Partie III

La programmation orientée objets

Informatique 1
Introduction à la programmation

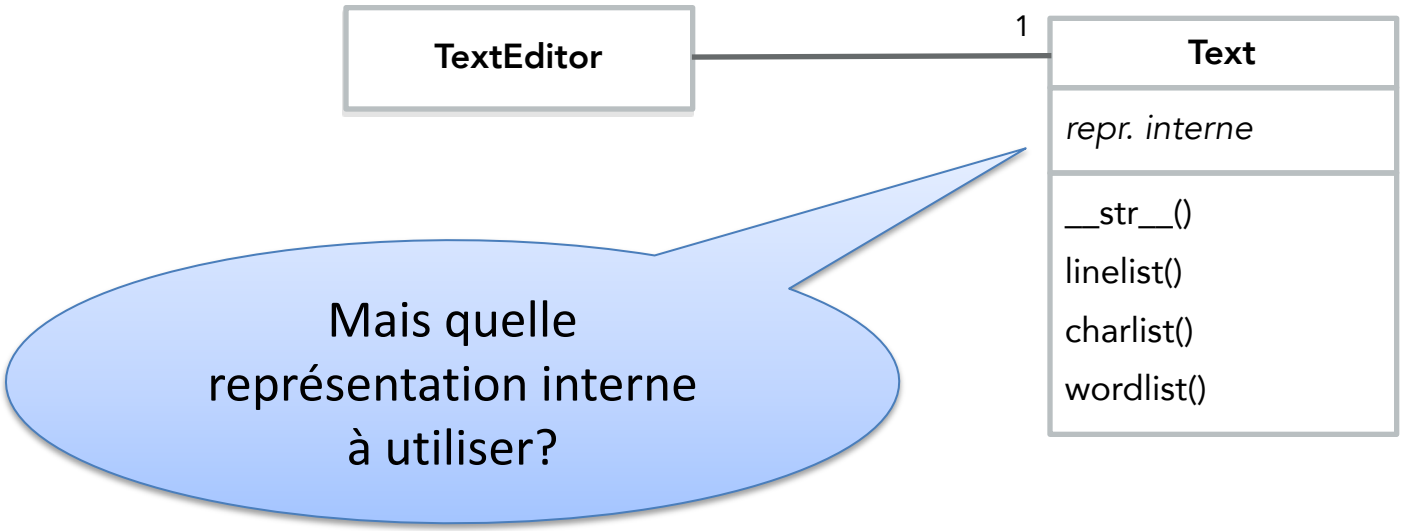
Mission 10 : INTRODUCTION

La programmation orientée objets : le polymorphisme

Kim Mens – Siegfried Nijssen – Charles Pecheur

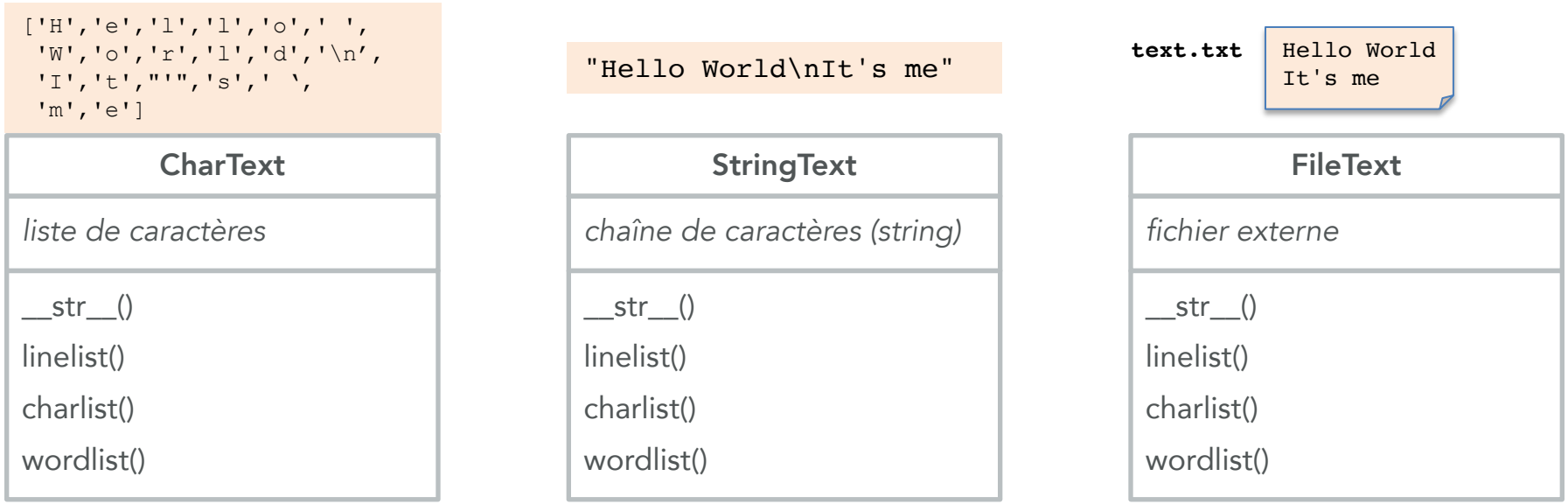
Un éditeur de texte

On veut implémenter un éditeur de texte :



Un éditeur de texte

Quelle représentation interne à utiliser?



Peu importe, pour autant que les classes implémentent la même *interface publique*

(mêmes méthodes et attributs publics)

Polymorphisme !

Le polymorphisme

Le mot « **polymorphisme** » vient du grec

et signifie « *qui peut prendre plusieurs formes* »

Le **polymorphisme** en programmation orientée objets :

- des méthodes de mêmes noms peuvent avoir des comportements différents ou effectuer des opérations sur des données de types différents
- des classes ayant des méthodes de mêmes noms peuvent avoir avoir des implémentations différents, ou des attributs privés de types différents

Le polymorphisme « ad-hoc »

Le polymorphisme ad hoc :

- des objets de classes différentes et indépendantes, mais qui implémentent la même interface publique (même méthodes et attributs publics) sont interchangeables

Grace au typage dynamique de Python

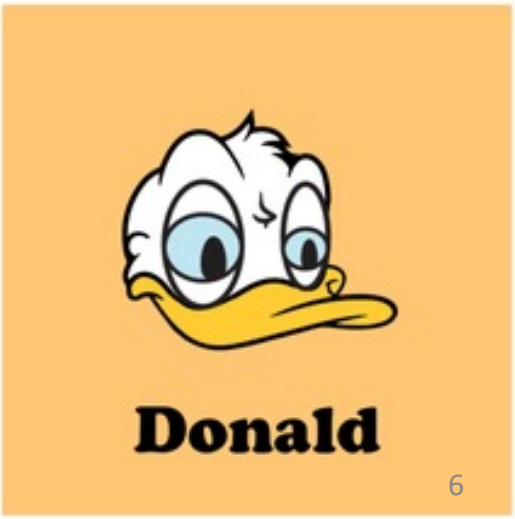
- alias « Duck Typing » (le typage canard) : la sémantique d'un objet est déterminée par l'ensemble de ses méthodes et ses attributs, et non par un type défini et nommé explicitement par le programmeur
- *"If it walks like a duck and it quacks like a duck, then it must be a duck"*

Le typage canard

```
class Duck :
  def __init__(self,nom):
    self.nom = nom
  def talk(self):
    return "Quack, quack!"
  def fly(self):
    return "Flap, flap!"
  def __str__(self):
    return self.talk() + \
           " " + self.fly()
```

```
class Interview :
  def __init__(self,thing):
    self.t = thing
  def __str__(self):
    return self.t.nom + \
           ': "' + \
           str(self.t) + '"'
```

```
>>> donald = Duck("Donald Duck")
>>> interview = Interview(donald)
>>> print(interview)
Donald Duck: "Quack, quack! Flap, flap!"
```



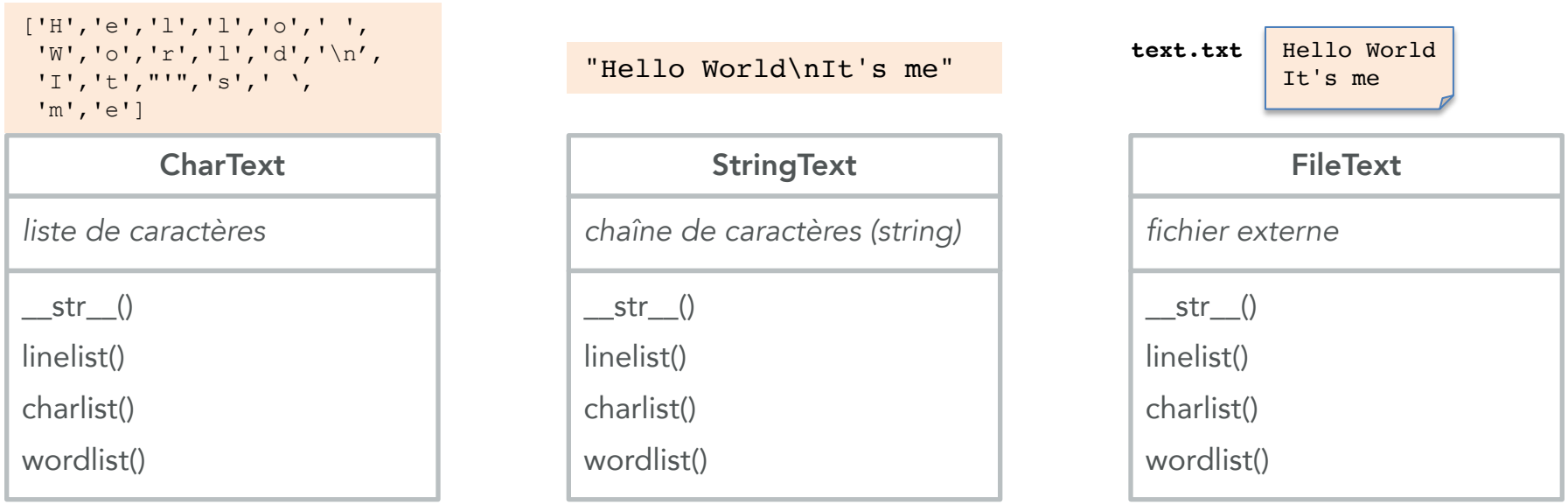
Le typage canard

```
class Person :

  def __init__(self,nom) :
    self.nom = nom

  def talk(self) :
    return "I won the elections

  def fly(self) :
    return "I'm flying."

  def __str__(self):
    return "I'm " + self.nom + \
           ". " + self.talk() + \
           " " + self.fly()
```

```
class Interview :

  def __init__(self,thing):
    self.t = thing

  def __str__(self):
    return self.t.nom + \
           ': "' + \
           str(self.t) + '"'
```

```
>>> donald = Person("Donald Trump")
>>> interview = Interview(donald)
>>> print(interview)
Donald Trump: "I'm Donald Trump. I won
the elections. I'm flying."
```

Le polymorphisme « ad hoc »



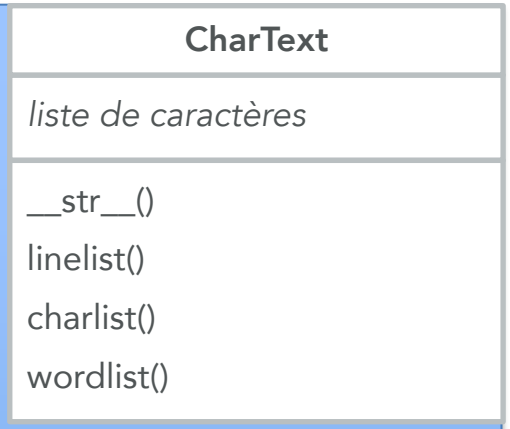
Le polymorphisme « ad hoc »

```
class StringText :

  def __init__(self,string) :
    self.__textstring = string

  def textstring(self) :
    return self.__textstring

  def __str__(self) :
    return self.textstring()

  def linelist(self) :
    return self.textstring().split("\n")

  def charlist(self) :
    return list(self.textstring())

  def wordlist(self) :
    return self.textstring().replace("\n"," ")  \
           .split(" ")
```

StringText
chaîne de caractères (string)
__str__() linelist() charlist() wordlist()

Le polymorphisme « ad hoc »

```
class CharText :

  def __init__(self,charlist) :
    self.__charlist = charlist



  def __str__(self) :
    return "".join(self.charlist())

  def linelist(self) :
    return str(self).split("\n")

  def charlist(self) :
    return self.__charlist

  def wordlist(self) :
    return str(self).replace("\n"," ").split(" ")
```



Le polymorphisme « ad hoc »

```
class FileText :
  def __init__(self,filename) :
    self.__filename = filename
  def filename(self) :
    return self.__filename
  def __str__(self) :
    …
  def linelist(self) :
    … # read from file
  def charlist(self) :
    …
  def wordlist(self) :
    … # read from file
```

FileText
fichier externe
__str__() linelist() charlist() wordlist()

```
with open(self.filename(),'r') as f:
  lines = f.readlines()
return "".join(lines)
```

```
lst = []
with open(self.filename(),'r') as f:
  while True:
    c = f.read(1)
    if not c: # end of line
      break
    lst.append(c)
return lst
```

Tester le code avec des asserts

```
from CharText import CharText
from StringText import StringText
from FileText import FileText

ct = CharText(['H','e','l','l','o',' ','W','o','r','l','d','\n',
               'I', 't', "'", 's', ' ', 'm', 'e'])
st = StringText("Hello World\nIt's me")
ft = FileText("text.txt")

assert str(ct) == str(st) == str(ft) == "Hello World\nIt's me"

assert ct.linelist() == st.linelist() == ft.linelist()
       == ['Hello World', "It's me"]

assert ct.wordlist() == st.wordlist() == ft.wordlist()
       == ['Hello', 'World', "It's", 'me']

assert ct.charlist() == st.charlist() == ft.charlist()
       == ['H','e','l','l','o',' ','W','o','r','l','d','\n',
           'I','t',"'",'s',' ','m','e']
```

Tester le code avec une classe test

```
import unittest
from CharText import CharText

class CharTextTest(unittest.TestCase):

  def setUp(self):
    self.text_string = "Hello World\nIt's me"
    self.line_list   = ['Hello World', "It's me"]
    self.word_list   = ['Hello', 'World', "It's", 'me']
    self.char_list   = ['H','e','l','l','o',' ','W','o','r','l','d','\n',
                        'I','t',"'",'s',' ','m','e']
    self.t = CharText(self.char_list)

  def test_str(self) :
    self.assertEqual(str(self.t), self.text_string)

  def test_charlist(self) :
    self.assertEqual(self.t.charlist(), self.char_list)

  def test_wordlist(self) :
    self.assertEqual(self.t.wordlist(), self.word_list)

  def test_linelist(self) :
        self.assertEqual(self.t.linelist(), self.line_list)

if __name__ == '__main__':
    unittest.main()
```

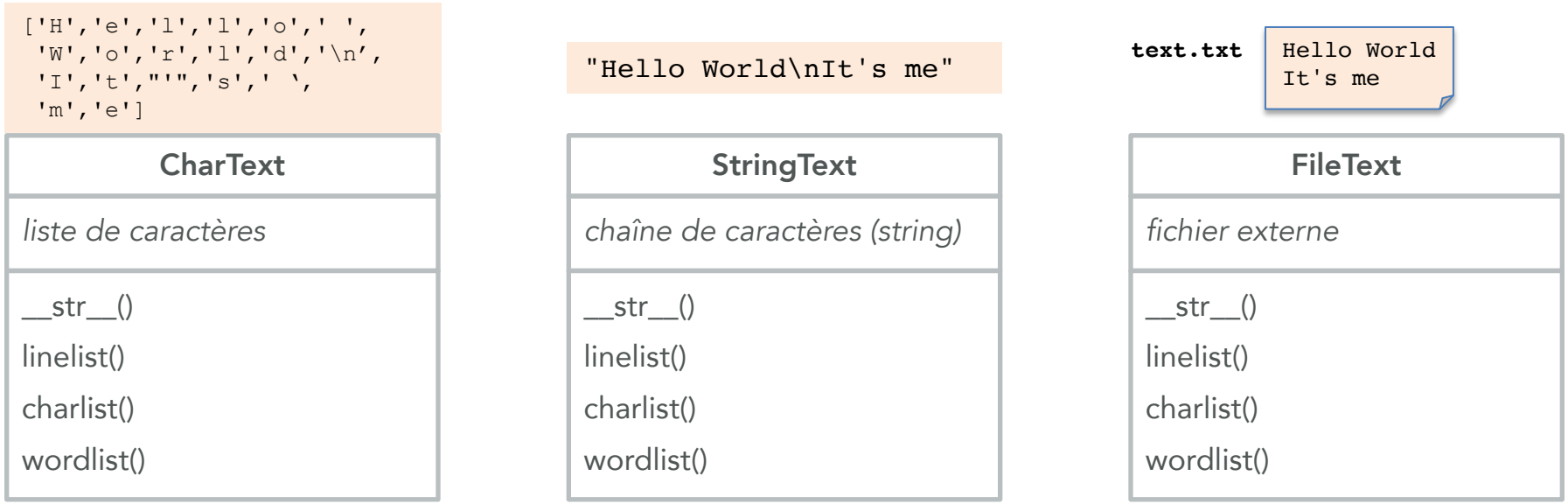
Les tests unitaires

Structure d'un test unitaire pour tester une autre classe

```
import unittest
from CharText import CharText

class CharTextTest(unittest.TestCase):

  def setUp(self):
    self.text_string = "Hello World\nIt's me"
    self.line_list   = ['Hello World', "It's me"]
    self.word_list   = ['Hello', 'World', "It's", 'me']
    self.char_list   = ['H','e','l','l','o',' ','W','o','r','l','d','\n',
                        'I','t',"'",'s',' ','m','e']
    self.t = CharText(self.char_list)

  def test_str(self) :
    self.assertEqual(str(self.t), self.text_string)

  def test_charlist(self) :
    self.assertEqual(self.t.charlist(), self.char_list)

  def test_wordlist(self) :
    self.assertEqual(self.t.wordlist(), self.word_list)

  def test_linelist(self) :
        self.assertEqual(self.t.linelist(), self.line_list)

if __name__ == '__main__':
    unittest.main()
```

Les tests unitaires

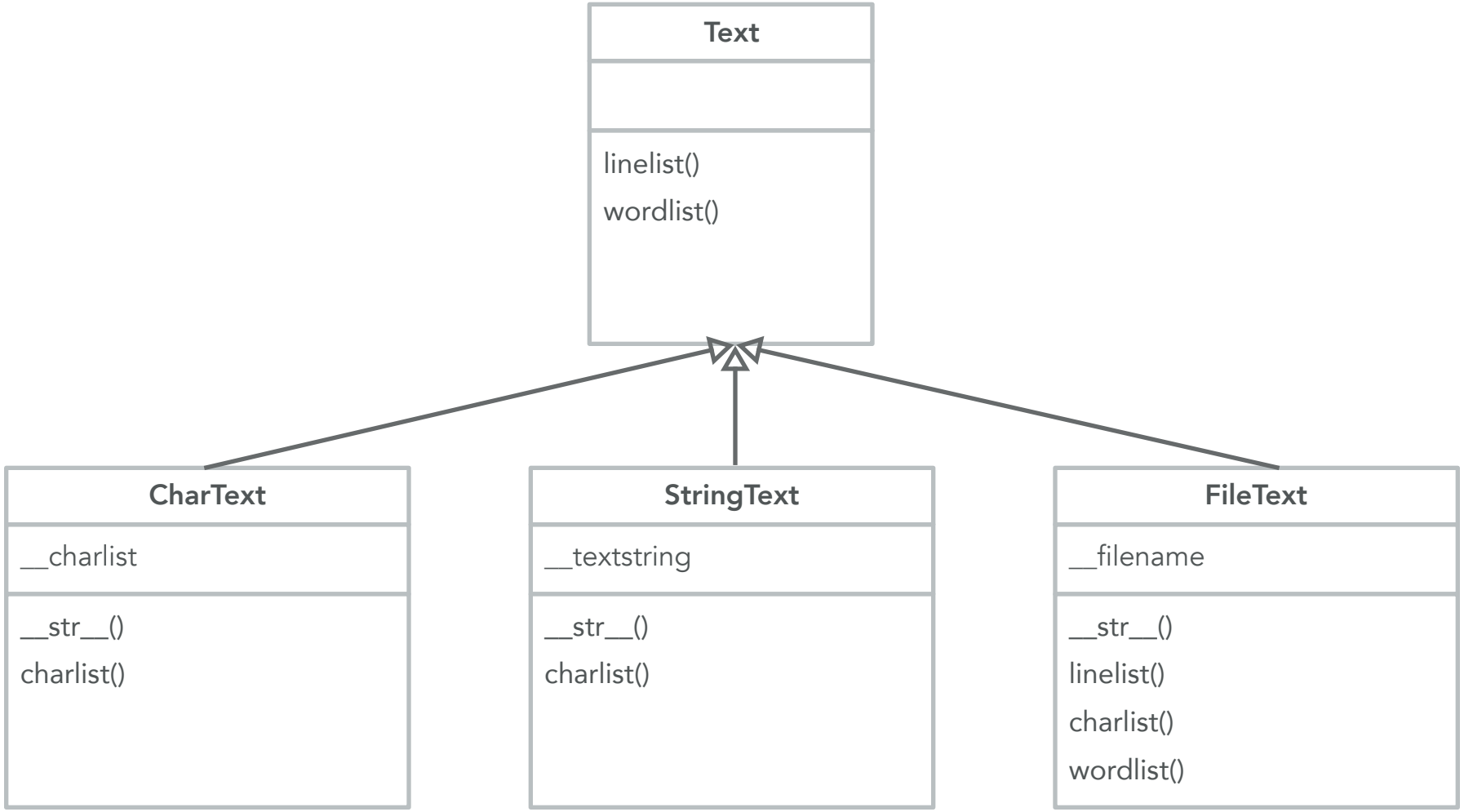
Le polymorphisme « ad hoc »



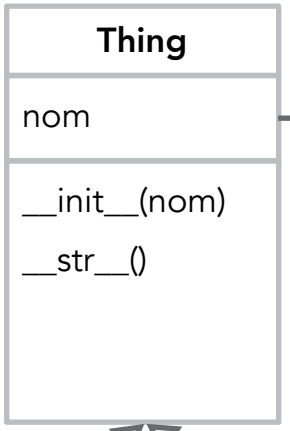
Le polymorphisme par héritage

- La notion du polymorphisme est très liée à celle d'héritage.
- Grâce à la redéfinition, il est possible de redéfinir une méthode dans des classes héritant d'une classe mère.
- Par ce mécanisme, une classe fille peut modifier le comportement de certaines méthodes héritées pour être adaptées aux besoins de la classe fille.
- On peut aussi mettre des méthodes communes à différentes classes filles dans la classe mère.
- A chaque endroit où un objet de la classe mère est attendu, on peut utiliser un objet d'une de ces classes filles (interchangeabilité).

Le polymorphisme par héritage



Le polymorphisme par héritage



1

```
class Interview :

  def __init__(self,thing):
    self.t = thing

  def __str__(self):
    return self.t.nom + \
           ': "' + \
           str(self.t) + '"'
```

Duck

Person

Le polymorphisme par héritage

```
class Thing :

  def __init__(self,nom):
    self.nom = nom

  def __str__(self):
    return self.talk() + \
           " " + self.fly()
```

```
class Duck(Thing) :

  def talk(self) :
    return "Quack, quack!"

  def fly(self) :
    return "Flap, flap!"
```

Ajout de méthodes

```
class Person(Thing) :

 def talk(self) :
    return "I won the elections."
 def fly(self) :
    return "I'm flying..."

 def __str__(self):
    return "I'm " + self.nom \
           + ". " + \
           super().__str__()
```

Redéfinition de méthode

Le polymorphisme par héritage

```
class Thing :

  def __init__(self,nom):
    self.nom = nom

  def __str__(self):
    return self.talk() + \
           " " + self.fly()
```

```
class Duck(Thing) :

  def talk(self) :
    return "Quack, quack!"

  def fly(self) :
    return "Flap, flap!"
```

```
class Interview :

  def __init__(self,thing):
    self.t = thing

  def __str__(self):
    return self.t.nom + \
           ': "' + \
           str(self.t) + '"'
```

```
>>> donald = Duck("Donald Duck")
>>> interview = Interview(donald)
>>> print(interview)
Donald Duck: "Quack, quack! Flap,
flap!"
```

Le polymorphisme par héritage

```
class Thing :

  def __init__(self,nom):
    self.nom = nom

  def __str__(self):
    return self.talk() + \
           " " + self.fly()
```

```
class Interview :

  def __init__(self,thing):
    self.t = thing

  def __str__(self):
    return self.t.nom + \
           ': "' + \
           str(self.t) + '"'
```

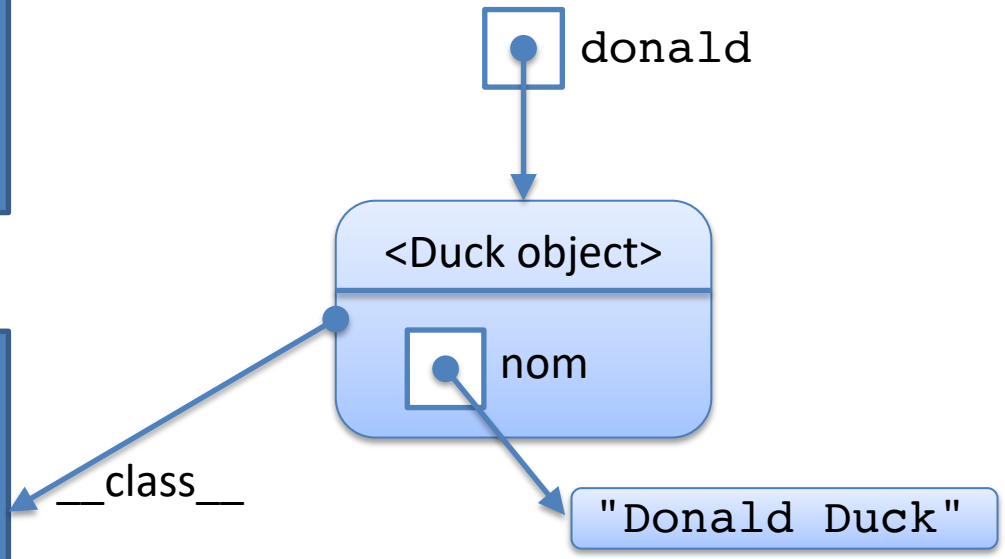
```
class Person(Thing) :

 def talk(self) :
    return "I won the elections."
  def fly(self) :
    return "I'm flying..."

  def __str__(self):
    return "I'm " + self.nom \
           + ". " + \
           super().__str__()
```

```
>>> donald = Person("Donald Trump")
>>> interview = Interview(donald)
>>> print(interview)
Donald Trump: "I'm Donald Trump. I
won the elections. I'm flying."
```

La liaison dynamique

```
class Thing :

  def __init__(self,nom):
    self.nom = nom

  def __str__(self):
    return self.talk() + \
           " " + self.fly()
```

```
class Duck(Thing) :

  def talk(self) :
    return "Quack, quack!"

  def fly(self) :
    return "Flap, flap!"
```




```
>>> donald = Duck("Donald Duck")
```

La résolution de méthode

```
class Thing :
  def __init__(self,nom):
    self.nom = nom
  def __str__(self):
    return self.talk() + \
           " " + self.fly()
```

```
class Duck(Thing) :
  def talk(self) :
    return "Quack, quack!"
  def fly(self) :
    return "Flap, flap!"
```

La résolution de méthodes :

- **On cherche la méthode à exécuter d'abord dans la classe de l'objet auquel le message a été envoyé.**
- Si pas si pas trouvé on cherche dans la classe mère, et ainsi de suite.
- Si la méthode n'existe pas dans cette hierarchie, on obtiendra une erreur du type AttributeError

```
message = __str__()
self = donald
```

```
>>> donald = Duck("Donald Duck")

>>> donald.__str__()
```

La liaison dynamique

```
class Thing :
  def __init__(self,nom):
    self.nom = nom
  def __str__(self):
    return self.talk() + \
           " " + self.fly()
```

```
class Duck(Thing) :
  def talk(self) :
    return "Quack, quack!"
  def fly(self) :
    return "Flap, flap!"
```

```
message = __str__()
self = donald
```

Même si on trouve la méthode dans la classe mère

- self refère toujours à l'objet auquel le message initial a été envoyé
- ceci permet d'exécuter la méthode comme si elle était définie dans la classe fille

```
>>> donald = Duck("Donald Duck")

>>> donald.__str__()
```

La liaison dynamique

```
class Thing :
  def __init__(self,nom):
    self.nom = nom
  def __str__(self):
    return self.talk() + \
           " " + self.fly()
```

```
class Duck(Thing) :
  def talk(self) :
    return "Quack, quack!"
  def fly(self) :
    return "Flap, flap!"
```

on exécute la méthode pour
self = donald

```
return self.talk() + \
           " " + self.fly()
```

```
>>> donald = Duck("Donald Duck")
>>> donald.__str__()
```

La liaison dynamique

```
class Thing :

  def __init__(self,nom):
    self.nom = nom

  def __str__(self):
    return self.talk() + \
           " " + self.fly()
```

```
class Duck(Thing) :

  def talk(self) :
    return "Quack, quack!"

  def fly(self) :
    return "Flap, flap!"
```

```
return donald.talk() + \
           " " + donald.fly()
```

```
message = talk()
self = donald
return "Quack, quack!"
```

```
>>> donald = Duck("Donald Duck")

>>> donald.__str__()
```

La liaison dynamique

```
class Thing :

  def __init__(self,nom):
    self.nom = nom

  def __str__(self):
    return self.talk() + \
           " " + self.fly()
```

```
class Duck(Thing) :

  def talk(self) :
    return "Quack, quack!"

  def fly(self) :
    return "Flap, flap!"
```

```
return "Quack, quack!" + \
           " " + donald.fly()
```

```
message = fly()
self = donald
return "Flap, flap!"
```

```
>>> donald = Duck("Donald Duck")

>>> donald.__str__()
```

La liaison dynamique

```
class Thing :
  def __init__(self,nom):
    self.nom = nom
  def __str__(self):
    return self.talk() + \
           " " + self.fly()
```

```
class Duck(Thing) :
  def talk(self) :
    return "Quack, quack!"
  def fly(self) :
    return "Flap, flap!"
```

```
return "Quack, quack! " + \
           " " + "Flap, flap!"
```

```
>>> donald = Duck("Donald Duck")

>>> donald.__str__()

"Quack, quack! Flap, flap!"
```

La sémantique de super()

```
class Thing :

  def __init__(self,nom):
    self.nom = nom

  def __str__(self):
    return self.talk() + \
           " " + self.fly()
```

```
class Person(Thing) :

 def talk(self) :
    return "I won the elections."
  def fly(self) :
    return "I'm flying..."

  def __str__(self):
    return "I'm " + self.nom \
           + ". " + \
           super().__str__()
```

`super( ).__str__(…)` veut dire :

- ne cherchez pas l'implementation de la méthode `__str__(…)` pas dans cette classe
- mais cherchez-la dans la classe mère de cette classe
- (ou encore plus haut dans l'hiérarchie)

```
donald = Person("Donald Trump")

donald.__str__()
```

La sémantique de super()

```
class Thing :

  def __init__(self,nom):
    self.nom = nom

  def __str__(self):
    return self.talk() + \
           " " + self.fly()
```

```
class Person(Thing) :

 def talk(self) :
    return "I won the elections."
  def fly(self) :
    return "I'm flying..."

 def __str__(self):
    return "I'm " + self.nom \
           + ". " + \
           super().__str__()
```

```
message = __str__()
self = donald
```

```
return "I'm Donald Trump. "
       + super().__str__()
```

```
donald = Person("Donald Trump")

donald.__str__()
```

La sémantique de super()

```
class Thing :

  def __init__(self,nom):
    self.nom = nom

  def __str__(self):
    return self.talk() + \
           " " + self.fly()
```

on exécute la méthode pour
self = donald

```
message = __str__()
self = donald
```

```
return self.talk() + \
           " " + self.fly()
```

```
class Person(Thing) :

 def talk(self) :
    return "I won the elections."
  def fly(self) :
    return "I'm flying..."

  def __str__(self):
    return "I'm " + self.nom \
           + ". " + \
           super().__str__()
```

```
return "I'm Donald Trump. "
       + super().__str__()
```

```
donald = Person("Donald Trump")

donald.__str__()
```

La sémantique de super()

```
class Thing :

  def __init__(self,nom):
    self.nom = nom

  def __str__(self):
    return self.talk() + \
           " " + self.fly()
```

```
return donald.talk() + \
           " " + donald.fly()
```

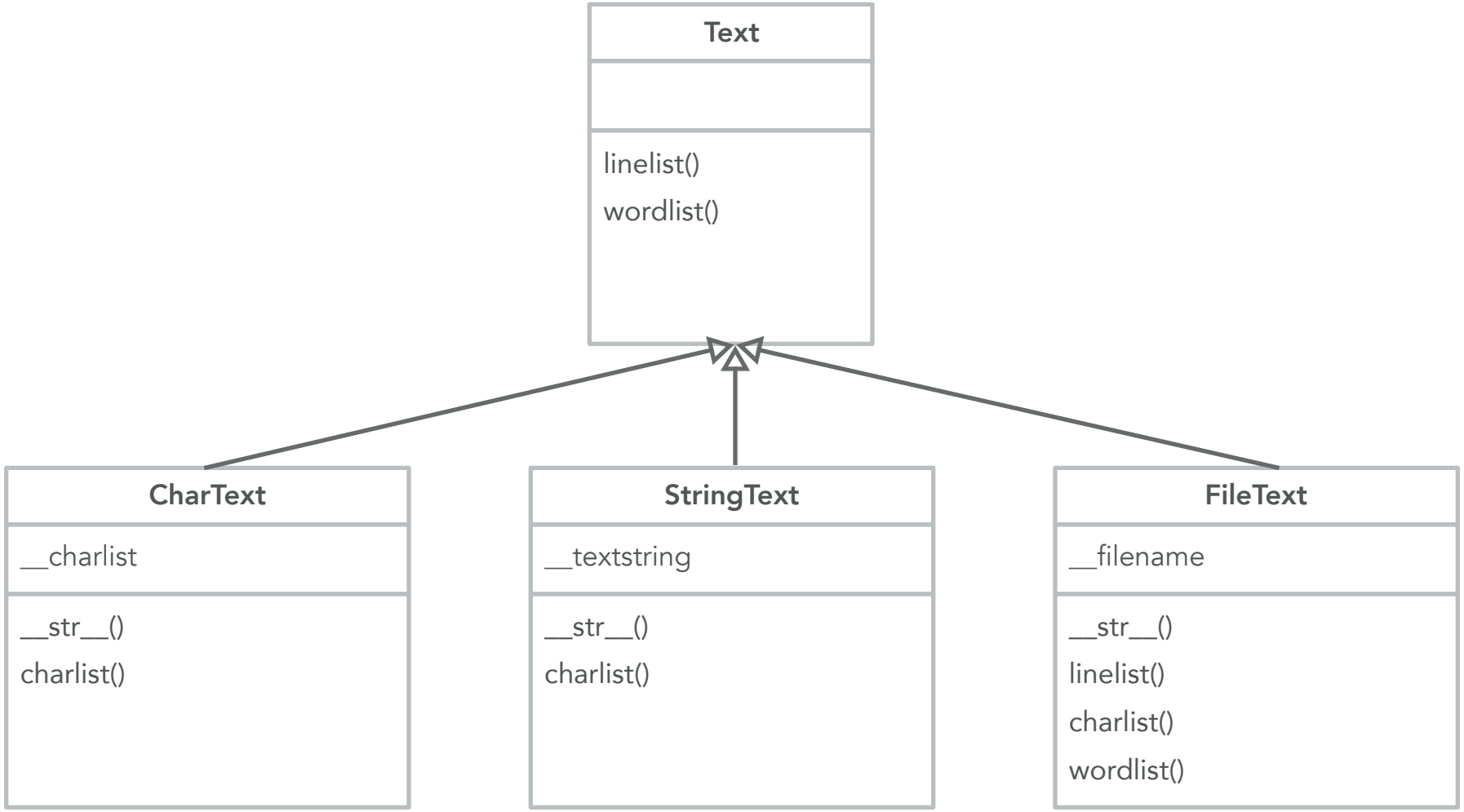
"I won the elections." +
" " + "I'm flying..."

```
class Person(Thing) :

 def talk(self) :
    return "I won the elections."
 def fly(self) :
    return "I'm flying..."

 def __str__(self):
    return "I'm " + self.nom \
           + ". " + \
           super().__str__()
```

```
return "I'm Donald Trump. "
       + super().__str__()
```

```
donald = Person("Donald Trump")

donald.__str__()

"I'm Donald Trump. I won the
     elections. I'm flying..."
```

Le polymorphisme par héritage



La classe StringText (v1.0)

```
class StringText :

  def __init__(self,string) :
    self.__textstring = string

  def textstring(self) :
    return self.__textstring

  def __str__(self) :
    return self.textstring()

  def linelist(self) :
    return self.textstring().split("\n")

  def charlist(self) :
    return list(self.textstring())

  def wordlist(self) :
    return self.textstring().replace("\n"," ")  \
           .split(" ")
```

StringText
chaîne de caractères (string)
__str__() linelist() charlist() wordlist()

La classe StringText (v1.1)

```
class StringText :

  def __init__(self,string) :
    self.__textstring = string

  def textstring(self) :
    return self.__textstring

  def __str__(self) :
    return self.__textstring

  def linelist(self) :
    return str(self).split("\n")

  def charlist(self) :
    return list(str(self))

  def wordlist(self) :
    return str(self).replace("\n"," ").split(" ")
```

StringText
chaîne de caractères (string)
__str__() linelist() charlist() wordlist()

La classe CharText (v1.0)

```
class CharText :

  def __init__(self,charlist) :
    self.__charlist = charlist



  def __str__(self) :
    return "".join(self.charlist())

  def linelist(self) :
    return str(self).split("\n")

  def charlist(self) :
    return self.__charlist

  def wordlist(self) :
    return str(self).replace("\n"," ").split(" ")
```

CharText
liste de caractères
__str__()
linelist()
charlist()
wordlist()

La classe Text

```
class Text :
```

charlist() et __init__()
à ajouter par les classes filles

Text
__str__() linelist() wordlist()

```
  def __str__(self) :
    return ""
```

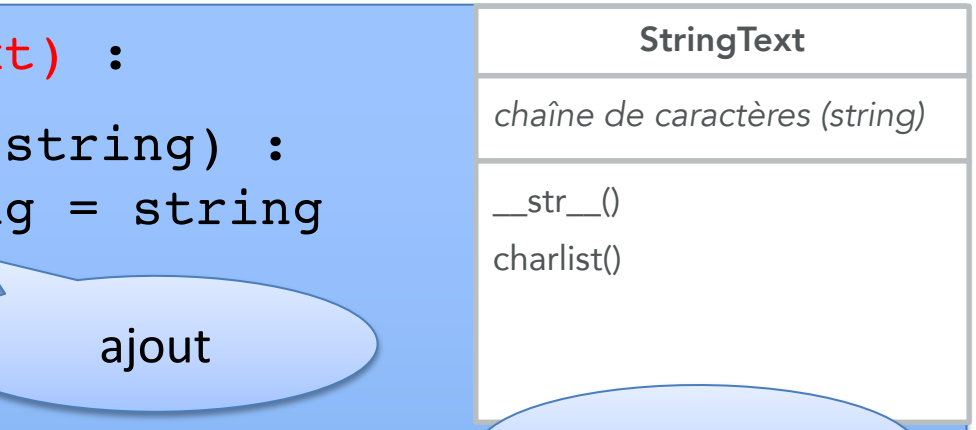
Implémentation par défaut à redéfinir par les classes filles

```
  def linelist(self) :
    return str(self).split("\n")
```

```
  def wordlist(self) :
    return str(self).replace("\n"," ").split(" ")
```

La classe StringText (v2.0)

```
class StringText(Text) :

  def __init__(self,string) :
    self.__textstring = string

  def __str__(self) :
    return self.__textstring

  def charlist(self) :
    return list(str(self))
```



ajout

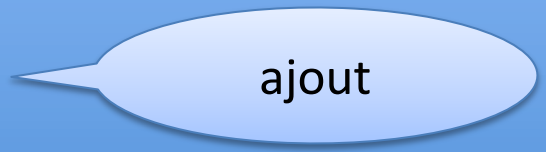
redéfinition

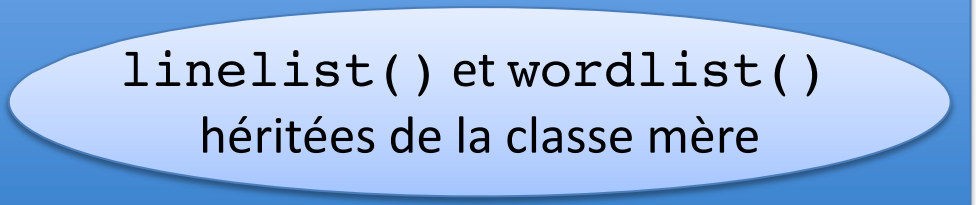
ajout

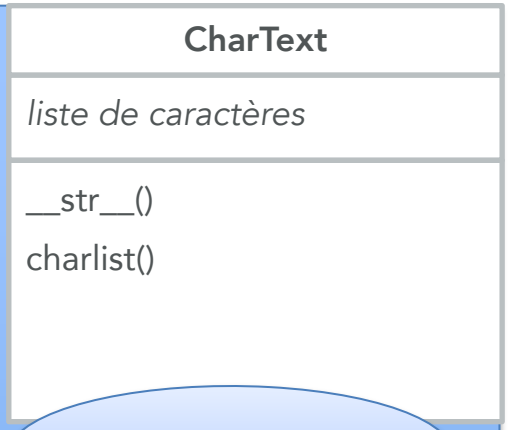
`linelist()` et `wordlist()` héritées de la classe mère

La classe CharText (v2.0)

```
class CharText(Text) :

  def __init__(self,charlist) :
    self.__charlist = charlist

  def __str__(self) :
    return "".join(self.charlist())

  def charlist(self) :
    return self.__charlist
```







La classe FileText (v2.0)

```
class FileText(Text) :
  def __init__(self,filename) :
    self.__filename = filename
  def filename(self) :
    return self.__filename
  def __str__(self) :
    …
  def linelist(self) :
    … # read from file
  def charlist(self) :
    …
  def wordlist(self) :
    … # read from file
```

seule modification

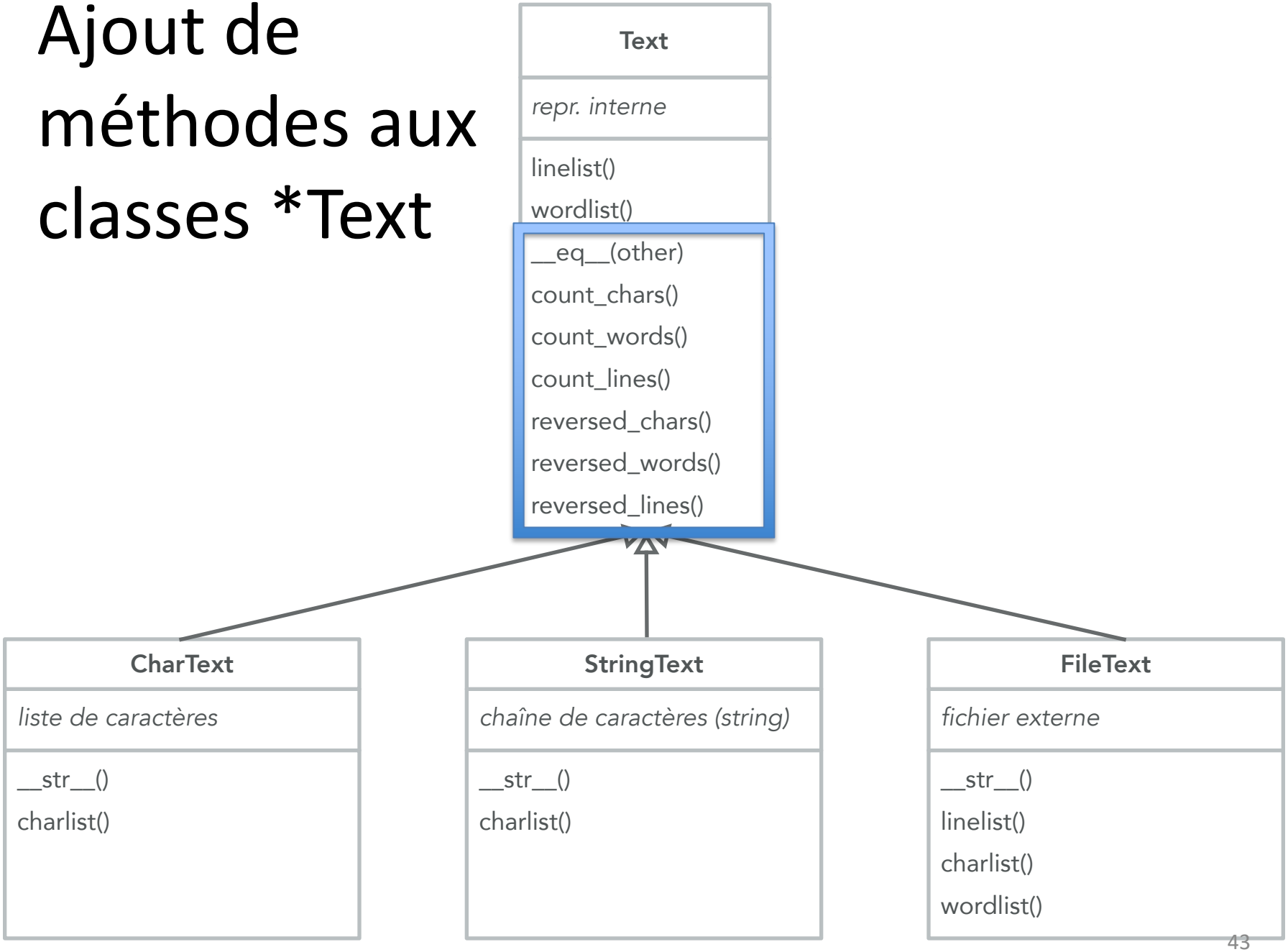
écrasement

écrasement

écrasement

FileText
fichier externe
__str__() linelist() charlist() wordlist()

Ajout de méthodes aux classes *Text



Ajout de méthodes à la classe Text

```
class Text :

  ...

  def count_chars(self) :
    return len(self.charlist())

  def count_words(self) :
    return len(self.wordlist())

  def count_lines(self) :
    return len(self.linelist())

  def __eq__(self,other) :
    return self.charlist() == other.charlist()

  ...
```

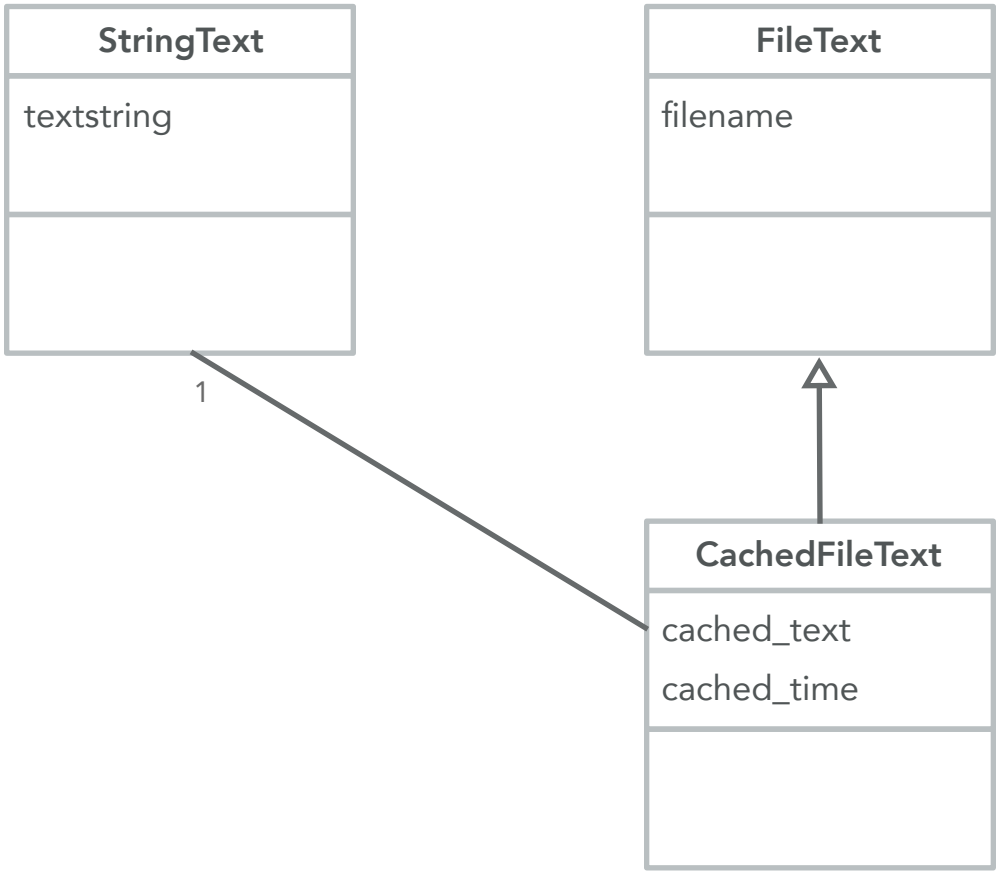
Text
repr. interne
linelist() wordlist() __eq__(other) count_chars() count_words() count_lines() reversed_chars() reversed_words() reversed_lines()

Ajout de méthodes à la classe Text

```
class Text :

  ...

  def reversed_chars(self) :
    chars = self.charlist().copy()
    chars.reverse()
    return "".join(chars)

  def reversed_words(self) :
    words = self.wordlist().copy()
    words.reverse()
    return " ".join(words)

  def reversed_lines(self) :
    lines = self.linelist().copy()
    lines.reverse()
    return "\n".join(lines)
```

Text
repr. interne
linelist() wordlist() __eq__(other) count_chars() count_words() count_lines() reversed_chars() reversed_words() reversed_lines()

Etendre la classe FileText



Etendre la classe FileText

```
class CachedFileText(FileText) :

  def __init__(self,filename) :
    super().__init__(filename)
    self.set_cache()

  def cached_text(self) :
    return self.__cached_text

  def cached_time(self) :
    return self.__cached_time

  def set_cache(self) :
    self.__cached_text = StringText(super().__str__())
    self.__cached_time = os.path.getmtime(self.filename())

 def update_cache_if_changed(self) :
   if self.cached_time() != os.path.getmtime(self.filename()):
     self.set_cache()
```

Héritage

Appel à super()

Appel à self

Méthode accesseur

Attribut privé

Méthode mutateur

Appel à super()

Etendre la classe FileText

```
class CachedFileText(FileText) :

  …

  def __str__(self) :
    self.update_cache_if_changed()
    return str(self.cached_text())

  def linelist(self) :
    self.update_cache_if_changed()
    return self.cached_text().linelist()

  def charlist(self) :
    self.update_cache_if_changed()
    return self.cached_text().charlist()

  def wordlist(self) :
    self.update_cache_if_changed()
    return self.cached_text().wordlist()
```

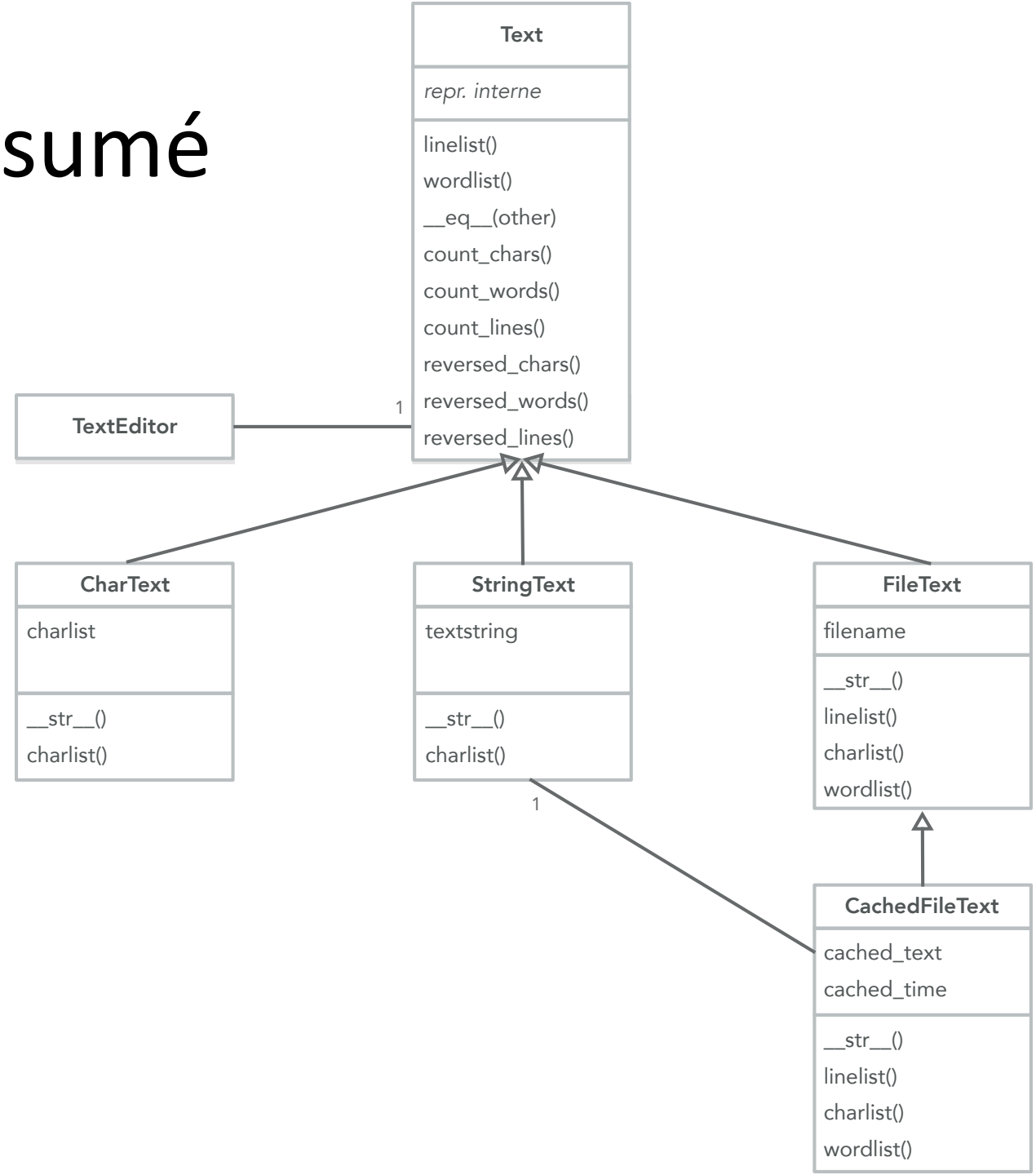
délégation

délégation

délégation

délégation

Résumé


Text
repr. interne
linelist()
wordlist()
__eq__(other)
count_chars()
count_words()
count_lines()
reversed_chars()
reversed_words()
reversed_lines()
1
TextEditor
CharText
charlist
__str__()
charlist()
StringText
textstring
__str__()
charlist()
1
FileText
filename
__str__()
linelist()
charlist()
wordlist()
CachedFileText
cached_text
cached_time
__str__()
linelist()
charlist()
wordlist()

Mission 10

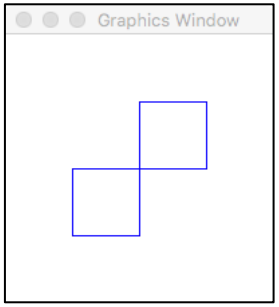
Objectifs :

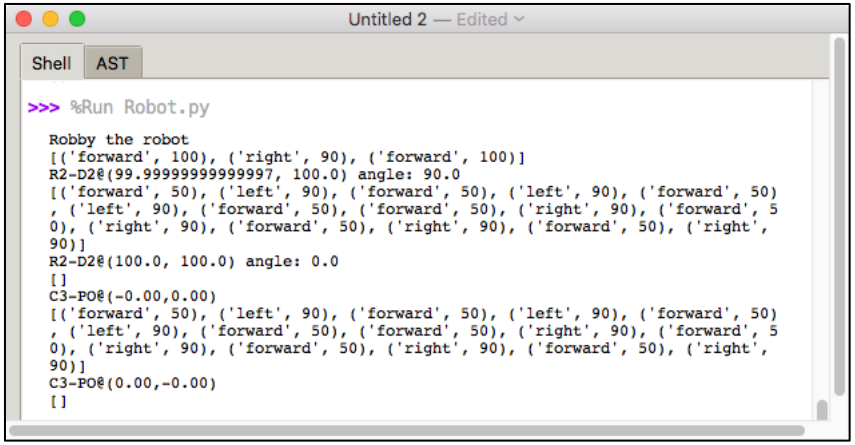
- Polymorphisme
- Tests unitaires

Problème :

- Deux implémentations alternatives d'un robot

XYRobot





TurtleBot