# Héritage en Java – Casting - Génériques

# Objectifs

Revoir des principes de la POO (Programmation orientée objet) déjà exposés dans le document « POO - Principaux Concepts de la programmation à objets en Java & PHP »

# Syntaxe

Mot clef extends.

Toutes les classes héritent de la classe Object.

# Modèle utilisé comme exemple

# 

Ici il s’agit d’un diagramme de classe UML pour la conception, c’est-à-dire l’implémentation d’une application, et non pas d’analyse.

On a choisi de faire figurer l’équipage, liste de Marins, dans Navire (choix contraire à l’implémentation forcée relationnelle / SQL où la clef étrangère irait dans Marin pour relier à un Navire).

On fait figurer un nouveau concept, l’interface Piraterie et l’implémentation de cette interface par le Pirate.

# Redéfinition

Les classes filles peuvent redéfinir les méthodes de la classe mère.

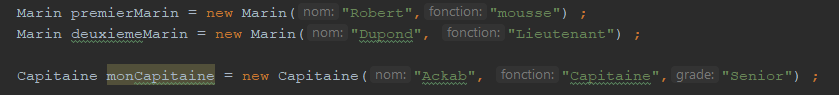
Il est recommandé de redéfinir les méthodes **equals** et **toString** de la classe Object.

Si on a une classe Pirate, la méthode **equals** nous permettra de préciser qu’est-ce qu’on considère être l’égalité de 2 pirates ?

La méthode **toString** permettrad’afficher une instance de Pirate (sinon par défaut c’est la méthode toString de la classe Object qui est appelée et affiche le hashcode).

# Casting

Supposons que l’on ait une classe Marin et une classe Capitaine qui étend (est fille de) la classe Marin.



Le code ci-dessus instancie 2 marins et un capitaine.

## Casting ascendant - Upcasting

On peut aussi déclarer un Marin qui référence un Capitaine.



Ou directement instancier un Marin comme Capitaine :



Ce casting est toujours autorisé et il est **implicite** : on parle de Casting ascendant ou **Upcasting.** Le principe est que « qui peut le plus peut le moins ». Un Marin instancié comme Capitaine ne pose pas de problème car un Capitaine ***est*** un Marin.

Pour l’upcasting il est effectué automatiquement sans avoir besoin de préciser le **cast** entre **parenthèses**.

En revanche, les méthodes de la classe Capitaine ne seront pas accessibles.

Par exemple :



Ne fonctionne pas car l’accesseur getGrade de Capitaine n’est pas accessible pour un Marin.

## Casting descendant - Downcasting

Le casting descendant doit être explicite, sinon il génère une erreur à la compilation (le programme ne peux même pas être exécuté).

**Il n’est pas garanti**, et peut générer un ClassCastException.

Exemple : downcasting réussi



L’exemple ci-dessous est Ok, car monCapitaineMarin est instancié comme Capitaine. La méthode getGrade n’est pas accessible au Marin, mais le downCasting marche car le Marin a été instancié comme Capitaine.

Renvoie Senior.

L’instanciation d’un nouveau Capitaine par downCasting marche également :



Exemple : downcasting plantés

1. DownCasting sans enfant

L’exemple ci-dessous est KO, à **l’exécution** :



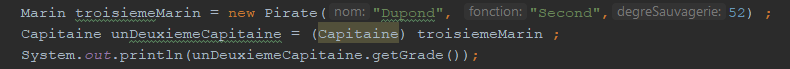
Un Marin ***n’est pas*** un Capitaine et donc le DownCasting ne marche pas.

Erreur générée :



1. Downcasting vers le « mauvais » enfant

L’exemple ci-dessous est KO, à **l’exécution** :



Renvoie un ClassCastException :



Le downcasting vers un Pirate aurait marché ici.

# Utilisation

## Upcasting

Utile lorsqu’on veut écrire du code qui ne concerne que la classe mère.

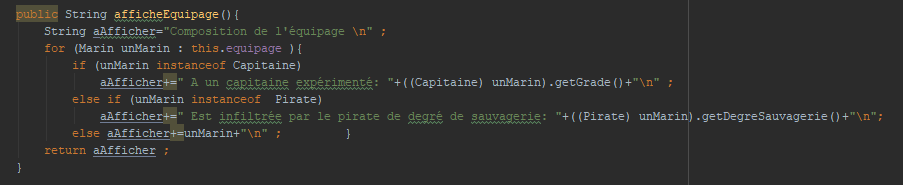
Cf dans notre exemple de projet l’attribut equipage de Navire qui est une liste de Marin, et dont le type réel, à savoir Marin, Capitaine ou Pirate n’a pas d’importance.

## Downcasting

Davantage utilisé. Critiqué à un niveau avancé de développement (préférer des Design Patterns).

On l’utilise souvent conjointement à l’operateur instanceof pour tester la classe d’instanciation de l’objet et en fonction appeler le code spécifique de la classe enfant nécessaire.

Par exemple, avec notre petit projet, on pourrait avoir une méthode afficheEquipage comme telle :



Exemple dans Android :

Dans les précédentes versions de SDK Android, il fallait downcaster les Widgets vers les instances correspondantes en Java. Cf par exemple les EDC BTS SIO de 2019 (Restiloc), 2017 (AHM) …



Pourquoi était ce nécessaire ?

findViewById est une méthode de la classe Activity dont la signature était :

**public View findViewById (int id)**

En paramètre l’entier est un id et elle renvoie une View.

Pourquoi renvoie t’elle une View ? Car il s’agit de la mère des Widgets et ainsi on a besoin d’une méthode générique qui renvoie la Widget qu’elle quelle soit.

Comme on « **sait »** (le programmeur) que notre View R.id.listv est en fait une ListView (qui étend View) puisqu’on l’a déclaré ainsi dans le fichier XML , on peut faire un downCast de View 🡪 ListView , mais si on « ment » au compilateur il nous renverra une ClassCastException.

Dorénavant les nouveaux SDK Android utilisent le mécanisme des Génériques en Java ce qui évite la nécessité du downcasting (hors périmètre pour le moment).