

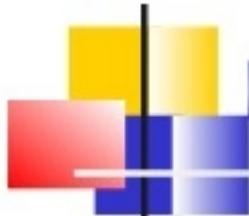
Introduction à la programmation objet (3)

L'héritage, troisième principe fondateur de la POO

Le feuilleton des comptes en banque

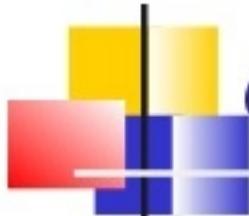
Méthodes et héritage

Classes abstraites



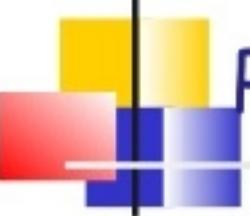
Dans les épisodes précédents

- Cours 1
 - Introduction des concepts fondamentaux de la POO (et de Ruby) :
 - le système est composé d'objets
 - les objets interagissent en s'envoyant des messages
 - l'envoi d'un message provoque l'exécution d'une méthode dans le contexte de l'objet receveur
 - Tous les objets sont instances d'une classe
- Cours 2
 - présentation de la syntaxe Ruby.



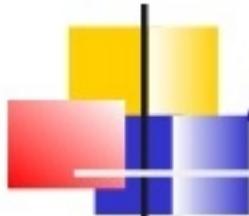
Cours 3

- L'héritage, le troisième principe fondateur de la POO (et de Ruby).
 - Nous allons nous centrer davantage sur les classes, c'est-à-dire sur ce qui peut être commun (et donc aussi ce qui est différent) aux objets appartenant à des classes distinctes.
 - L'héritage est un mécanisme qui permet de regrouper dans une superclasse ce qui est commun à plusieurs classes appelées sous-classes.
 - Ce cours présente ce principe de classement des classes, en quoi les sous-classes diffèrent de leur superclasse, comment l'héritage modifie l'association des méthodes aux messages et enfin ce cours donne un premier aperçu de l'utilisation de l'héritage dans le système.



Plan

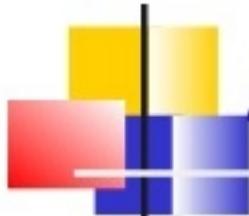
- A-Héritage
 - 1. Principe
 - 1. Exemple
 - 2. Lien d'héritage/d'instanciation
 - 2. Héritage
 - 1. Terminologie
 - 2. Utilisation
 - 3. Polymorphisme (Parenthèse)
 - 4. Programmation
 - 5. Héritage Statique
 - 6. Héritage Dynamique
 - 7. Message envoyé à super
 - 3. Classe Abstraite
- B-La hiérarchie des classes Ruby
- C-Exemple
- D-Conclusion



A-Objets, messages et classes

- 3 principes de base
 - B1 : approche simulatoire de la programmation
 - notion d'objets et de messages
 - B2 : distinction entre concept et représentant du concept
 - notion de classes et d'instances
 - B3 : classification des concepts
 - notion d'héritage

Tous les LOO
sont fondés sur ces principes



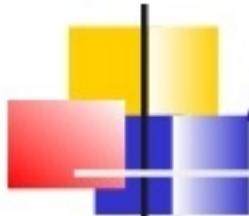
A-Résumé des principes 1 et 2

- Principe 1

- Un Objet = Une identité + un état + un comportement
 - En réponse à un message l'objet destinataire du message déclenche un comportement (une méthode / une fonction membre)

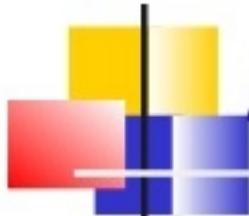
- Principe 2

- Une classe est un moule pour fabriquer des objets.
 - La classe regroupe ce qui est commun aux objets d'un même type.
 - La classe définit les attributs des objets
 - La structure de l'état des objets
 - Le comportement



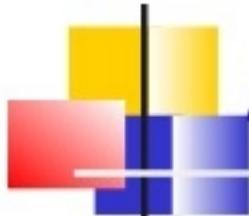
A-Résumé des principes 1 et 2

- Les instances d'une classe représentent toutes le même genre de composant du système :
 - toutes les instances d'une classe répondent au même ensemble de messages et traitent ces messages en exécutant le même ensemble de procédures
 - i.e. les instances d'une classe ont un comportement identique
 - toutes les instances d'une classe ont les mêmes champs , la même structure ; seules les valeurs de ces champs les distinguent.



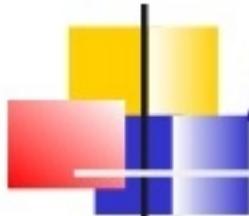
A-Objets, messages et classes

- 3 principes de base
 - B1 : approche simulative de la programmation
 - notion d'objets et de messages
 - B2 : distinction entre concept et représentant du concept
 - notion de classes et d'instances
 - ➡ ■ B3 : classification des concepts
 - notion d'héritage



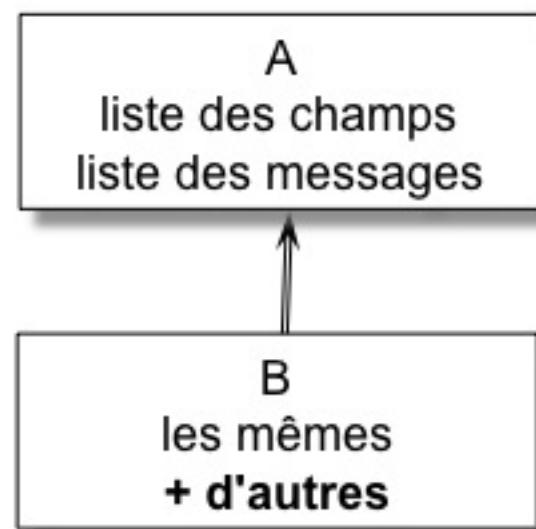
A1-Principe de l'héritage - Classification

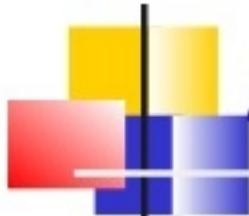
- L'héritage s'intéresse aux composants qui sont de même nature mais qui diffèrent quelque peu. Les différences peuvent être :
 - externes : ces composants répondent
 - soit à des messages différents
 - soit différemment à des messages identiques
 - internes : ces composants ont des structures différentes.
- L'héritage est ainsi **un** moyen d'organiser l'ensemble des classes en regroupant celles qui ont un certain type de propriétés en commun.



A1-Principe de l'héritage - Classification

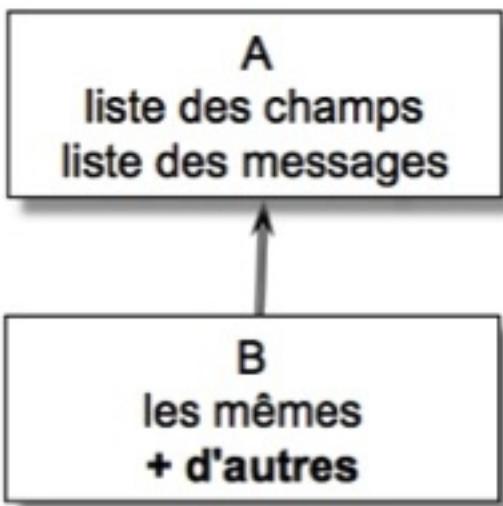
- Une classe modélise un concept abstrait, un type utilisateur
- Une classification des concepts (ou des types) fondée sur "une" relation de généralisation/ spécialisation
- Notion d'héritage : représentation partielle de cette relation
- Hypothèses : POO, A et B deux classes
 - B est une spécialisation de Assi
 - tous les champs et tous les comportements des objets de A se retrouvent dans les objets de B



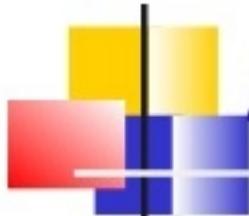


A1-Principe de l'héritage - Classification

- Les instances de B auront les mêmes caractéristiques que celles A
 - Seules différences acceptées
 - Toutes les instances d'une sous-classe B par rapport à celle de la superclasse A peuvent :



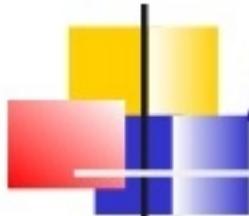
- éventuellement posséder des champs supplémentaires (variables d'instance supplémentaires)
- éventuellement savoir traiter des messages supplémentaires (méthodes supplémentaires)
- éventuellement traiter différemment certains messages (méthodes de la superclasse sont redéfinies i.e. les messages ont même nom dans la superclasse et dans sa sous-classe mais les méthodes associées sont différentes)



A11-Exemple : La classe Mère

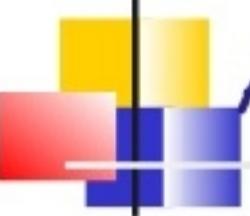
- un Compte
 - connaît son titulaire, son solde, son numéro
 - est capable de
 - retourner son solde
 - déposer une somme
 - retirer une somme
 - retourne le seuil minimum de retrait
 - afficher ce qu'il connaît
- Certains types de comptes fournissent des services supplémentaires
 - autoriser des découverts (CompteCourant)
 - rapporter des intérêts (CompteEpargne)

```
# Les méthodes qui suivent sont publiques puisque on ne précise rien de particulier.-
# Pour déposer une somme on incrémente la variable d'instance @solde -
def depose (uneSomme)-
> @solde += uneSomme-
end-
-
# Le seuil de retrait est 0 (pas de découvert autorisé)-
def seuil ()-
> return 0-
end-
-
# Pour retirer une somme on décrémente la variable d'instance @solde si le seuil n'est pas -
# atteint-
def retire (uneSomme)-
> if @solde - uneSomme > self.seuil-
>   > @solde -= uneSomme-
> else-
>   > print "# Plus de Sous \n"-
> end-
end-
-
# Afficher les informations relatives au compte receveur-
def to_s-
  return "# Je suis le Compte N°#{@numero}\n" +-
         "#\tTitulaire : #{@titulaire}\n#\tSolde      : #{@solde}\n"-
end-
```



A11-Exemple : une première sous-classe

- un CompteCourant est un Compte qui
 - connaît
 - ce que connaissent tous les Comptes (partie héritée)
 - PLUS : le montant du découvert autorisé
 - est capable de
 - faire ce que font tous les Comptes (partie héritée)
 - MAIS il fournit un service spécial pour retirer des sous : le seuil de retrait est
 - non pas 0 (par défaut)
 - mais - le découvert autorisé



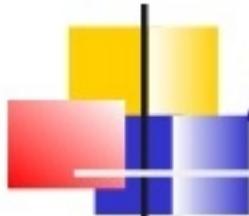
A11-Exemple : CompteCourant

```
# Une Variable d'instance supplémentaire
@decouvert    # Précise le découvert autorisé sur le compte
# Génération des méthodes découvert et découvert=
> attr :decouvert, true

# Le seuil de retrait est - découvert autorisé
# Méthode surchargé de la classe compte
def seuil ()-
  > return -self.decouvert()
end

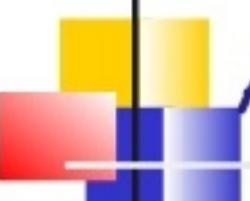
# Pour ouvrir un compte on ne peut utiliser que la méthode de classe ouvrir avec 4 arguments
def CompteCourant.ouvrir (unNombre, unNom, uneSomme, unDecouvert)-
  new(unNombre, unNom, uneSomme, unDecouvert)
end
```

On dit que CompteCourant **redéfinit** ("override", surdéfinit ou masque, ou écrase) la méthode seuil de Compte.



A11-Exemple : une seconde sous-classe

- un CompteEpargne est un Compte qui
 - connaît
 - ce que connaissent tous les Comptes (partie héritée)
 - PLUS : le taux d'intérêts
 - est capable de
 - faire ce que font tous les Comptes (partie héritée)
 - EN PLUS il fournit un service spécial : il peut verser des intérêts



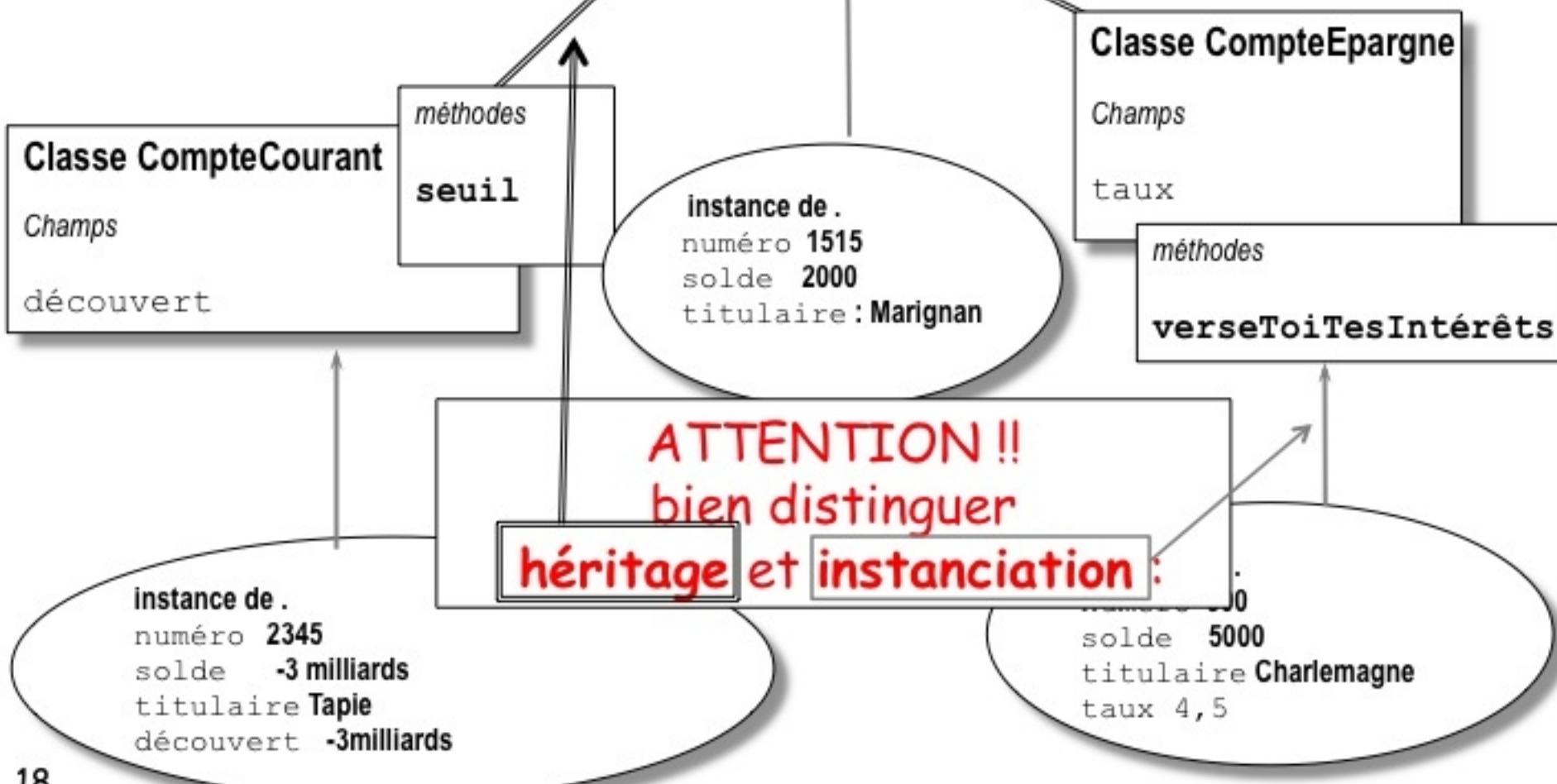
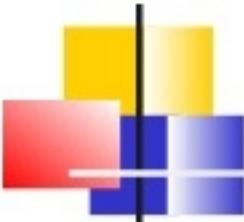
A11-Exemple : CompteEpargne

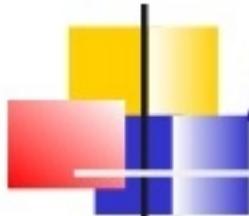
```
# Une Variable d'instance supplémentaire
@taux    # Précise le taux de rémunération
# Génération des méthodes taux et taux=
> attr :taux, true
->

# La méthode qui verse les intérêts
def verseToiTesInterets()
  > self.depose(@solde * @taux/100) / 12
end
->

# Pour ouvrir un compte on ne peut utiliser que la méthode de classe ouvrir avec 4 arguments
def CompteEpargne.ouvrir (unNombre, unNom, uneSomme, unTaux)
  new(unNombre, unNom, uneSomme, unTaux)
end
```

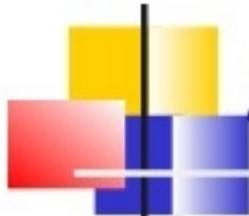
A11-La hiérarchie





A12-Lien d'héritage et d'instanciation

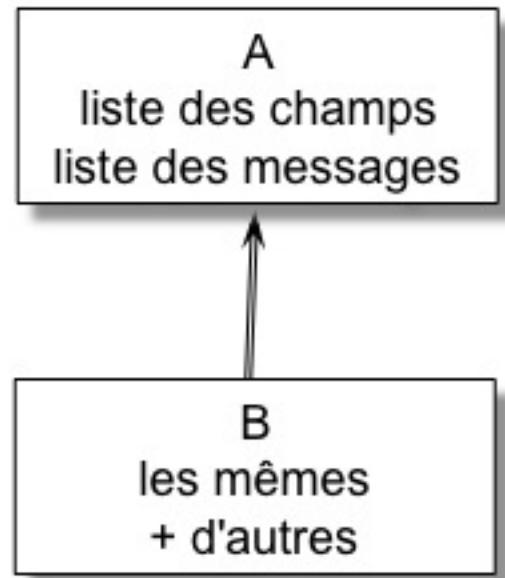
- Le lien d'héritage entre classes est un lien
 - de spécialisation entre 2 concepts abstraits
 - de sous-typage
 - d'inclusion ensembliste.
- Le lien d'instanciation entre un objet et une classe est un lien
 - de représentation
 - de déclaration de type
 - de fabrication d'objets
 - d'appartenance.

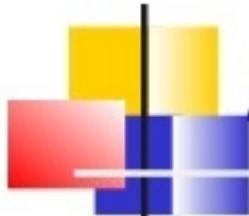


A21-Héritage : Terminologie

- A est plus générale que B
- A est une super classe de B
(superclass en Smalltalk)
- A est une classe de base pour B
(en C++)
- A est la classe mère de B

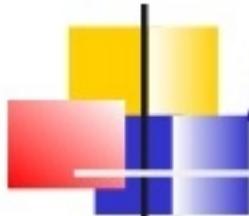
- B est une spécialisation de A
- B hérite de A
- B dérive de A (en C++)
- B est une sous-classe de A
(subclass en Smalltalk)
- B est une classe fille de A





A22-Héritage : Utilisation

- Règle N°1 :
(s'en souvenir jusqu'au restant de ses jours)
- L'héritage (simple) modélise une relation "est-un" (inclusion, ss-ensemble, ss-type)
- L'héritage ne modélise jamais "a-un" ou "est-composé-de"
(ni "est-implémenté-en-termes-de")
- Distinguer
 - la relation d'héritage
 - la relation de composition
- Organisation de classes en un graphe (arbre)
d'héritage



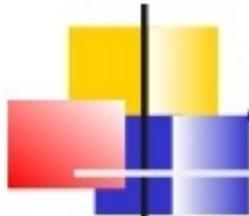
A23-Polymorphisme (parenthèse)

- Concept très important en programmation et particulièrement en POO :
 - du grec, signifie plusieurs (*poly*) formes (*morphos*)
 - définition : Un même nom représente plusieurs objets selon le contexte
- 3 types de polymorphisme
 - le polymorphisme **ad hoc**
 - le polymorphisme d'**héritage**
 - le polymorphisme **paramétrique**



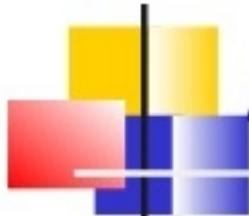
A23-Polymorphisme (parenthèse)

- Le polymorphisme fait intervenir la notion de type.
 - Pourtant les langages objets non typés (comme Smalltalk ou Ruby) autorisent les deux premiers types de polymorphisme :
 - Polymorphisme Ad Hoc
 - Polymorphisme d'Héritage
 - Le polymorphisme paramétrique : C++ et plus récemment Java (depuis 1.5).



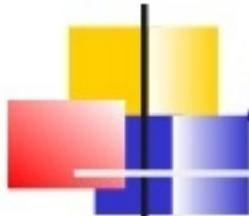
A23-Polymorphisme Ad Hoc

- **Polymorphisme ad hoc** (overloading): possibilité pour deux classes indépendantes d'avoir des méthodes de même nom, on dit de *surcharger une méthode*
 - **afficheToi** est une méthode polymorphe : on peut afficher un point, un compte en banque, un arbre...



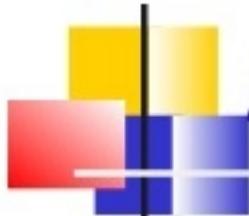
A23-Polymorphisme d'Héritage

- **Polymorphisme d'héritage** (overriding) : possibilité pour une sous-classe de *redéfinir* le code d'une méthode héritée
 - exemples en Smalltalk :
 - la méthode `isNil` est définie dans `Object` et redéfinie dans la classe `UndefinedObject`
 - la méthode `ifTrue:ifFalse:` est définie dans `Boolean` et redéfinie dans les classes `True` et `False`
 - Exemple en Ruby voir les classes Compte plus loin



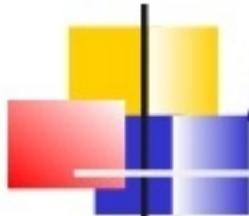
A23-Surcharge en Java/C++

- réaliser le "polymorphisme ad hoc"
- mécanisme :
 - un même nom désigne différentes fonctions dans une même portée
 - le compilateur choisit la fonction à appliquer **en fonction du type** des arguments
- En Ruby pas de type mais des fonctions à nombre variable d'arguments et des arguments par défaut.



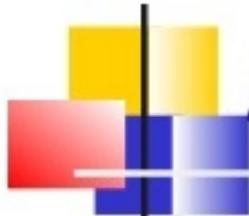
A24-Héritage : Programmation

- Technique de développement par raffinements successifs
 - en partant des classes les plus générales assez simples
 - pour arriver à des classes spécialisées complexes
- Pour définir une sous-classe on se borne à définir :
 - les champs et comportements supplémentaires
 - les comportements qui diffèrent
- Réalisation informatique
 - héritage statique des champs
 - héritage dynamique des procédures qui sont amenées à différer



A25-Héritage statique des champs

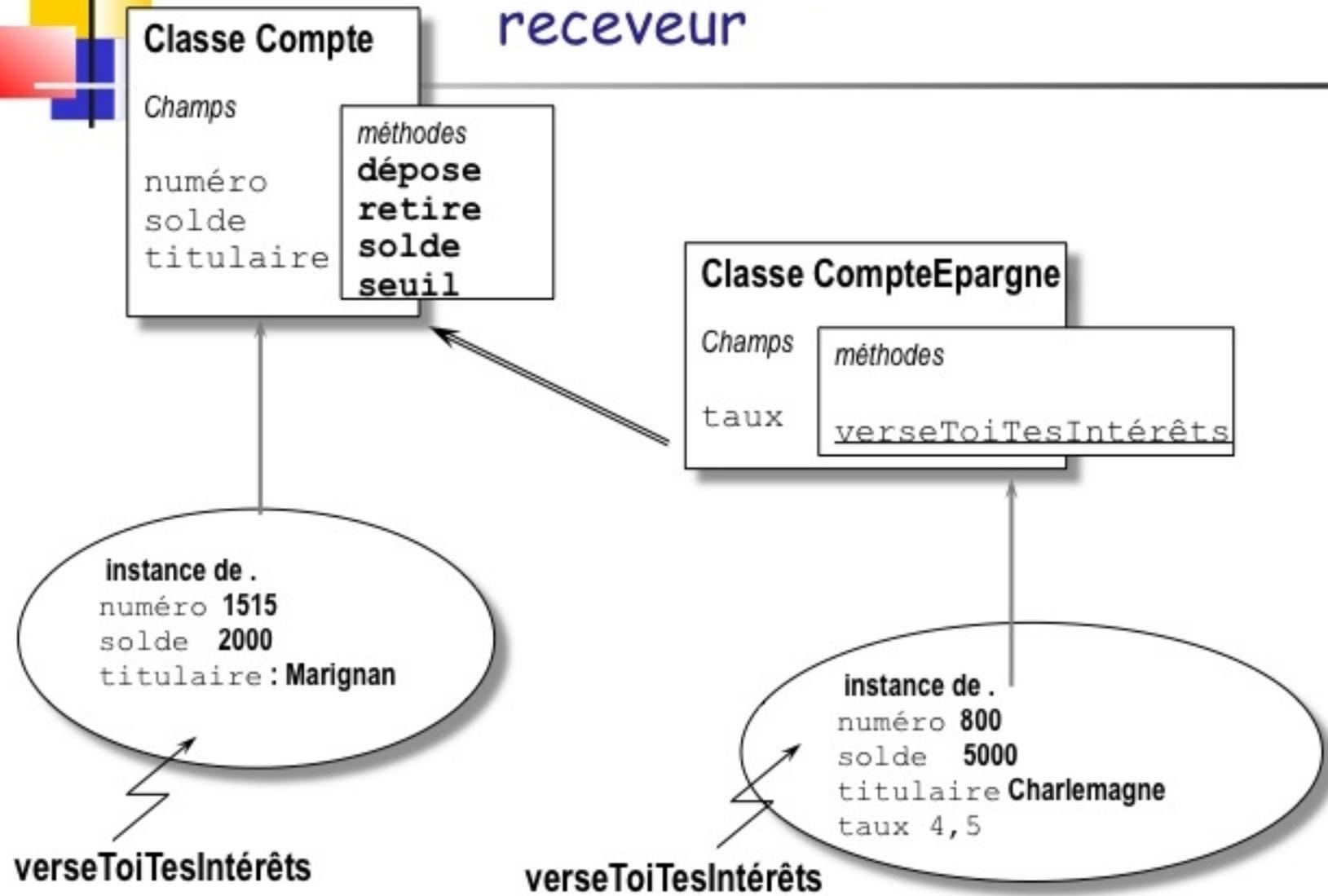
- tous les champs de la (des) superclasse(s) sont ajoutés à ceux qui sont déclarés dans la classe héritière
- une instance possède physiquement tous les champs de sa classe et de ses superclasses
- exemple :
 - unCompteCourant possède
 - un numéro, un solde, un titulaire
(champs hérités de la classe Compte)
 - un découvert autorisé
(champ propre à la sous-classe)



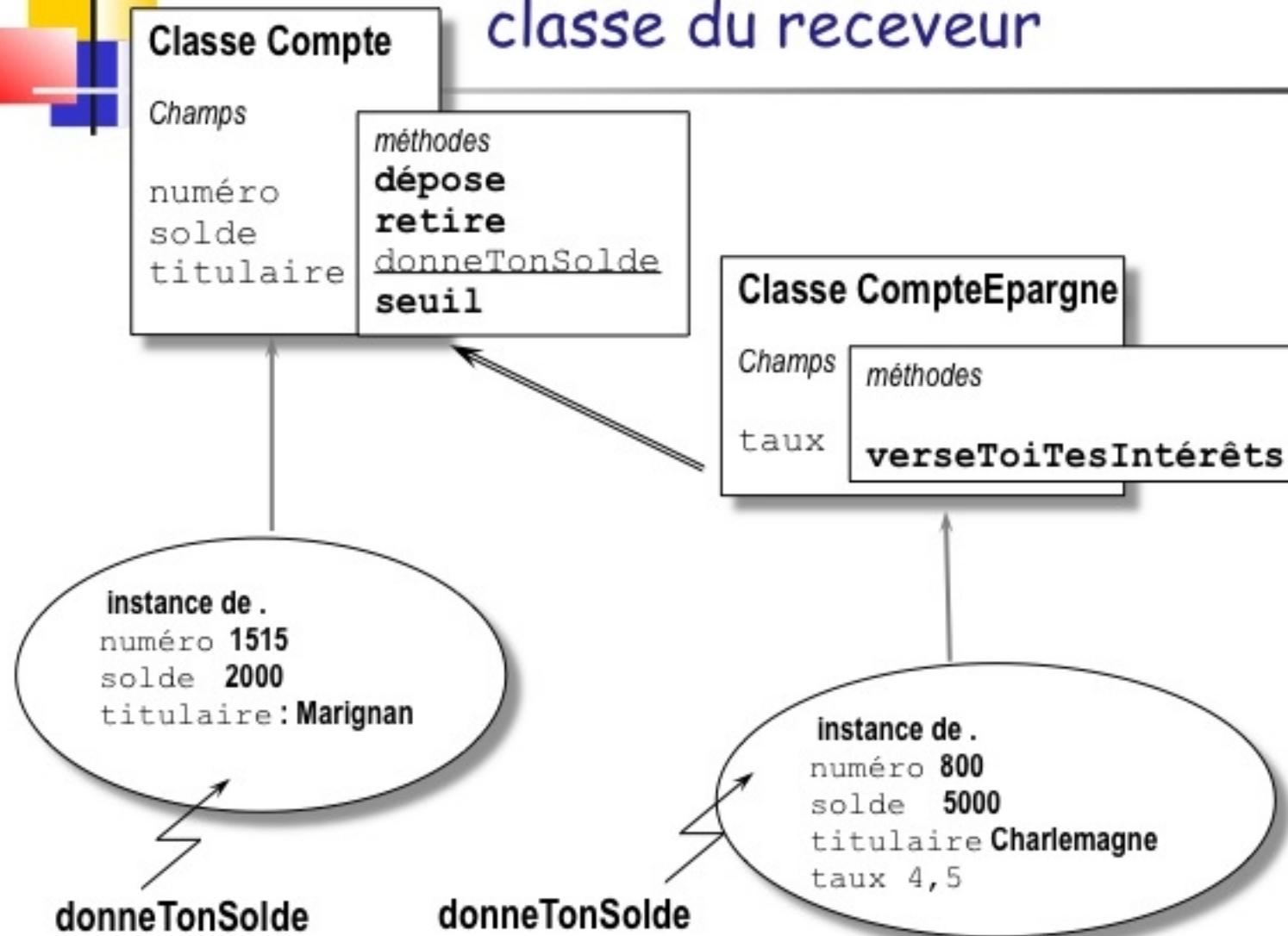
A26-Héritage dynamique des méthodes

- les textes des procédures d'une superclasse sont partagés par tous les objets de la classe ET de ses sous-classes
 - factorisation du code
- recherche de la procédure à exécuter : en remontant dans le graphe d'héritage de la classe de l'objet receveur vers ses superclasses
 - à la réception d'un message 4 cas se présentent :
 - la méthode n'est définie que dans la sous-classe (propre)
 - la méthode n'est définie que dans la super-classe (héritée)
 - la méthode est définie dans les 2 (redéfinie)
 - la méthode n'est définie nulle part (erreur)

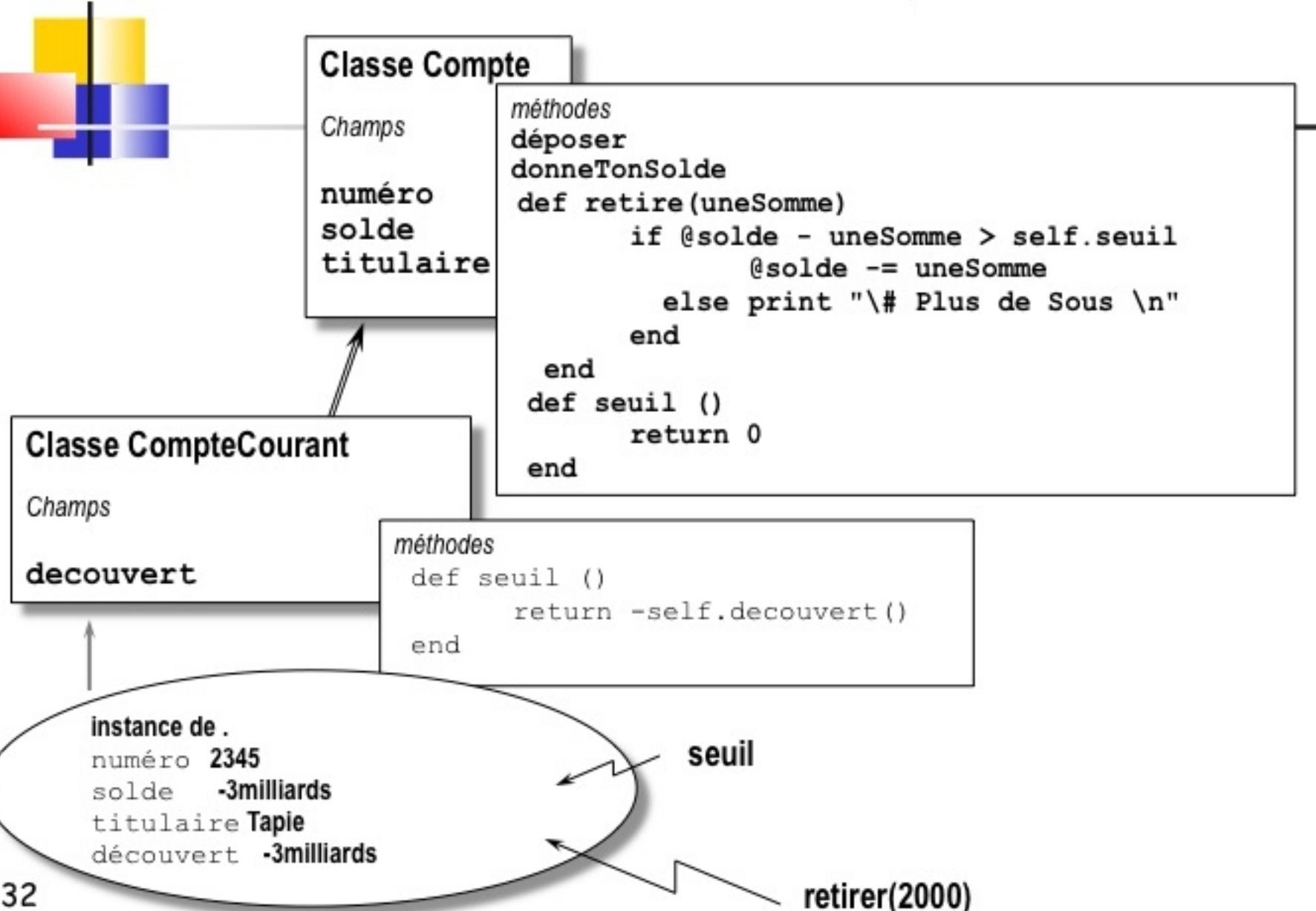
A26-Méthode définie que dans la classe du receveur

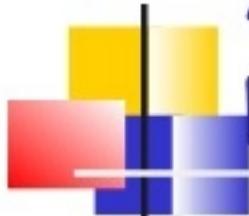


A26-Méthode définie que dans une super-classe du receveur



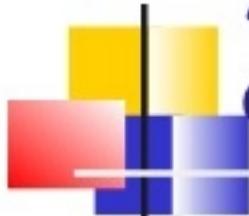
A26-Méthode héritée et redéfinie





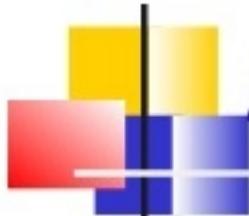
A26-Héritage dynamique des procédures (suite)

- Surdéfinition d'une méthode (redéfinition)
 - Si une classe définit une procédure de même nom qu'une procédure d'une classe mère,
 - on dit que la procédure héritée est "redéfinie" ("to override" : supplanter, outrepasser)
- Nécessité de liaison dynamique, retardée (dynamic binding, late binding)
 - le lien entre le nom du message et la procédure à invoquer doit être établi à l'exécution (et non à la compilation)



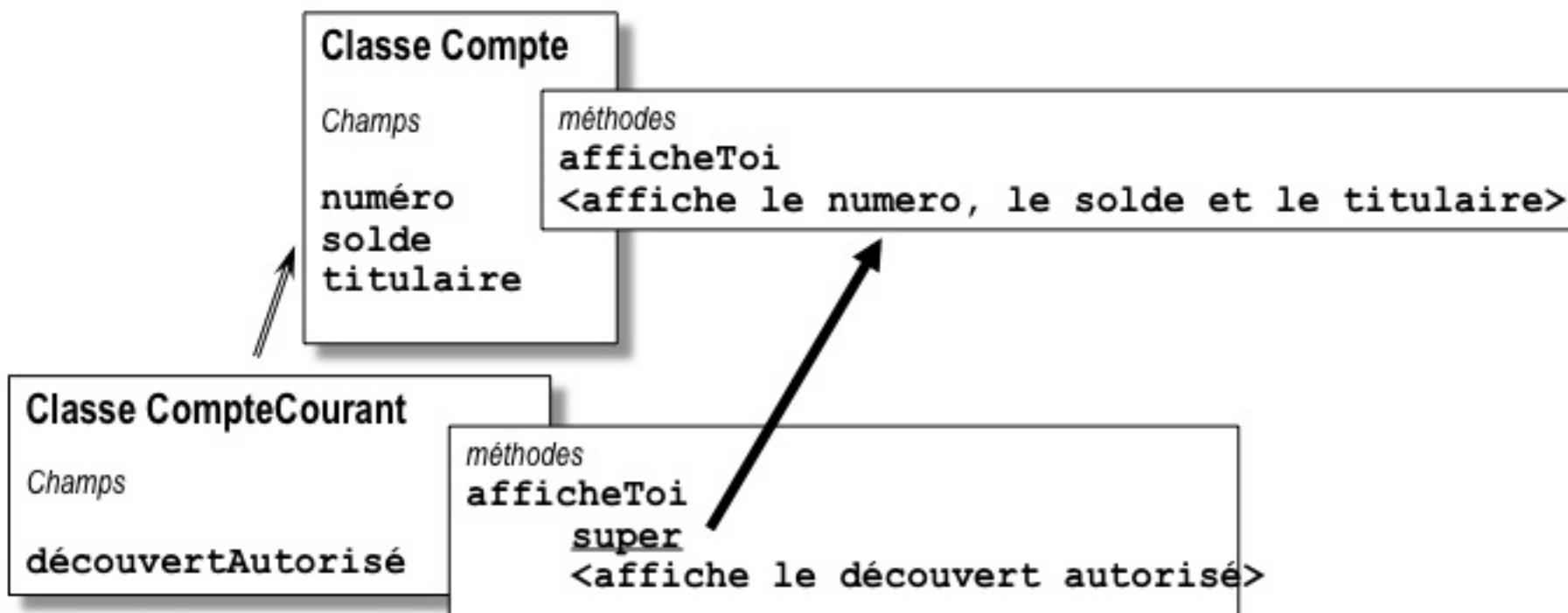
A26-Détermination de la méthode à appliquer

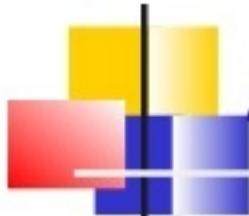
- Quand est message est reçu, l'interprète cherche dans la classe du receveur une méthode de même sélecteur
 - s'il la trouve il l'exécute
 - sinon : il cherche dans la superclasse du receveur
 - et ainsi de suite en remontant dans l'arbre d'héritage jusqu'à ce
 - qu'il en trouve une, et alors celle-ci est appliquée
 - qu'il n'en trouve aucune, et alors il envoie le message d'erreur à l'objet receveur
(ce qui déclenche une erreur d'exécution)



A27-Messages envoyés à super (1)

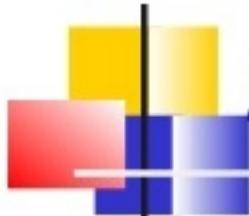
- **super** est une pseudo-variable qui permet de faire référence à une méthode de même nom définie dans la classe supérieure (mère) de celle du receveur du message





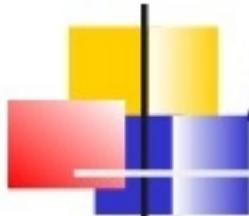
A27-Messages envoyés à super (2)

- Utilisation de **super**
 - Règle n°1 : **super** est utilisée pour redéfinir une méthode dans une sous-classe en spécialisant la méthode héritée tout en réutilisant le code déjà écrit
 - Règle n°2 : l'emploi de **super** prévient le lecteur du programme que la classe se décharge d'une part de sa responsabilité en déléguant à une de ses superclasses
- N'employez **super** que dans ces cas-là



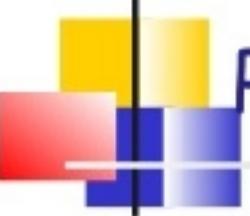
A3-Classes abstraites en POO (1)

- Une classe peut être une description partielle mais générale d'objets dont les détails seront spécifiés ultérieurement (par héritage)
- Une classe abstraite
 - déclare des champs et des comportements communs à plusieurs classes,
 - la définition précise des comportements étant de la responsabilité des sous-classes
- Une classe abstraite est une classe qui est définie pour être sous-classée pas pour être instanciée.



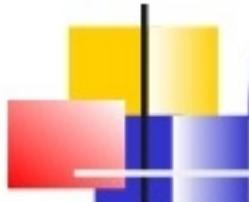
A3-Classes abstraites en POO (2)

- Une classe abstraite n'a pas d'instance mais est la racine d'une hiérarchie ;
 - elle ne définit pas (ou pas tous) les comportements des objets qu'elle décrit
- Une classe concrète est une classe qui a des instances et où sont définis tous les comportements des objets qu'elle décrit
- Souvent une classe abstraite
 - n'a pas d'attributs mais définit un ensemble d'opérations communes à toute la hiérarchie
 - fournit une interface commune sans exposer la réalisation



Plan

- A-Héritage
 - 1. Principe
 - 1. Exemple
 - 2. Lien d'héritage/d'instanciation
 - 2. Héritage
 - 1. Terminologie
 - 2. Utilisation
 - 3. Polymorphisme (Parenthèse)
 - 4. Programmation
 - 5. Héritage Statique
 - 6. Héritage Dynamique
 - 7. Message envoyé à super
 - 3. Classe Abstraite
- B-La hiérarchie des classes Ruby
- C-Exemple
- D-Conclusion

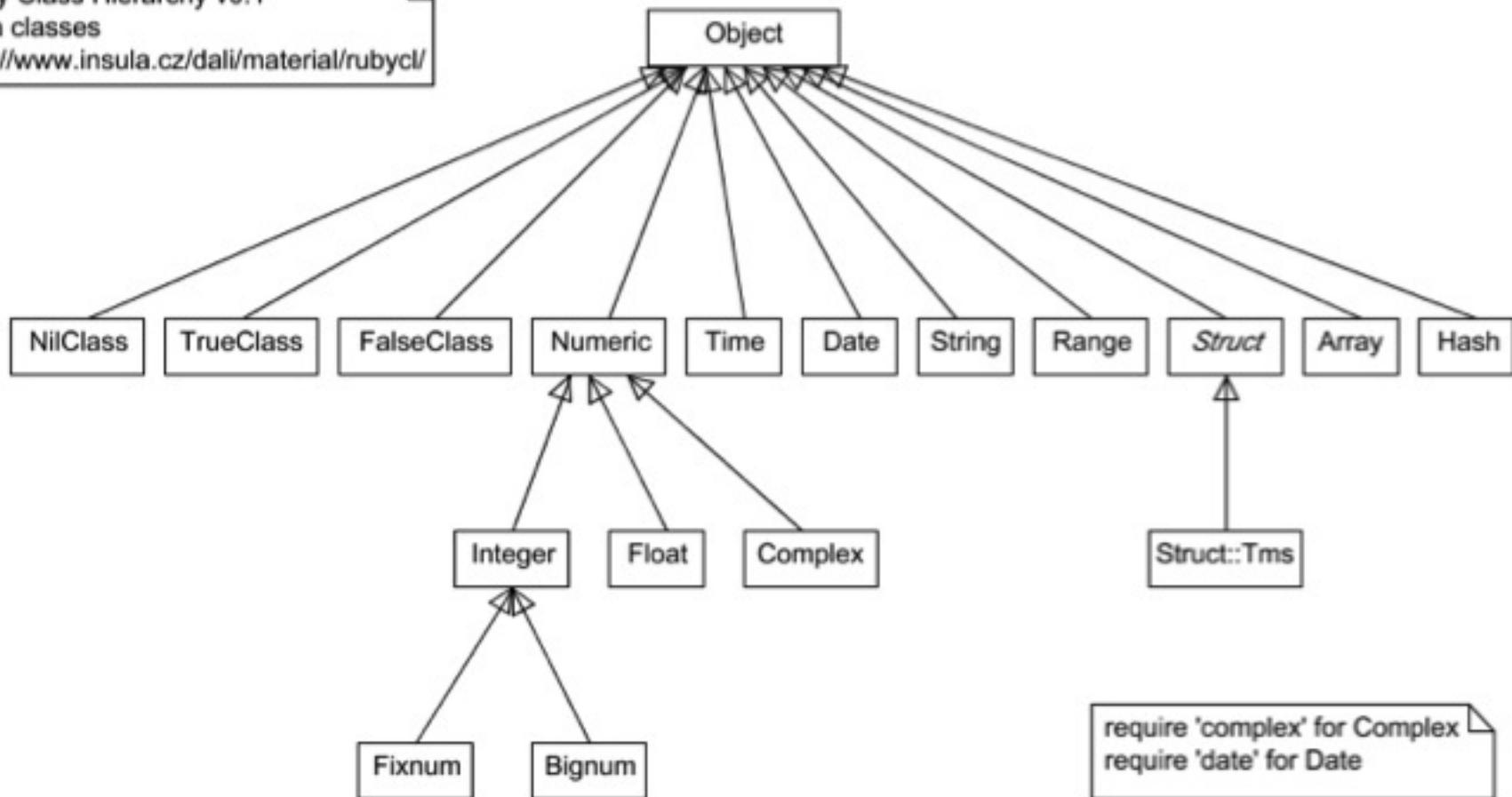


B-Hiérogarchie des classes Ruby

Ruby Class Hierarchy v0.1

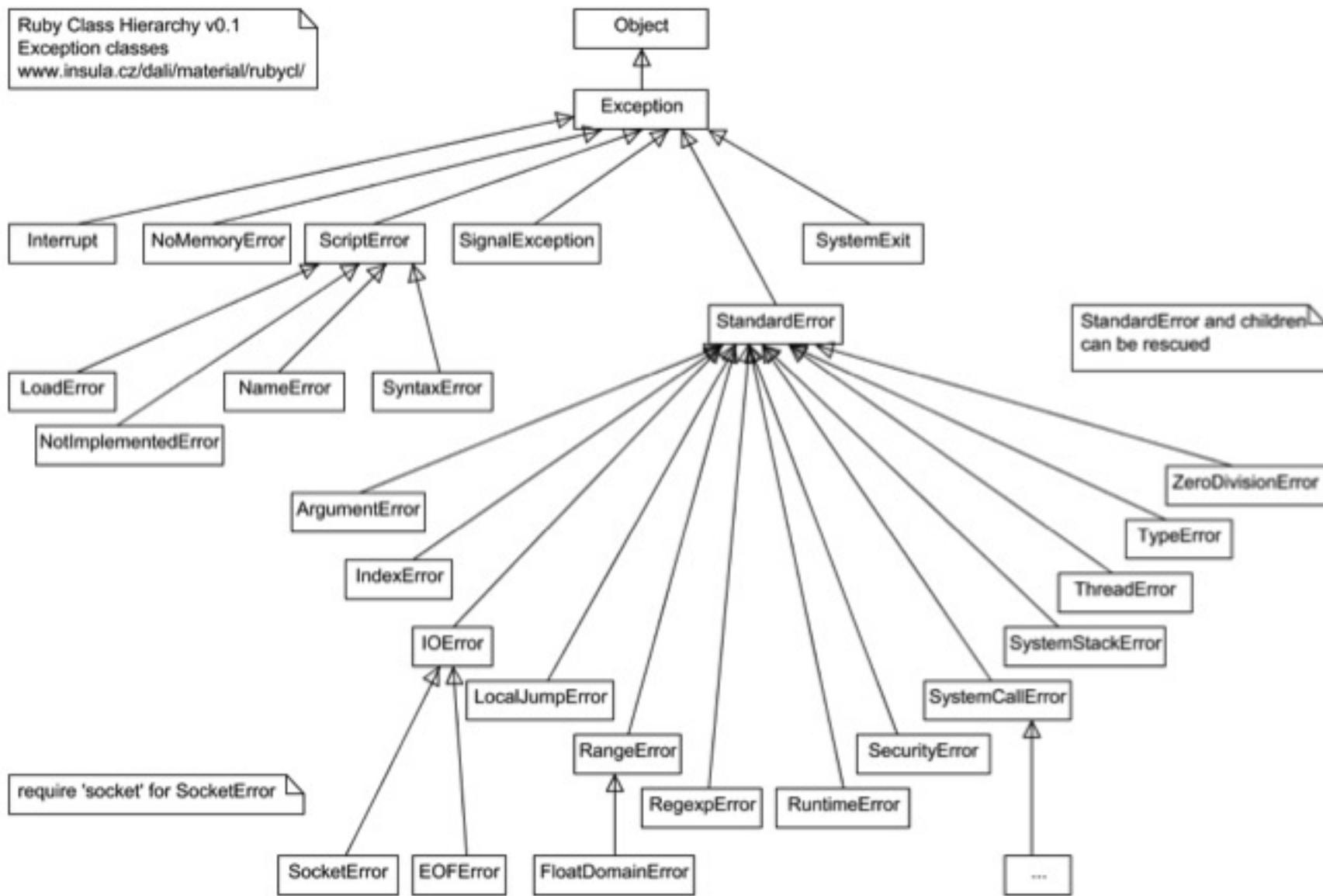
Data classes

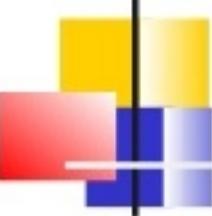
<http://www.insula.cz/dali/material/rubycl/>



require 'complex' for Complex
require 'date' for Date

Ruby Class Hierarchy v0.1
Exception classes
www.insula.cz/dali/material/rubycl/

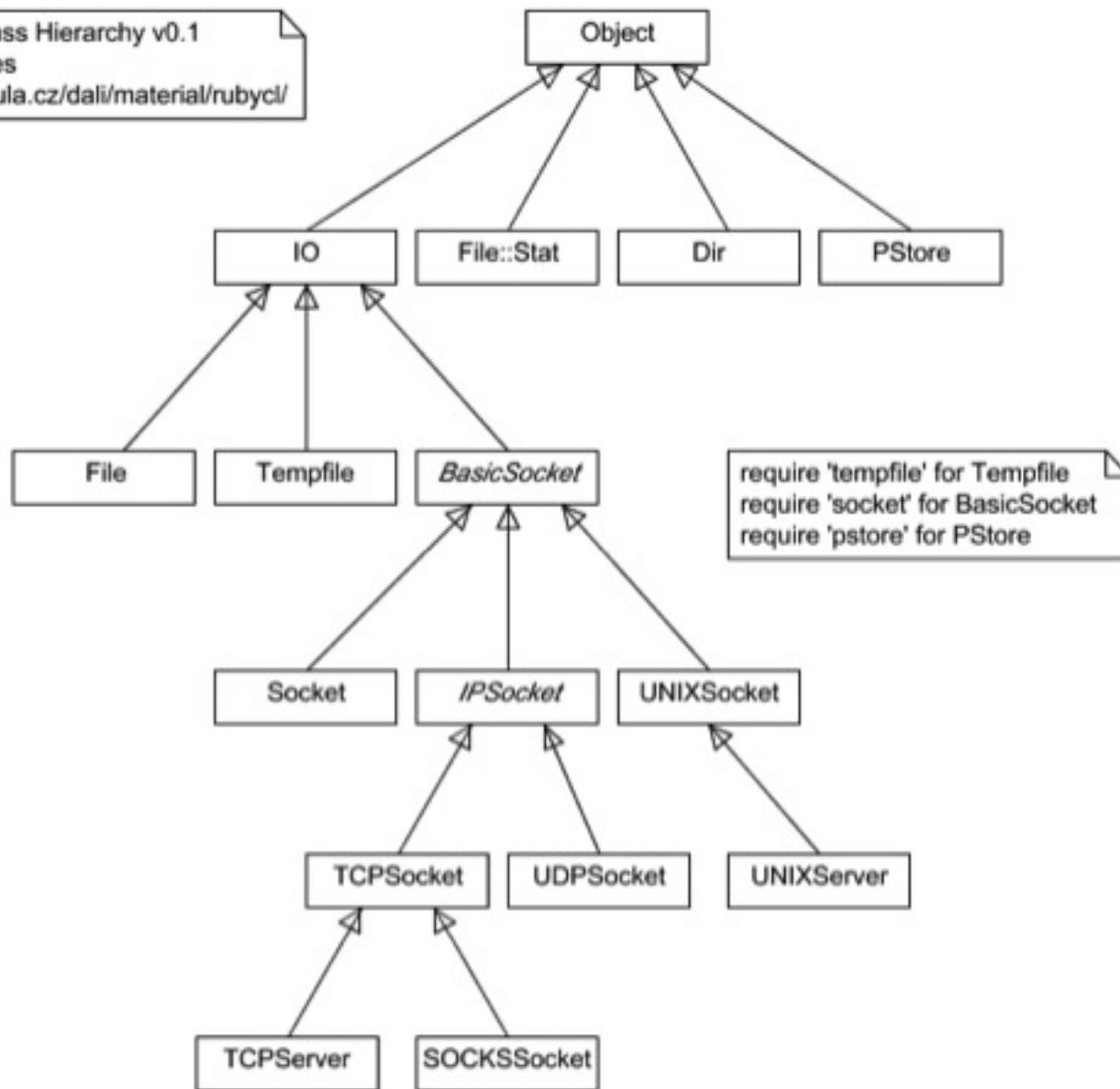




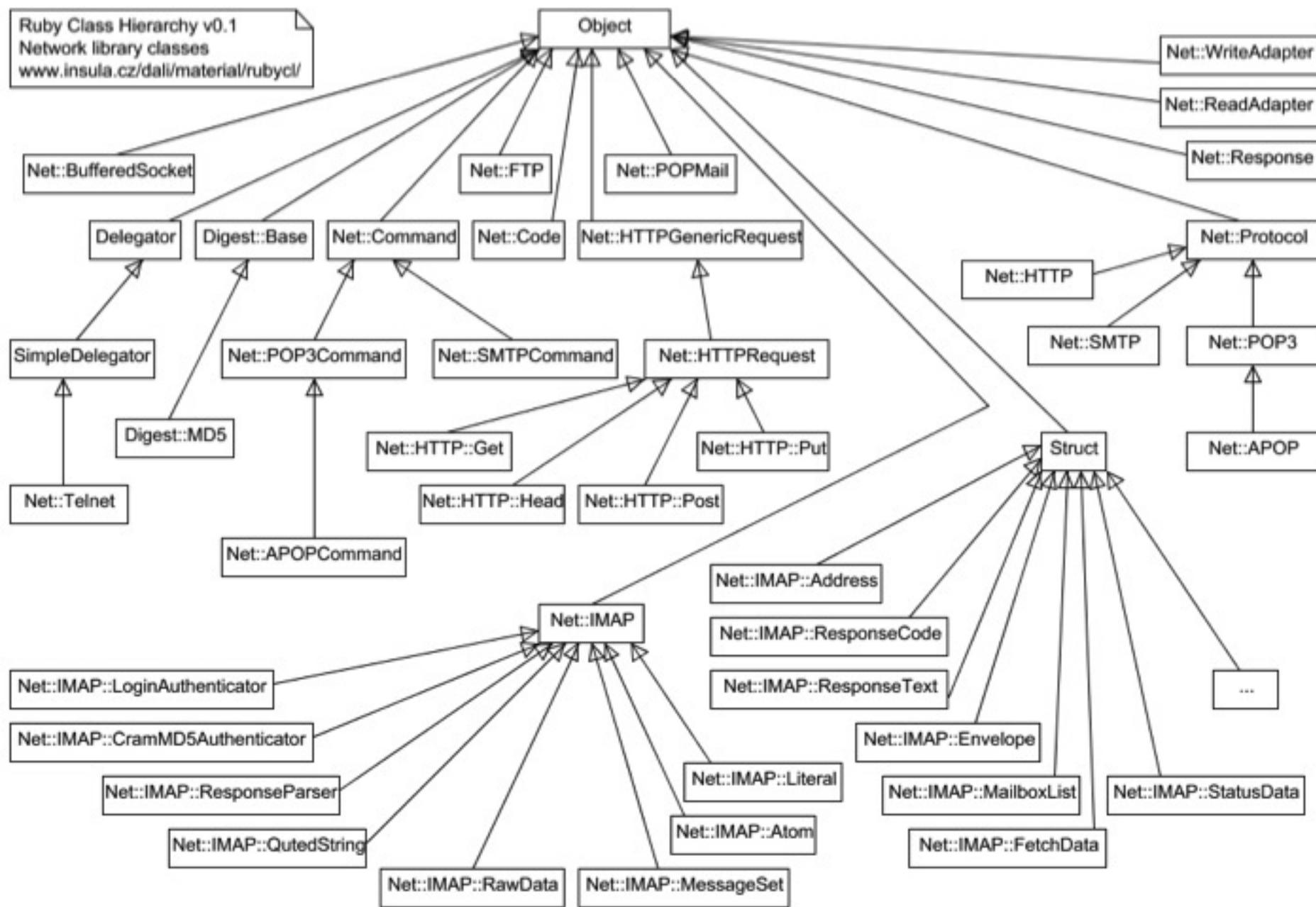
Ruby Class Hierarchy v0.1

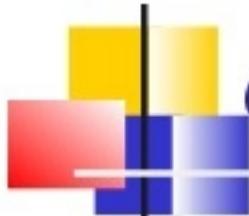
IO classes

www.insula.cz/dali/material/rubycli/



Ruby Class Hierarchy v0.1
Network library classes
www.insula.cz/dali/material/rubycl/





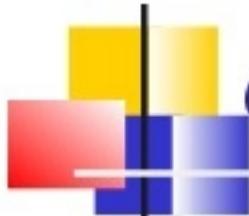
C-Exemple

```
class Chanson

    def initialize(nom, artiste, duree)
        @nom      = nom
        @artiste  = artiste
        @duree   = duree
    end

end

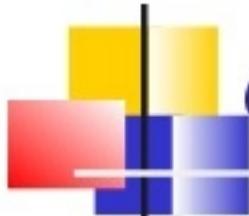
uneChanson = Chanson.new("Wish You Were Here",
                           "Pink Floyd", 340)
```



C-Affichage des objets

```
# La méthode inspect permet d'afficher des
# informations sur les objets
print uneChanson.inspect, "\n"
#<Chanson:0x2777918 @nom="Wish You Were Here",
@duree=340, @artiste="Pink Floyd">

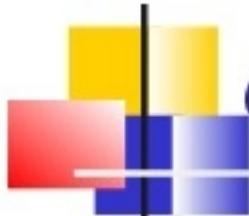
#la méthode to_s permet d'afficher une
# représentation sous forme de chaîne des objets
# Elle est appelée automatiquement lors de
# l'utilisation d'une méthode d'affichage
print uneChanson, "\n"
#<Chanson:0x2777918>
```



C-Affichage avec to_s

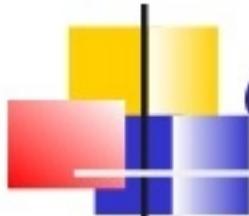
```
# Pour obtenir un affichage convenable il suffit
# donc de redéfinir to_s
class Chanson
  def to_s
    "# Chanson : #{@nom}--#{@artiste} (#{@duree})"
  end
end

print uneChanson, "\n"
# Chanson : Wish You Were Here--Pink Floyd (340)
```



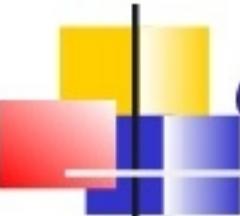
C-Une classe dérivée

```
class ChansonKaraoke < Chanson
    def initialize(nom, artiste, duree, texte)
        super(nom, artiste, duree)
        @paroles = texte
    end
end
autreChanson = ChansonKaraoke.new("My Way",
                                    "Sinatra", 225, "And now, the...")
print autreChanson, "\n"
# Chanson : My Way--Sinatra (225)
# Evidemment cela n'affiche pas les paroles, puisque la
# méthode to_s utilisée est celle définie dans la
# classe Chanson qui ne gère pas les paroles
```



C-Surcharge de to_s

```
# il faut donc la redéfinir  
# Solution 1 on redéfinit complètement la méthode  
class ChansonKaraoke  
  
  def to_s  
    "# Chanson : #{@nom}--#{@artiste} (#{@duree}) [#{@paroles}]"  
  end  
  
end  
  
  
# Maintenant cela marche  
print autreChanson, "\n"  
# Chanson : My Way--Sinatra (225) [And now, the...]
```



C-Utilisation de super

```
# Cette solution n'est pas bonne on ne tire pas
# partie de l'héritage. Il faut réutiliser ce qui
# est déjà écrit dans la classe supérieur
# Solution 2

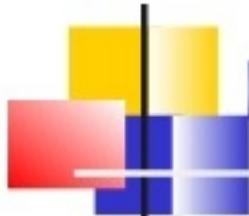
class ChansonKaraoke

  def to_s
    # On concatène (+) la chaîne produite par to_s
    # de la classe supérieur avec les paroles
    super + " #{@paroles}""
  end

end

print autreChanson, "\n"

# Chanson : My Way--Sinatra (225) [And now, the...]
```



D-Conclusion : Héritage en POO

Plusieurs points de vue

- Ensembliste : inclusion
- Type : notion de sous-type et d'héritage dynamique des méthodes
- Style de programmation
 - factorisation du code
 - programmation par raffinements successifs