes et lambdas

- Une fonction anonyme n'a pas de nom, le code s'écrit directement à la place du nom
- **Expressions lambda** (fonctions anonymes) : permettent d'encapsuler un traitement que l'on pourra passer en paramètres dans d'autres traitements. On les trouve en Java, mais aussi en Scala, entre autres.
- Une expression lambda peut être perçue comme une représentation concise d'une fonction anonyme qui peut être transmise : elle n'a pas de nom, mais elle a une liste de paramètres, un corps, un type de retour et peut-être aussi une liste d'exceptions :
  - **Fonction** : une expression lambda n'est pas associée à une classe particulière comme l'est une méthode.
  - **Transmissible** : une expression lambda peut être passée en argument à une méthode ou stockée dans une variable.
  - **Concise** – Il y a beaucoup moins de code à écrire.

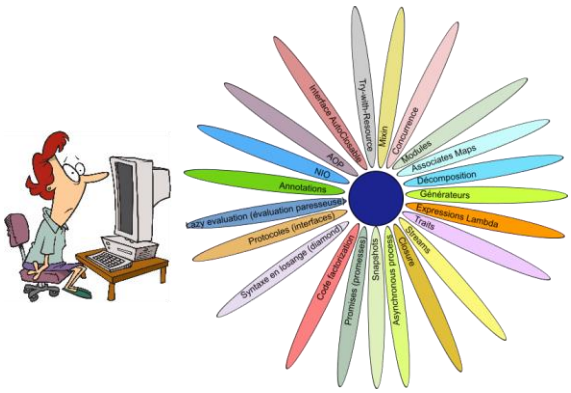
Et d'autres techniques...



- Une **fermeture** ou « **closure** » est une fonction qui accède à des variables qui se situent en dehors de son contexte local (« scope »). C'est le cas lorsque l'on intégre une fonction A dans une fonction B et que la fonction A accède aux variables de B.
- On dit que la fonction est "accompagnée" de son environnement lexical, constitué de l'ensemble des variables non locales, qu'elle capture, par valeur ou référence.
- **Closure** (fermeture) est utilisée dans un grand nombre de langages, dont PHP, Ruby, JavaScript ou CoffeeScript, en plus des **langages OO**. Il s'agit d'une fonction (par exemple), définie dans le corps d'une autre fonction et qui utilise des paramètres ou des variables locales de cette dernière. Une closure peut être passée en argument dans l'environnement où elle a été créée (passage vers le bas) et renvoyée comme valeur de retour (passage vers le haut).

- **Promises** (promesses) pour les scripts, qui prennent en compte de manière intelligente, la fin des opérations asynchrones et leur associe un gestionnaire d'erreurs (ECMAScript 2015, Node.js, Dart…) et des propriétés d'états.
- **NIO** (Non Blocking IO) des JDK 7 et 8, des API susceptibles de permettre à une application de prendre en compte d'importants volumes d'IO. Cette disposition existe depuis Java 1.4, mais retrouve des couleurs avec les JDK les plus récents.
- La **factorisation du code** pour des modules script, comme dans ECMAScript 2015, pour importer des éléments définis ailleurs dans l'application, éléments qui peuvent être une séquence de code ou une simple variable.

Et d'autres techniques...



- Le **traitement global ou séquentiel** sur des collections tel que les Streams de Ceylon.
  - Un objet iterable au sens Ceylon est un objet qui produit un flot (stream) de valeurs. Il satisfait à l'interface Iterable.
- La **syntaxe en losange** de Java, pour éviter la duplication du paramétrage des Generics, lorsque le contexte le permet (quand le compilateur peut le déduire du reste du code). Syntaxe qui consiste à laisser le paramétrage vide, à charge pour le compilateur de le déduire de son contexte. C'est du sucre syntaxique.
- **Evaluation paresseuse** (lazy evaluation) de F# ou Python, ou évaluation retardée, dans laquelle l'évaluation d'un paramètre ne se fait pas tant que les résultats de cette évaluation ne sont pas réellement nécessaires
- **Traits** de Scala, Rust, PHP, Java ou Haskell, qui sont des sous-classes abstraites, contenant des méthodes concrètes, que l'on peut ajouter à une classe pour étendre ses fonctionnalités. Les traits ressemblent aux AOP et aux interfaces, pour encapsuler un ensemble cohérent de méthodes, de manière à les réimplanter ailleurs. Un trait est habituellement constitué d'une méthode abstraite qui fait le lien avec la classe à laquelle il s'applique et diverses méthodes additionnelles.

Les concepts nouveaux de Programmation Objet

30 Juin 2020

Nos prochains rendez-vous

Mercredi 4 septembre	**: La fin des mots de passe**
Vendredi 11 septembre	**: IBN et la programmation des réseaux**
Vendredi 18 septembre	**: Le "machine learning", c'est quoi au juste**
Vendredi 25 septembre	**: Les secrets du "deep learning"**
Vendredi 2 octobre	**: Le grave danger que représentent les GAFAM**
Vendredi 9 octobre	**: Au cœur des backbones Internet, comprendre...**
Vendredi 16 octobre	**: Cyberguerre, entre fantasmes et réalités**
Vendredi 23 octobre	**: Les avancées concrètes des villes intelligentes**
Vendredi 30 octobre	**: Les algorithmes de chiffrement, ces inconnus**
Vendredi 6 novembre	**: L'IA et la fin de la démocratie**
Vendredi 13 novembre	**: Les certifications pour remplacer les diplômes**
Vendredi 20 novembre	**: IA et la démocratie**
Vendredi 27 novembre	**: La médecine du futur, les barrières explosent**
Vendredi 4 décembre	**: La transformation digitale, mythe ou réalité**
Vendredi 18 décembre	**: Panorama des architectures globales du TI**
Mercredi 23 décembre	**: Une journée comme les autres en... 2070**