**Les Types de langages**

* **Langage compilé** : le code source est donné à un programme appelé **compilateur** qui va lire le code source et le convertir dans un langage que l'ordinateur sera capable d'interpréter : c'est le langage binaire, fait de 0 et de 1. Les langages comme le C ou le C++ sont des langages dits compilés.
* **Langage précompilé** : ici, le code source est compilé partiellement, généralement dans un code plus simple à lire pour l'ordinateur, mais qui n'est pas encore du binaire. Ce code intermédiaire devra être lu par ce que l'on appelle une « machine virtuelle », qui exécutera ce code. Les langages comme le C# ou le Java sont dits précompilés.
* **Langage interprété** : dans ce cas, il n'y a pas de compilation. Le code source reste tel quel, et si on veut exécuter ce code, on doit le fournir à un interpréteur qui se chargera de le lire et de réaliser les actions demandées.

**Les interpréteurs de Javascript**

* **Internet Explorer**, son interpréteur Javascript s'appelle ***JScript*** (l'interpréteur de la version 9 s'appelle ***Chakra***),
* **Mozilla Firefox** se nomme ***SpiderMonkey***
* **Google Chrome** est ***V8***.

**Historique**

L'ECMA International standardise le langage sous le nom d'ECMAScript.

Depuis, les versions de l'ECMAScript ont évolué. La version la plus connue et mondialement utilisée est la version ECMAScript 3, parue en décembre 1999.

**En résumé**

* Le Javascript est un langage de programmation interprété, c'est-à-dire qu'il a besoin d'un interpréteur pour pouvoir être exécuté.
* Le Javascript est utilisé majoritairement au sein des pages Web.
* Tout comme le HTML, le Javascript est exécuté par le navigateur de l'internaute : on parle d'un comportement *client-side*, par opposition au *server-side* lorsque le code est exécuté par le serveur.
* Le Javascript est standardisé par l'ECMA International sous le nom d'ECMAScript qui constitue la référence du langage. D'autres langages découlent de l'ECMAScript, comme ActionScript, EX4 ou encore JScript.NET.
* La dernière version standardisée du Javascript est basée sur l'ECMAScript 5, sorti en 2009.

Condition ternaire

[**Programmation Orientée Objet**](http://www.commentcamarche.net/contents/poo-2497231587)

**Visibilité**

**public, protected, ou private**

**Proprietes**

Public : visibilité public

Protected : visible pour la redéclaration dans une sous class

Private : réserver pour la class elle même

**Méthodes**

Public : visibilité public

Protected : visible pour les sous class

Private : réserver pour la class elle même

**Interfaces**

Les interfaces objet vous permettent de créer du code qui spécifie quelles méthodes une classe doit implémenter, sans avoir à définir comment ces méthodes fonctionneront.

Les interfaces sont définies en utilisant le mot-clé interface, de la même façon que pour une classe standard, mais sans qu'aucune des méthodes n'ait son contenu de spécifié.

De par la nature même d'une interface, toutes les méthodes déclarées dans une interface doivent être publiques.

En résumé

* Une interface représente un comportement.
* Les interfaces ne sont autres que des classes 100% abstraites.
* Une interface s'utilise dans une classe grâce au mot-clé **implements**.
* Il est possible d'hériter ses interfaces grâce au mot-clé **extends**.
* Il existe tout un panel d'interfaces pré-définies vous permettant de réaliser tout un tas de fonctionnalités intéressantes, comme la création « d'objets-tableaux ».

**Polymorphisme**

Son principe réside dans son nom : [poly-] signifie beaucoup et [-morphisme] signifie formes, états. Ainsi, intégrer le polymorphisme dans la programmation permet au développeur d’initialiser un seul objet qui se comportera différemment selon son type. Pratiquement parlant, le développeur écrit une classe de base qui définit les attributs de l’objet et autant de sous-classes que de comportements différents de l’objet seront nécessaires.

Class Voiture Extends Vehicule {

}

**LES DESIGN PATTERNS**

* Un design pattern est un moyen de conception répondant à un problème récurrent.
* Laisser une classe créant les objets : **le pattern Factory**
  + **Le pattern factory** a pour but de laisser des classes usine créer les instances à votre place.
* Écouter ses objets : **le pattern Observer**
  + **Le pattern observer** permet de lier certains objets à des « écouteurs » eux-mêmes chargés de notifier les objets auxquels ils sont rattachés.
* Séparer ses algorithmes : **le pattern Strategy**
  + **Le pattern strategy** sert à délocaliser la partie algorithmique d'une méthode afin de le permettre réutilisable, évitant ainsi la duplication de cet algorithme.
* Séparer ses algorithmes : **le pattern Strategy**
  + **Le pattern singleton** permet de pouvoir instancier une classe une seule et unique fois, ce qui présente quelques soucis au niveau des dépendances entre classes.
  + Pour empêcher la création d'une instance de cette façon, c'est très simple : il suffit de mettre le **constructeur** de la classe en **privé** ou en **protégé** !
  + Bien sûr que si ! Nous allons créer une **instance** de notre classe à **l'intérieur d'elle-même** ! De cette façon nous aurons accès au **constructeur**.
* Le pattern injection de dépendances a pour but de rendre le plus indépendantes possible les classes.

**Les Traits**

* Les traits sont un moyen pour éviter la duplication de méthodes.
* C’est un moyen de **réutiliser** le code d'une méthode dans deux classes indépendantes
* Un trait s'utilise grâce au mot-clé use.
* Il est possible d'utiliser une infinité de traits dans une classe en résolvant les conflits éventuels avec **insteadof**.
* Un trait peut lui-même utiliser un autre trait.
* Il est possible de changer la visibilité d'une méthode ainsi que son nom grâce au mot-clé **as**.
* Déclaratrion :

trait **NomDuTrait**

{

public function **nomDuMethode**()

{

echo 'Hello world !';

}

}

* Appel du Trait :

Use **NomDuTrait** ;

* Appel d’une méthode d’un Trait :

$this-> **nomDuMethode(**)

* Appel d’une méthode d’un Trait :
* Appel Multiple :

use **HTMLFormater**, **TextFormater**

{

**HTMLFormater::***format* ***insteadof*** **TextFormater**;

}

* La classe plus forte que le trait
* Le trait plus fort que la mère
* Changer la visibilité et le nom des méthodes

class **MaClasse**

{

use A

{

saySomething **as**  protected sayWhoYouAre;

}

}

**service container symfony2**

Injection des serevices dans le kernel

Cet objet, appelé un conteneur de services, vous permettra de standardiser et centraliser la façon dont les objets sont construits dans votre application

Scope : pour traiter ici des champs d'applications

Tag :

**doctrine symfony2**

**l'ORM Doctrine**, est une **bibliothèque** dont l'objectif est de **mapper vos objets avec une base de données** relationnelle (comme MySQL, PostGresSQL ou Microsoft SQL). Si vous préférez utiliser des requêtes SQL brutes

**composer symfony 2**

**Composer** est une **bibliothèque de gestion de dépendances pour PHP**, que vous pouvez utiliser pour télécharger l'Édition Standard de Symfony2.

**composant sécurité symfony 2**

La sécurité est un processus comprenant 2 étapes, dont le but est de prévenir un utilisateur d'accéder à une ressource à laquelle il n'a pas accès.

1. Dans la première étape du processus, le système de sécurité identifie l'utilisateur en lui demandant de soumettre une sorte d'identification. C'est ce qu'on appelle **l'authentification**, et cela signifie que le système cherche à savoir qui vous êtes.
2. Une fois que le système sait qui vous êtes, l'étape suivante est de déterminer si vous avez accès à une ressource donnée. Cette étape du processus est appelée **autorisation**, et cela signifie que le système vérifie si vous avez les privilèges pour exécuter certaines actions.

app/config/security.yml

* Sécuriser par IP

|  |
| --- |
| # app/config/security.yml  security:  # ...  access\_control:  - { path: ^/cart/checkout, roles: IS\_AUTHENTICATED\_ANONYMOUSLY, ip: 127.0.0.1 } |

* Sécuriser un contrôleur
  + 1. le Bundle **JMSSecurityExtraBundle**, qui peut sécuriser un contrôleur en utilisant les annotations

/\*\*

\* @Secure(roles="ROLE\_ADMIN")

\*/

**Les types de caches**

* **Symfony2 Reverse Proxy**

Symfony2 contient un reverse proxy (aussi appelé passerelle de cache) écrit en PHP. Activez-le et les réponses de votre application qui peuvent être mise en cache seront immédiatement stockées. L'installer est aussi très simple. Chaque nouvelle application Symfony2 contient un noyau pré-configuré (AppCache) qui encapsule le noyau par défaut (AppKernel). Le cache kernel (cache du noyau) est le reverse proxy.

* **Le cache du navigateur** : tous les navigateurs ont leur propre cache qui est utile quand un utilisateur demande la page précédente ou des images et autres médias. Le cache du navigateur est un cache privé, car les ressources mises en cache ne sont partagées avec personne d'autre.
* **Le « cache proxy »** : un proxy est un cache partagé car plusieurs personnes peuvent être derrière un seul proxy. Il est habituellement installé par les entreprises et les FAIs pour diminuer le temps de réponse des sites et la consommation des ressources réseau.
* **Reverse Proxy (Passerelle de cache)** : comme un proxy, ce système de cache est également partagé, mais du côté serveur. Installé par des administrateurs réseau, il permet aux sites d'être plus extensibles, sûrs et performants.

Symfony2 contient un reverse proxy (aussi appelé passerelle de cache) écrit en PHP. Activez-le et les réponses de votre application qui peuvent être mise en cache seront immédiatement stockées. L'installer est aussi très simple. Chaque nouvelle application Symfony2 contient un noyau pré-configuré (AppCache) qui encapsule le noyau par défaut (AppKernel). Le cache kernel (cache du noyau) est le reverse proxy.

**memcached php**

» memcached est un système à haute performance, distribué et en mémoire, de cache d'objet, générique par nature, mais principalement destiné à l'utilisation avec des sites Web dynamiques, pour soulager les bases de données.