Les interfaces

Introduction

- Les interfaces ressemblent aux classes abstraites sur un seul point : elles contiennent des membres **expliquant certains comportements sans les implémenter**.

- Les classes abstraites et les interfaces se différencient principalement par le fait qu'**une classe peut implémenter un nombre quelconque d'interfaces**, alors qu'une classe abstraite ne peut hériter que d'**une seule classe** abstraite ou non.

Vocabulaire et concepts :

- Une interface est un contrat, elle peut contenir des **propriétés**, des **méthodes** et des **événements** mais **ne** doit contenir **aucun champ** ou **attribut**.
- Une interface **ne** peut **pas** contenir des méthodes déjà implémentées.
- Une interface doit contenir des méthodes **non** implémentées.
- Une interface est héritable.
- On peut construire une hiérarchie d'interfaces.
- Pour pouvoir construire un objet à partir d'une interface, il faut définir une classe non abstraite implémentant **toutes** les méthodes de l'interface.

Une classe peut implémenter plusieurs interfaces. Dans ce cas nous avons une excellente alternative à l'héritage multiple.

Lorsque l'on crée une interface, on fournit un ensemble de définitions et de comportements qui ne devraient plus être modifiés. Cette attitude de constance dans les définitions, protège les applications écrites pour utiliser cette interface.

Les variables de types interface respectent les mêmes règles de **transtypage** que les variables de types classe.

Les **objets** de type classe **clA** peuvent être transtypés et **référencés** par des variables d'interface **IntfA** dans la mesure où la classe **clA implémente l'interface IntfA**. (cf. polymorphisme d'objet)

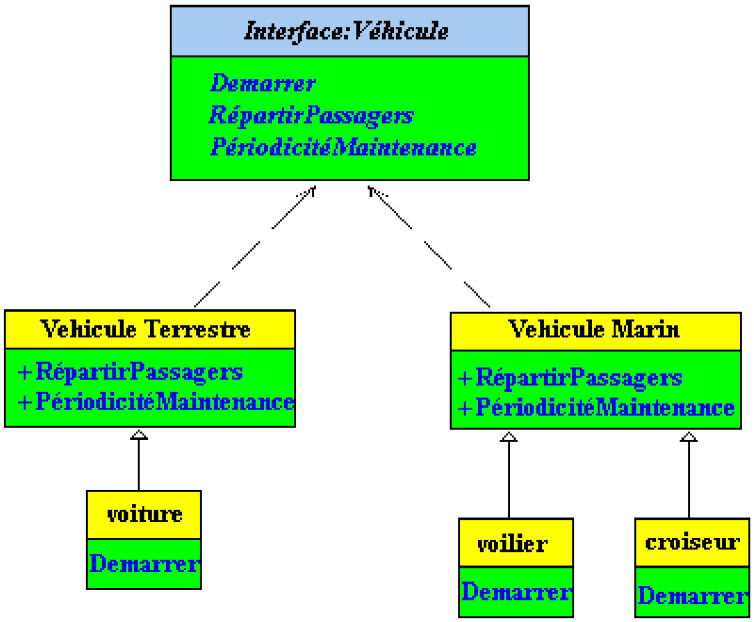
Si vous voulez utiliser la notion d'interface pour fournir un polymorphisme à une famille de classes, elles doivent toutes implémenter cette interface, comme dans l'exemple ci-dessous.

Exemple :

l'interface ***Véhicule*** définissant 3 méthodes (abstraites) **Démarrer**, **RépartirPassagers** de répartition des passagers à bord du véhicule (fonction de la forme, du nombre de places, du personnel chargé de s'occuper de faire fonctionner le véhicule...), et **PériodicitéMaintenance** renvoyant la périodicité de la maintenance obligatoire du véhicule (fonction du nombre de kms ou miles parcourus, du nombre d'heures d'activités,...)

Soit l'interface ***Véhicule*** définissant ces 3 méthodes :

Soient les deux classes **Véhicule terrestre** et **Véhicule marin**, qui implémentent partiellement chacune l'interface ***Véhicule*** , ainsi que trois classes **voiture**, **voilier** et **croiseur** héritant de ces deux classes :



- Les trois méthodes de l'interface ***Véhicule*** sont abstraites et publics par définition.

- Les classes **Véhicule terrestre** et **Véhicule marin** sont abstraites, car la méthode

abstraite **Démarrer** de l'interface ***Véhicule*** n'est pas implémentée elle reste comme "modèle" aux futurs classes. C'est dans les classes **voiture**, **voilier** et **croiseur** que l'on implémente le comportement précis du genre de démarrage.

Dans cette vision de la hiérarchie on a supposé que les classes abstraites **Véhicule terrestre** et **Véhicule marin** savent comment répartir leurs éventuels passagers et quand effectuer une maintenance du véhicule.

Les classes **voiture**, **voilier** et **croiseur** , n'ont plus qu'à implémenter chacune son propre comportement de démarrage.

Syntaxe de l'interface en Java (C# est semblable à Java) :

	Java
	Interface Vehicule **{** **void** Demarrer( ); **void** RépartirPassagers( ); **void** PériodicitéMaintenance( ); **}**

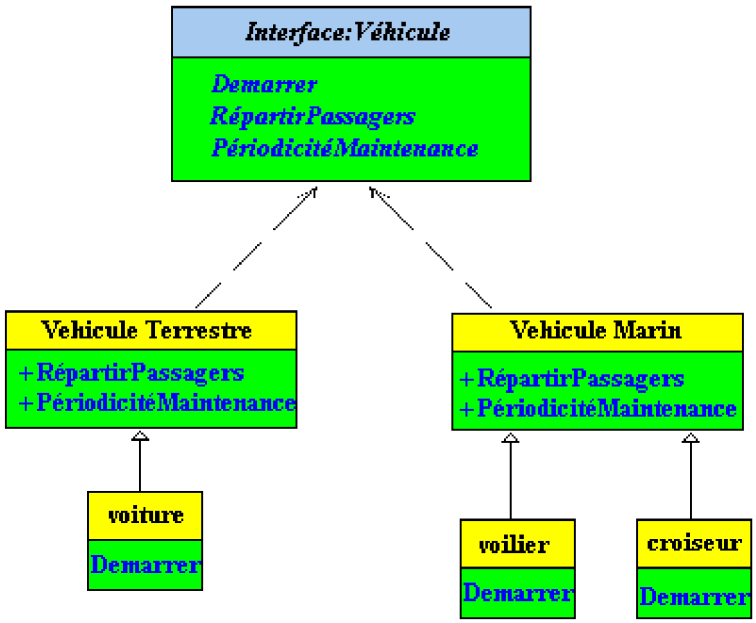
Utilisation pratique des interfaces

Quelques conseils prodigués par des développeurs professionnels (microsoft, Borland) :

- Les interfaces bien conçues sont plutôt petites et indépendantes les unes des autres.
- Un trop grand nombre de fonctions rend l'interface peu maniable.
- Si une modification s'avère nécessaire, une nouvelle interface doit être créée.
- La décision de créer une fonctionnalité en tant qu'interface ou en tant que classe abstraite peut parfois s'avérer difficile.
- Vous risquerez moins de faire fausse route en concevant des interfaces qu'en créant des arborescences d'héritage très fournies.
- Si vous projetez de créer plusieurs versions de votre composant, optez pour une classe abstraite.
- Si la fonctionnalité que vous créez peut être utile à de nombreux objets différents, faites appel à une interface.
- Si vous créez des fonctionnalités sous la forme de petits morceaux concis, faites appel aux interfaces.
- L'utilisation d'interfaces permet d'envisager une conception qui sépare la manière d'utiliser une classe de la manière dont elle est implémentée.

- Deux classes peuvent partager la même interface sans descendre nécessairement de la même classe de base.

Exemple de hiérarchie à partir d'une interface :



Dans cet exemple :

Les méthodes RépartirPassagers, PériodicitéMaintenance et Demarrer sont implantées soit comme des méthodes à liaison dynamique afin de laisser la possibilité pour des classes enfants de surcharger ces méthodes.

Soit l'écriture en Java de cet l'exemple :

```
interface IVehicule{
      void Demarrer( );
      void RépartirPassager( );
      void PériodicitéMaintenance( );
  }

abstract class Terrestre  implements IVehicule {
      public void RépartirPassager( ){.....};
      public void PériodicitéMaintenance( ){.....};
 }

class Voiture extends Terrestre {
      public void Demarrer( ){.....};
 }

abstract class Marin implements IVehicule {
```

```
        public void RépartirPassager( ){.....};
        public void PériodicitéMaintenance( ){.....};
}


class Voilier extends Marin {
        public void Demarrer( ){.....};
}
 class Croiseur extends Marin {
        public void Demarrer( ){.....};
}
```