Préambule

- Notion de programme :
  - "liste d'ordres indiquant à l'ordinateur ce qu'il doit faire"
  - "se présente sous la forme de listes d'instructions (et de données de base) exécutées par l'ordinateur dans un certain ordre
- Notion d'algorithme :
  - "un énoncé dans un langage bien défini d'une suite 'opérations permettant de résoudre un problème par un calcul"

Paradigmes et langages de programmation

- Programmation impérative (Assembleur, Basic, Pascal, ...)
  - états du programme + séquence d'instructions (assignation, branchements, bouclage)
- Programmation fonctionnelle (ML, OCaml, Lisp, ...)
  - évaluation des fonctions mathématiques
- Programmation logique (Prolog, ...)
  - ensemble de faits élémentaires, des règles logiques associant des conséquences +/- directes
  - un moteur d'inférence/un démonstrateur de théorème

Programmation Objet ? Kesako ?

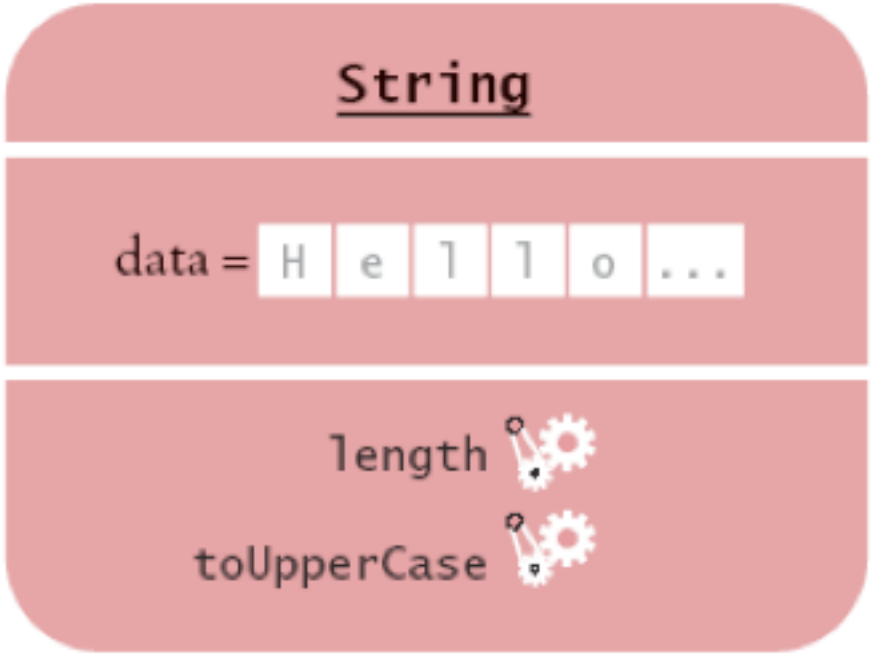
- Programmation dirigée par les données et non par les traitements
- Les procédures/méthodes existent toujours, mais on se concentre :
  - d'abord, sur les entités à manipuler
  - ensuite, comment les manipuler
- Notion d'encapsulation :
  - les données et les procédures liées sont regroupées au sein d'une même entité
  - cacher le fonctionnement interne d'une entité

Types

- Chaque valeur/expression a un type
- Exemple :
  - `"bonjour"` : type `str` (chaîne de caractères)
  - `'x'` : type `str` (caractère)
  - `27` : type `int` (entier)
  - `True` : type `bool` (valeur booléenne)

Objet

- Objet : une entité manipulée dans un programme (en appelant des méthodes)
- Un objet est caractérisé par :
  - Son identité :
    - Unicité
  - Son type
  - Son état :
    - valeurs des attributs à un moment donné
  - Son comportement :
    - ensemble des méthodes (consultation, mise à jour)

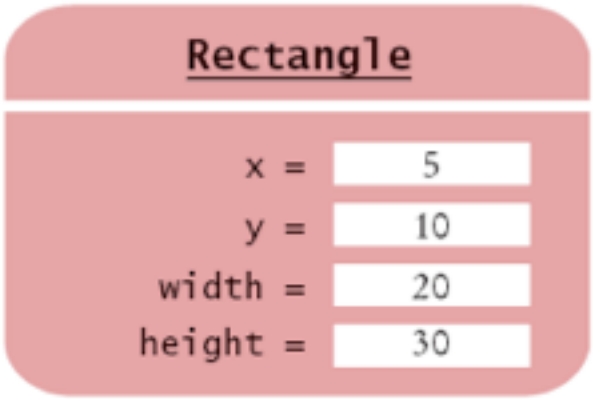


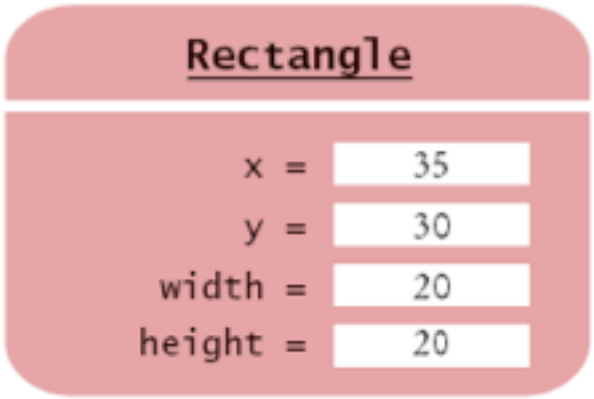
Classe

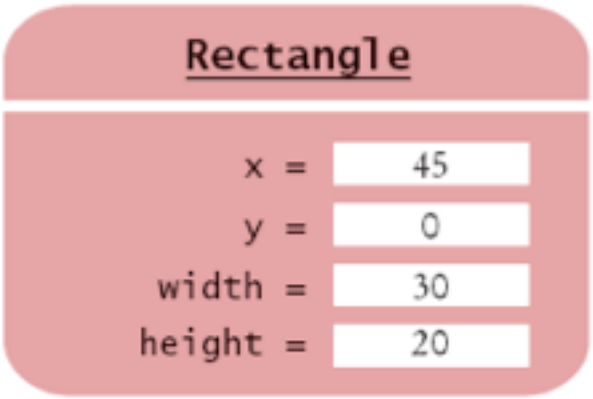
- Définition d'une famille d'objets ayant une même structure et un même comportement caractérisée par un nom
- Chaque objet appartient à une classe
- Permet d'instancier une multitude d'objets
- La classe d'un objet détermine les méthodes que l'on peut appeler sur un objet

Exemple d'objet : Rectangle

- Cette classe représente un `Rectangle` et non pas la figure Rectangle







- 3 objets = 3 instances de la classe `Rectangle`

Exemple de classe Python

```
Class Rectangle(object):
  def __init__(self, x, y, w, h):
    self.x = x
    self.y = y
    self.width = w
    self.height = h

  def translate(self, dx, dy):
    self.x = self.x + dx
    self.y = self.y + dy

  def getWidth():
    return self.width
```

Méthode

- Méthode : séquence d'instructions qui accèdent aux données d'un objet
- On manipule des objets par des appels de ses méthodes

```
boxA.translate(10.,10.)
```

- Interface publique :
  - définie ce l'on peut faire sur un objet

Paramètres explicites et receveur

- Paramètre (paramètre explicite) :
  - données en entrée d'une méthode
  - certaines méthodes n'ont pas de paramètres explicites

```
studentA.setName("Paul")
```

- Receveur (paramètre implicite) :
  - objet sur lequel on invoque la méthode (`self`)

```
def setName(self, new_name):
    self.name = new_name
```

Constructeurs

- Utilisation :

```
Rectangle(5, 10, 20, 30)
```

- Le constructeur :
  - construit l'objet de classe Rectangle
  - utilise les paramètres pour initialiser les attributs de l'objet
  - Retourne le nouvel objet
- Généralement, l'objet est stocké dans une variable :

```
box = Rectangle(5, 10, 20, 30)
```

Accesseur / Modificateur

- Accesseur : ne change pas l'état interne d'un objet (paramètre implicite)

```
width = box.getWidth()
```

- Modificateur : change l'état interne

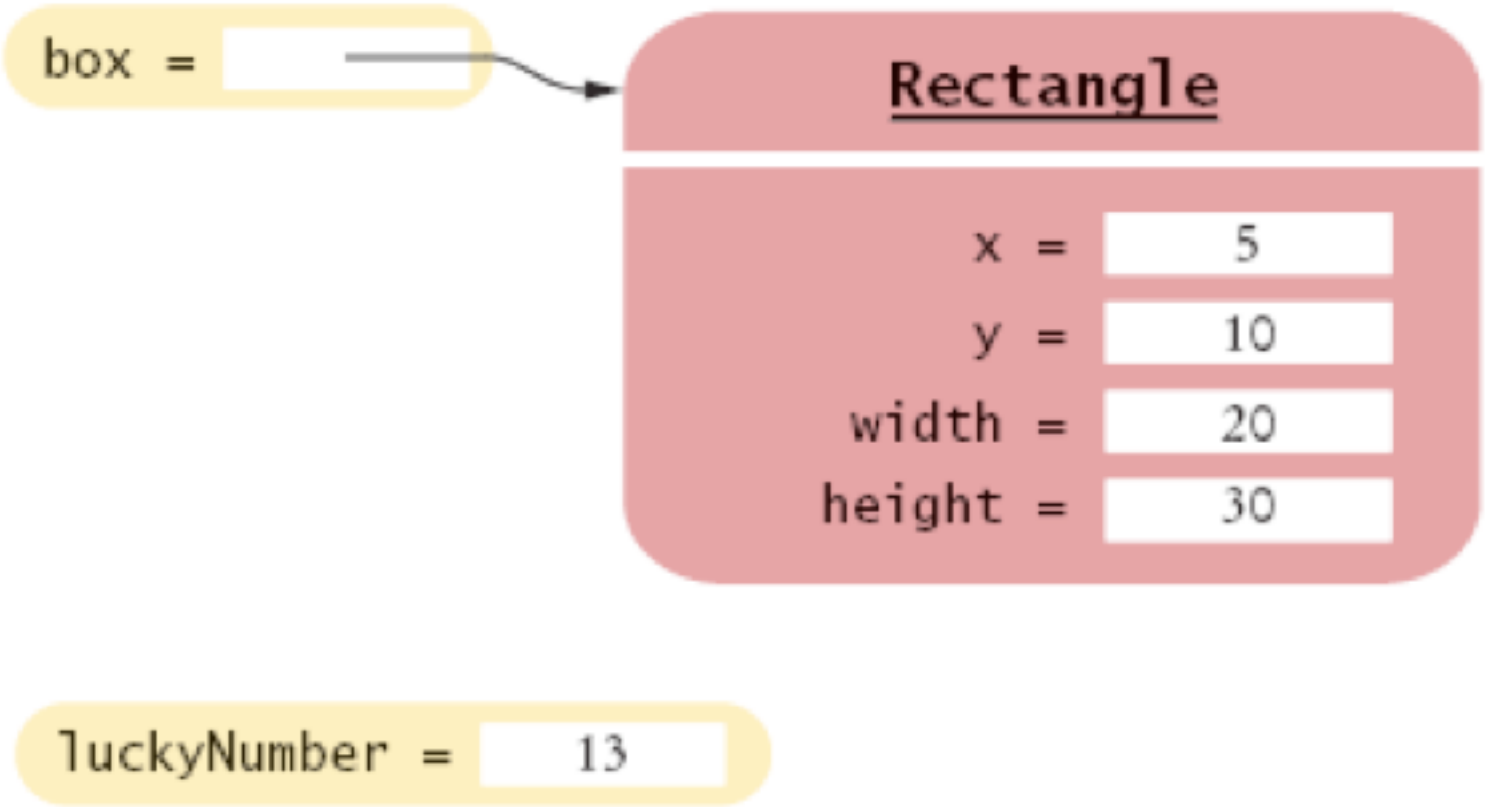
```
box.translate(15, 25)
```

Référence (d'un objet)

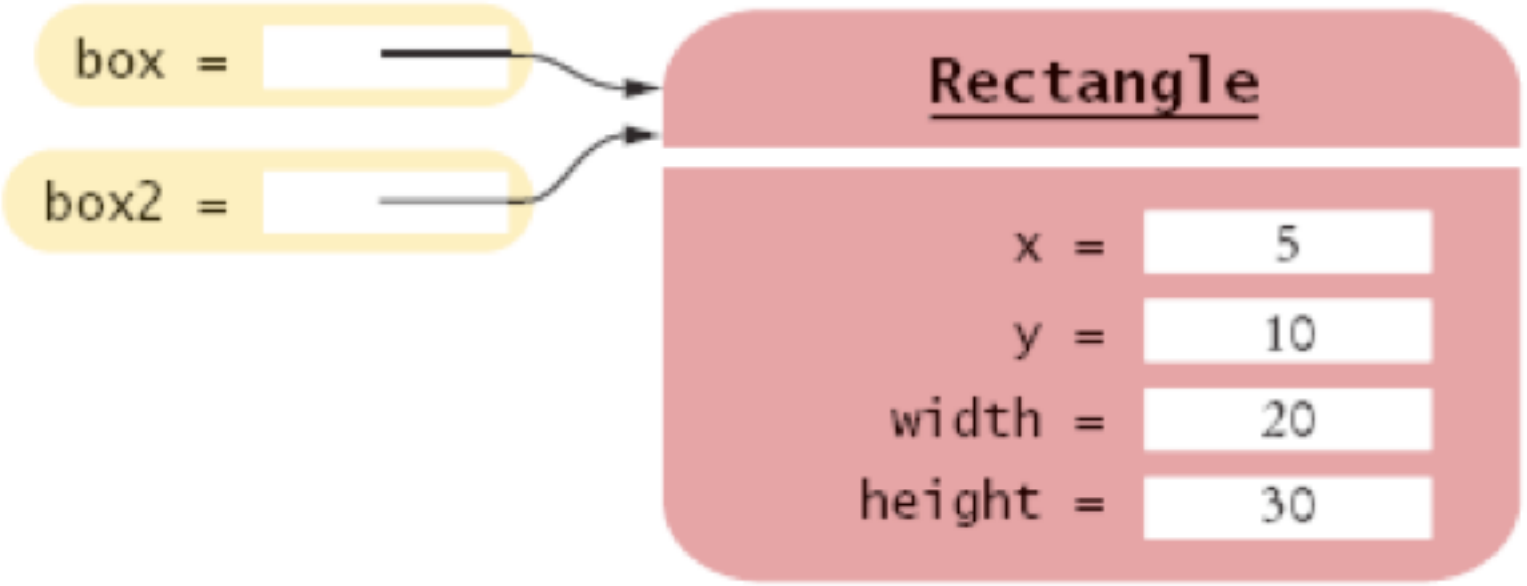
- Référence : décrit la localisation d'un objet
- Plusieurs variables peuvent référencer un même objet

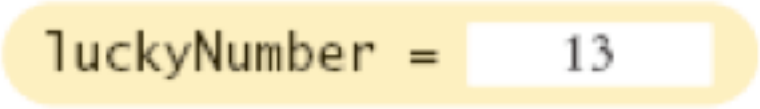
```
box = Rectangle(5, 10, 20, 30)
box2 = box
box2.translate(15, 25)
```

Référence /2

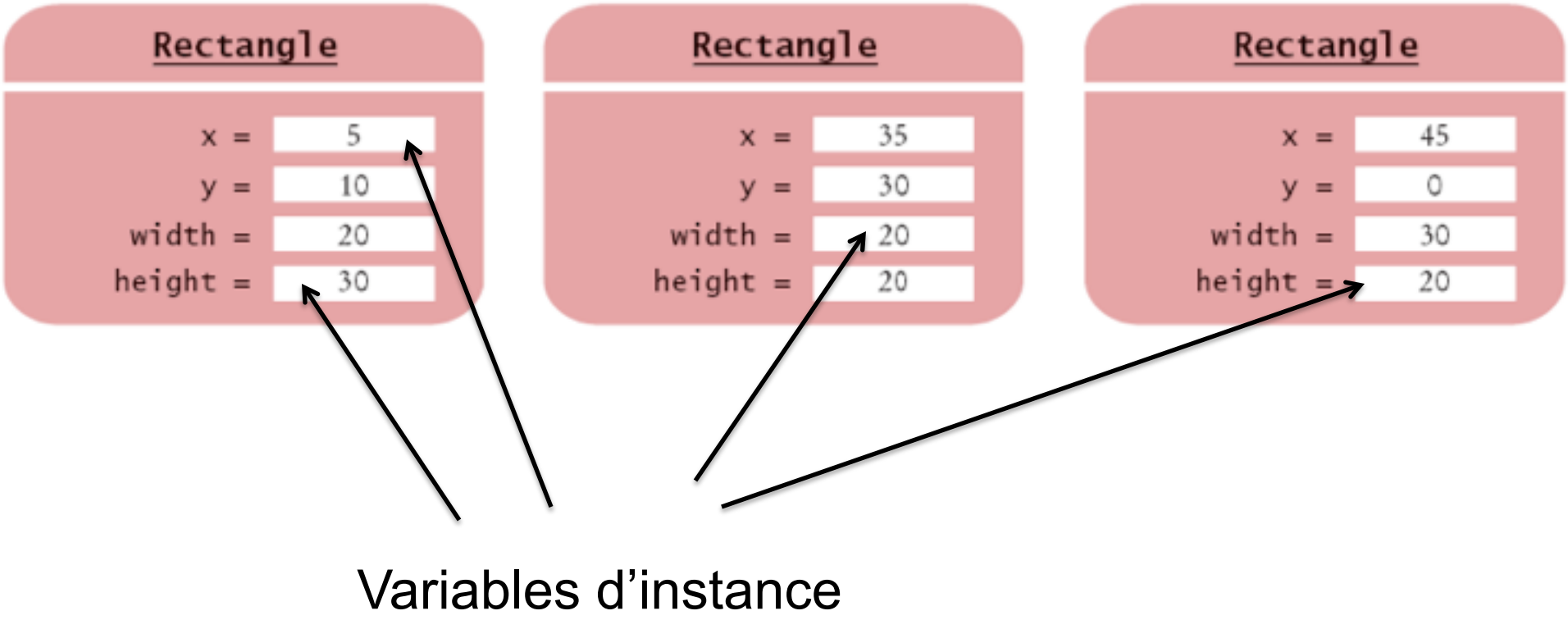


Référence /3





Variable/Champs d'une instance



Variables d'instance

Boîtes noires

- Une boîte noire réalise « magiquement » des choses
- Elle cache son fonctionnement interne
- **Encapsulation** : cacher les détails non important
- Quel est le bon *concept* pour chaque boîte noire particulière
- Concepts sont découverts par abstraction
- **Abstraction** : supprimer les fonctions non essentielles tant que l'essence du concept reste présente
- En *programmation orientée objet*, les objets sont les boîtes noires à partir desquels un programme est construit

Niveaux d'abstraction : Génie Logiciel

- Il y a bien longtemps : les programmes informatiques manipulaient des types primitifs tels que les nombres et les caractères
- Manipuler beaucoup de données de ce genre menait souvent à des erreurs
- Solution : Encapsulater les routines de calcul dans des boites noires logicielles
- L'abstraction est utilisée pour créer de nouveaux types de données de plus haut niveau
- En programmation orientée-objet, les objets sont les boites noires
- Encapsulation : Programmer en connaissant le comportement d'un objet et non pas sa structure interne

Niveaux d'abstraction : Génie Logiciel /2

- En génie logiciel, il est possible de concevoir de **bonnes** et de **mauvaises abstractions** offrant des **fonctionnalités identiques** ;
- Comprendre ce qu'est une bonne conception est l'une des enseignements les plus importants qu'un développeur peut apprendre.

- En premier, définir le comportement d'une classe

  => Interface publique

- Ensuite, implémenter cette classe

Découvrir et choisir des classes

- Une classe représente un unique concept/notion du monde du problème (chercher les noms dans l'énoncé du problème)
- Le nom d'une classe est généralement un nom qui décrit un concept
- Concepts mathématiques :

```
Point
Rectangle
Ellipse
```

- Concepts de la vie de tous les jours :

```
BankAccount
CashRegister
```

Découvrir et choisir des classes /2

- Acteurs – Objets qui ‘‘travaille pour vous’’
  `Scanner`
  `Random // meilleur nom: RandomNumberGenerator`
- Classes utilitaires – pas d’objet (instance) seulement des méthodes de classes
  `Math`
- Ne transformer pas (en général) les actions en classe :
  `Paycheck` est un meilleur nom que `ComputePaycheck`

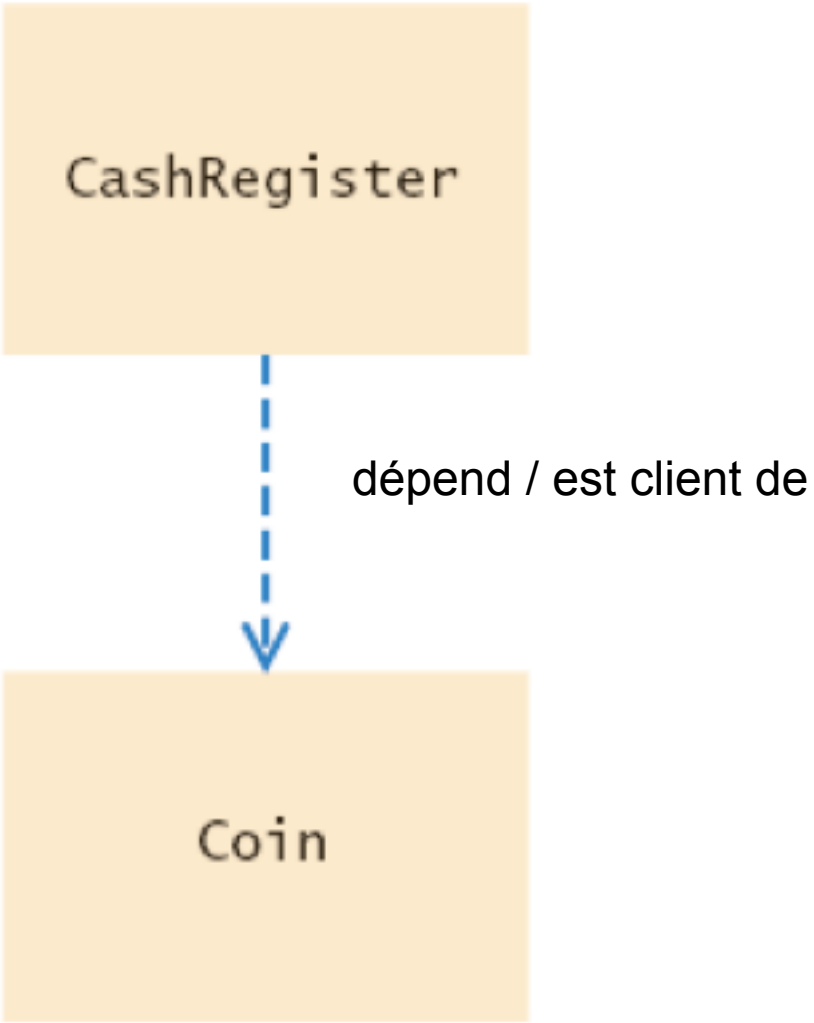
Cohésion

- Une classe doit représenter un seul concept
- L'interface publique d'une classe est cohésive si toutes ses fonctionnalités sont en relation avec le concept représenté

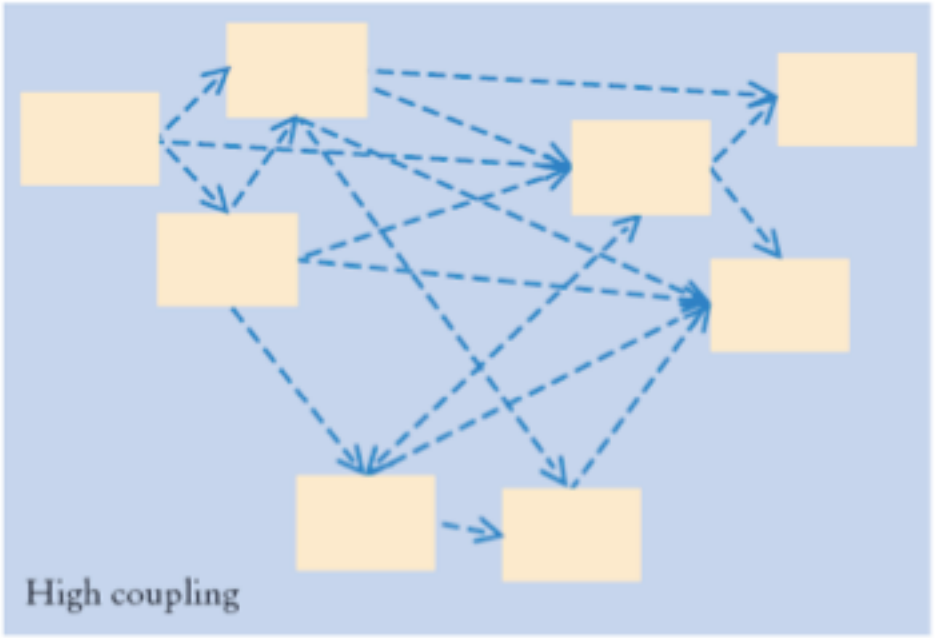
Couplage

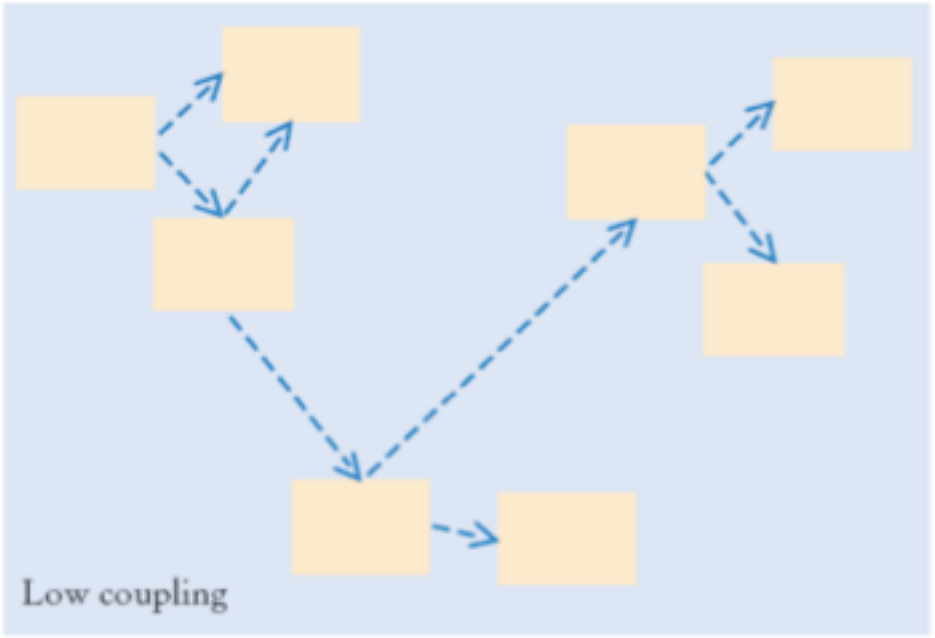
- Une classe *dépend* d'une autre classe si elle utilise des instances de cette seconde classe
- `CashRegister` dépend de `Coin` pour déterminer la valeur du paiement
- `Coin` ne dépend pas de `CashRegister`
- Couplage fort = beaucoup de dépendances entre classes
- Minimiser le couplage pour minimiser l'impact du changement d'une interface
- Pour visualiser les relations entre classes, dessinez des diagrammes
- UML: Unified Modeling Language. Une notation pour l'analyse et la conception orientée objet

Couplage



Couplage fort et faible entre des classes





Introduction à l'héritage

- Héritage : étendre des classes en ajoutant des méthodes et des variables d'instance
- Exemple : Compte d'épargne = compte bancaire avec des intérêts

```
class SavingsAccount(BankAccount):
    # new methods
    # new instance fields
```

- `SavingsAccount` hérite automatiquement de toutes les méthodes et variables d'instance de la classe `BankAccount`

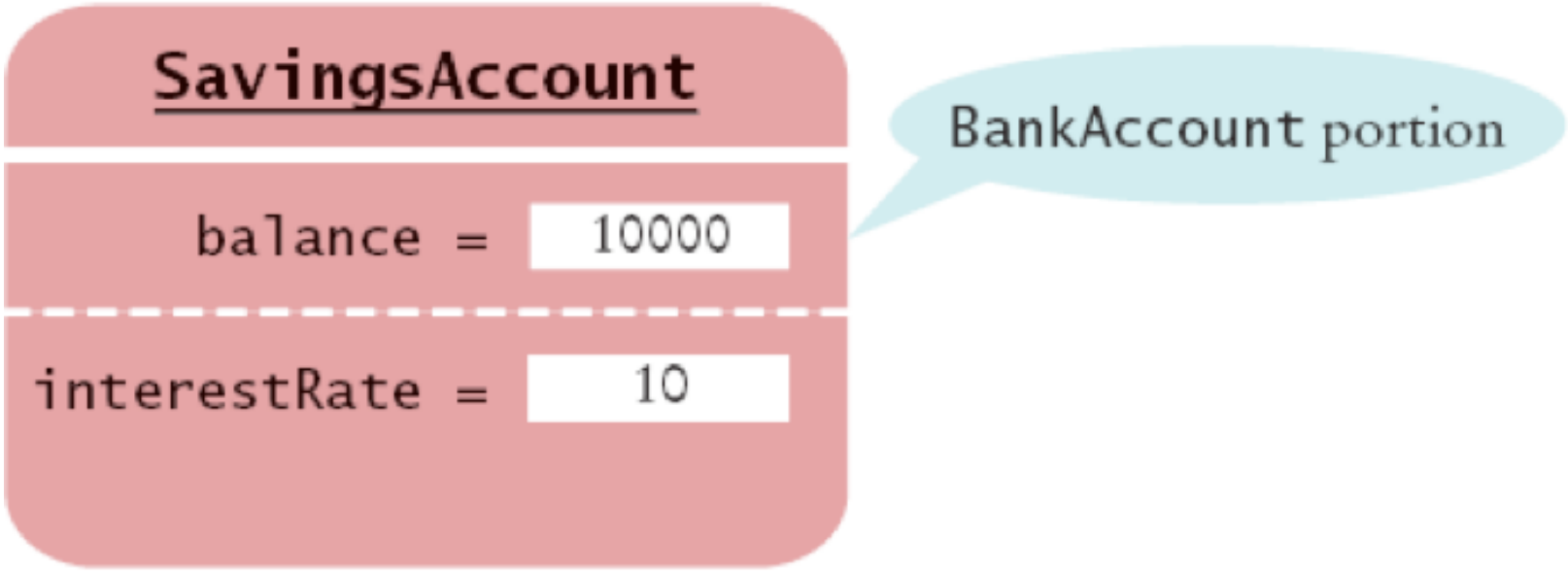
-
```
collegeFund = SavingsAccount(10)
# Savings account with 10% interest
collegeFund.deposit(500);
# OK to use BankAccount method with
SavingsAccount object
```

Introduction à l'héritage /2

- Classe étendue = Classe mère = *super classe* (`BankAccount`),
  Classe étendant = Sous classe (`Savings`)
- Hériter d'une classe ≠ d'implémenter une interface : une sous classe hérite de l'implémentation des méthodes et de l'état (variables d'instance)
- Un des avantages de l'héritage : la réutilisation de code

Sous classe

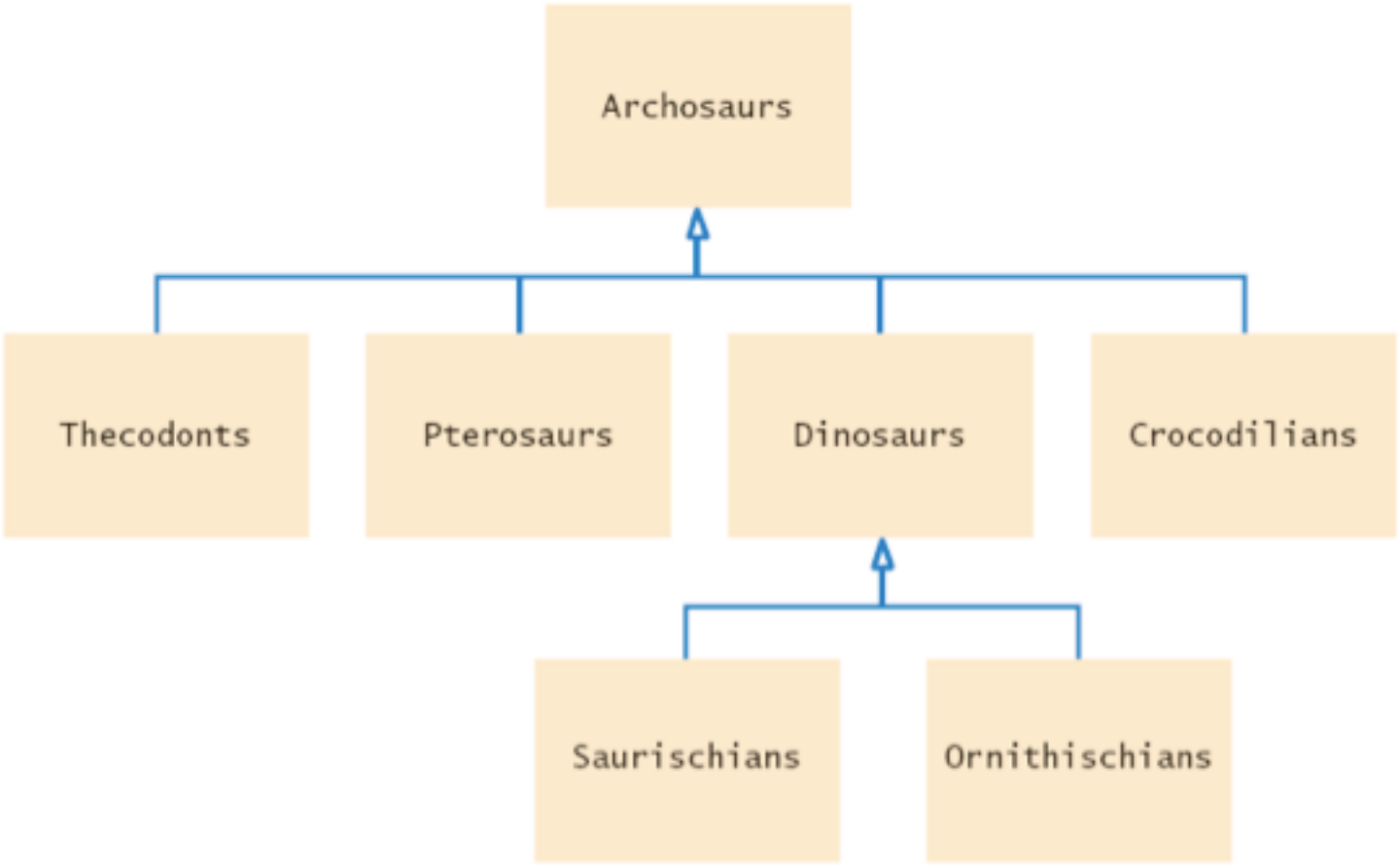
L'objet `SavingsAccount` hérite de la variable d'instance `balance` de la classe `BankAccount`, et gagne une variable additionnelle : `interestRate`:



- Dans la sous classe, sont spécifiés :
  - Les variables d'instance que l'on ajoute
  - Les méthodes que l'on ajoute
  - Les méthodes que l'on redéfinie (dont on change le comportement)

Hiérarchie de classes

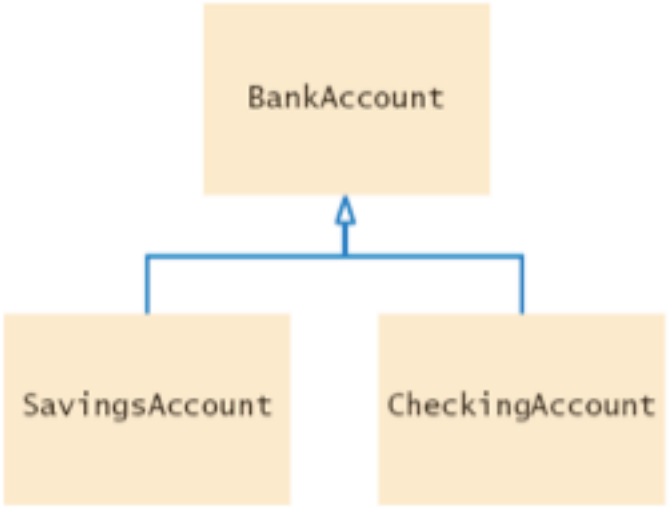
- Ensemble de classes qui forme arbre d'héritage
- Exemple :



Hiérarchie de classes /2

- Considérons une banque qui offre à ses clients deux types de compte :
  1. *Compte courant (Checking account): pas d'intérêt; un nombre (peu élevé) de transactions gratuites, des frais additionnels pour chaque transaction supplémentaire*
  2. *Compte d'épargne (Savings account) : des intérêts chaque mois*

- Hiérarchie de classe :



- Tous les comptes supportent la méthode `getBalance`
- Tous les comptes supportent les méthodes `deposit` et `withdraw`, mais leur l'implémentation différe
- Compte courant requiert une méthode `deductFees`; Compte d'éparne requiert une méthode `addInterest`

Héritage de méthodes

- Rédéfinition de méthodes (overriding) :
  - *Fournir une implémentation différente d'une méthode existante dans la classe mère*
  - *Doit avoir la même signature (même nom et même nombre et type de paramètres)*
  - *Si une méthode est appliquée sur un objet de la sous classe, la redéfinission de cette méthode est exécutée (cf. TD)*
- Méthodes héritées :
  - *Ne pas fournir de nouvelle implémentation pour une méthode de la classe mère*
  - *Les méthodes de la classes mère peuvent être appliquée sur des instances de la classe fille*
- Méthodes ajoutées :
  - *Fournir une méthode qui n'existe pas dans la classe mère*
  - *Cette nouvelle méthode ne peut être appliquée que sur les objets de la classe fille*

Héritage des variables d'instance

- On ne peut rédéfinir les variables d'instance de la classe mère
- Variables héritées: Toutes les variables de la classe mère sont automatiquement héritées
- Variables ajoutées : Définir de nouvelles variables qui n'existaient pas dans la classe mère
- Que se passe-t-il si l'on définit une nouvelle variable avec le même nom qu'une variable de la classe mère ?
    - *Chaque objet possèdera deux variables d'instances avec le même nom*
    - *Ces variables pourront contenir des valeurs différentes*
    - *Possible mais clairement déconseillé*