Langage et algorithmique

Rodéric Moitié

ENSTA Bretagne

Sommaire

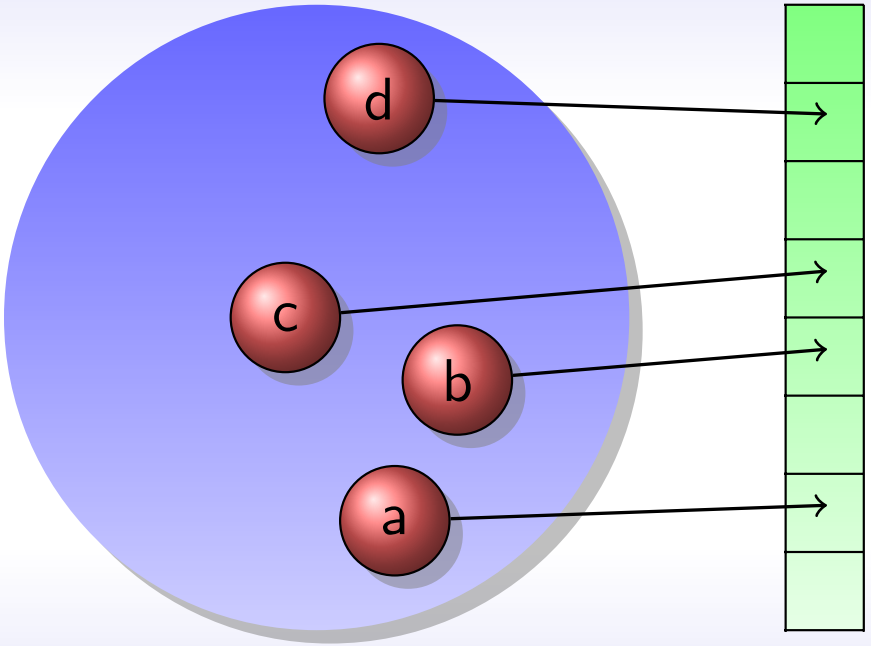
Principe

- Objectif : structure de données accessible en $\Theta(1)$

Principe

- Objectif : structure de données accessible en $\Theta(1)$
- Idée : ranger les éléments dans un tableau

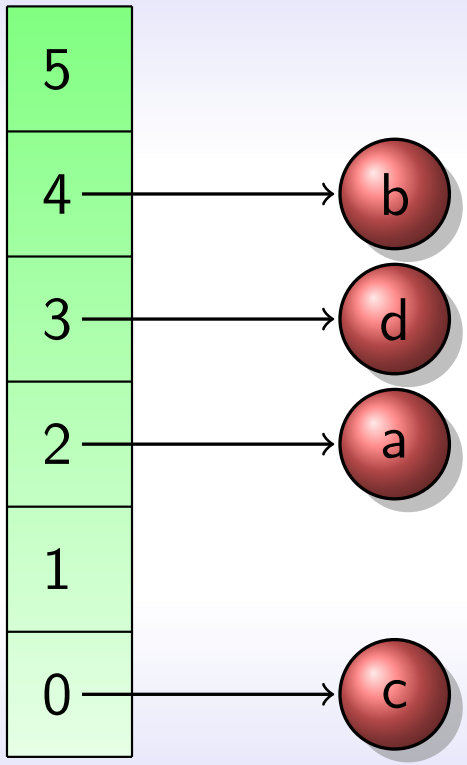
Principe



Principe

- Objectif : structure de données accessible en $\Theta(1)$
- Idée : ranger les éléments dans un tableau
- Associer à une donnée une clé
- Clé = case du tableau
- Clé : fonction de hachage

Principe



Fonction de hachage

- Cas idéal : fonction injective

Fonction de hachage

- Cas idéal : fonction injective
- Mais contrainte de mémoire

Fonction de hachage

- Cas idéal : fonction injective
- Mais contrainte de mémoire

$\Rightarrow$ limiter la valeur maximale de la fonction

Fonction de hachage

- Cas idéal : fonction injective
- Mais contrainte de mémoire

$\Rightarrow$ limiter la valeur maximale de la fonction

$\Rightarrow$ risque de collisions

Fonction de hachage

- Cas idéal : fonction injective
- Mais contrainte de mémoire

$\Rightarrow$ limiter la valeur maximale de la fonction

$\Rightarrow$ risque de collisions

$\Rightarrow$ chaque case du tableau doit contenir un ensemble de valeurs

Fonction de hachage

- Cas idéal : fonction injective
- Mais contrainte de mémoire

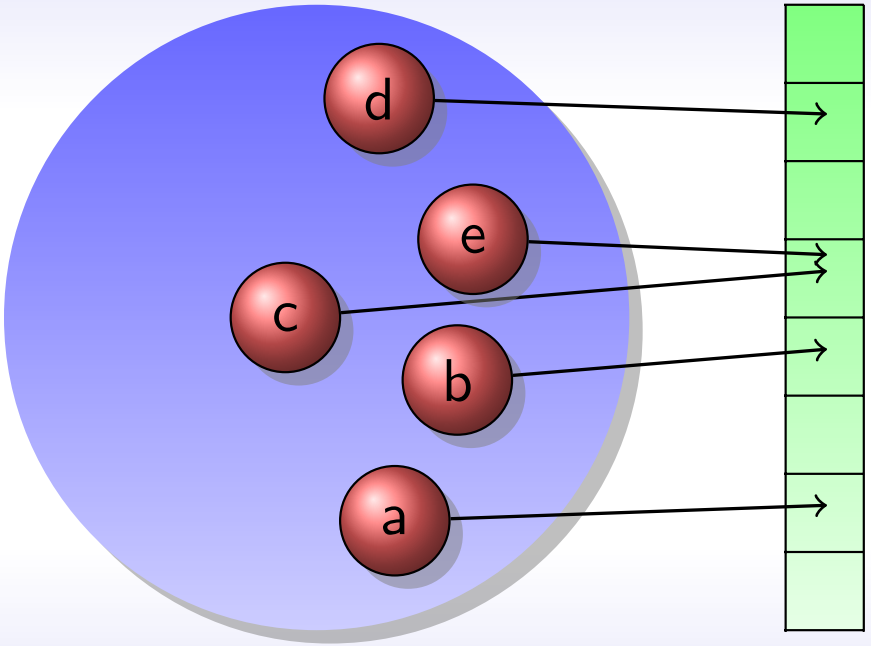
$\Rightarrow$ limiter la valeur maximale de la fonction

$\Rightarrow$ risque de collisions

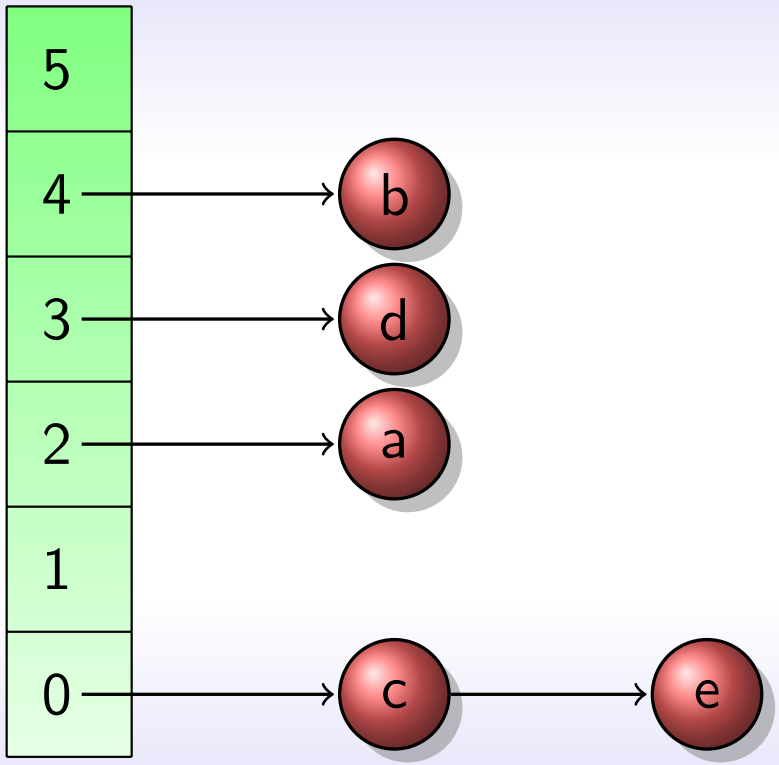
$\Rightarrow$ chaque case du tableau doit contenir un ensemble de valeurs

$\Rightarrow$ utilisation de listes chaînées

Fonction de hachage



Fonction de hachage



Algorithmes

Accéder à la table de hachage :

- Calculer le code k associé à l'élément
- Accéder à la liste de la case k du tableau
- cf. algorithmes sur les listes chaînées

Calcul de la clé :

- Utilisation d'un tableau de n cases
- Fonction de hachage :

$$\begin{array}{rccl} h: & \mathcal{K} & \rightarrow & [0, n-1] \subset \mathbb{N} \\ & e & \mapsto & h(e) \end{array}$$

Algorithmes

Insertion sans collision

Algorithme 1: Insertion

Entrées : Table T, Element e

T[h(e)] ←e ;

Recherche sans collision

Algorithme 2: Recherche

Entrées : Table T, Element e

retourner T[h(e)];

Algorithmes

Insertion avec collision

Algorithme 3: Insertion

Entrées : Table T, Element e

```
T[h(e)].insereEnTeteDeListe(e);
```

Recherche avec collision

Algorithme 4: Recherche

Entrées : Table T, Element e

```
retourner T[h(e)].rechercheDansLaListe(e);
```

Hachage

Taux de remplissage :

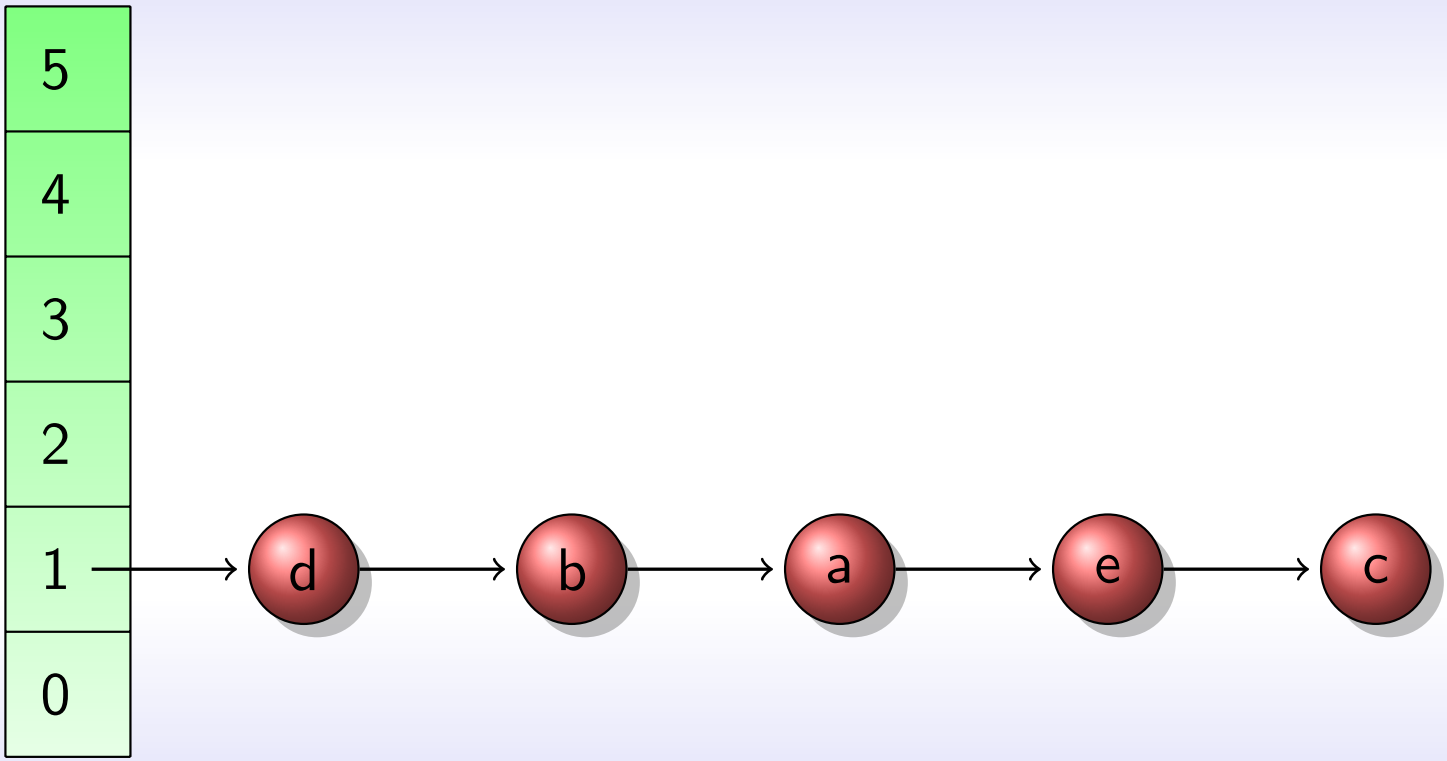
- Table T, m alvéoles, n éléments
- Taux de remplissage : $\alpha = n/m$

- $h(x)$ détermine une case du tableau

$\Rightarrow \forall x, h(x) < n$

- Cas extrême : uniquement des collisions $\rightsquigarrow \Theta(n)$

Hachage



Hachage

Taux de remplissage :

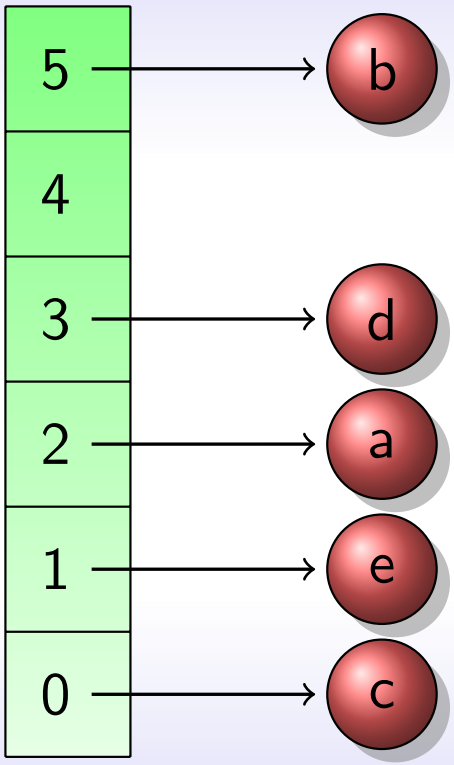
- Table T, m alvéoles, n éléments
- Taux de remplissage : $\alpha = n/m$

- $h(x)$ détermine une case du tableau

$\Rightarrow \forall x, h(x) < n$

- Cas extrême : uniquement des collisions $\rightsquigarrow \Theta(n)$
- Cas extrême : répartition uniforme $\rightsquigarrow \Theta(1 + \alpha)$

Hachage



Définition d'une fonction de hachage

- Bonne fonction de hachage ?

$\rightsquigarrow$ bonne répartition ($\simeq$ répartition uniforme)

Exemple (Méthode de la division)

- Associer un entier k à un élément
- $h(k) = k \mod m$
- Attention au choix de m
  - Éviter $m = 2^p$, éviter les nombres proches des puissances de 2
  - Bon choix : nombre premier éloigné des puissances de 2
  - Exemple : 2000 éléments, $\alpha \simeq 3$ accepté $\rightsquigarrow m = 701$

Définition d'une fonction de hachage

Exemple (Méthode de la multiplication)

- Soit $A \in]0,1[$, $h(k) = \lfloor m(kA \mod 1) \rfloor$
- Choix de m non critique ; en général $m = 2^p$
- Choix de A ?
  - On veut k sur un mot de w bits
  - $A = s/2^w$ avec $0 < s < 2^w$
  - Knuth suggère $A \approx (\sqrt{5} - 1)/2$
- Ex.
  - $k = 123456, p = 14, m = 2^{14} = 16384, w = 32$
  - $A = 2654435769/2^{32} \approx (\sqrt{5} - 1)/2$
  - $m(kA \mod 1) = 67.187744140625 \rightsquigarrow h(k) = 67$

Exemple

Exemple (Hachage de mots)

- Mot : suite de lettres ($w = a_0 a_1 a_2 \dots a_{p-1}$)
- Lettres codées par des entiers (ex. *ASCII*)
- Si uniquement lettres minuscules, utiliser $[0, 25]$
- On note $'a'$ le code *ASCII* de la lettre a
- Exemple : coder "secret", "cretes"

Exemple

Mauvaise fonction : $h(w) = \sum_{i=0}^{p-1} a_i \mod m$

- Exemple avec $m = 43$
- "secret" $\rightsquigarrow$ 21
- "cretes" $\rightsquigarrow$ 21
- "cat" $\rightsquigarrow$ 21
- "av" $\rightsquigarrow$ 21
- "va" $\rightsquigarrow$ 21

Exemple

Bonne fonction : $h(w) = \sum_{i=0}^{p-1} a_i \times 26^i \mod m$

- Exemple avec $m = 43$
- "secret" $\rightsquigarrow$ 11
- "cretes" $\rightsquigarrow$ 25
- "cat" $\rightsquigarrow$ 26
- "av" $\rightsquigarrow$ 22
- "va" $\rightsquigarrow$ 13

Exemple

Comment calculer $h(w) = \sum_{i=0}^{p-1} a_i \times 26^i \mod m$?

- Problème : éviter de calculer explicitement 26^i

Exemple

Comment calculer $h(w) = \sum_{i=0}^{p-1} a_i \times 26^i \mod m$?

- Problème : éviter de calculer explicitement 26^i
- En Java, calcul de 26^i en entier OK jusqu'à $i = 6$

Exemple

Comment calculer $h(w) = \sum_{i=0}^{p-1} a_i \times 26^i \mod m$?

- Problème : éviter de calculer explicitement 26^i
- En Java, calcul de 26^i en entier OK jusqu'à $i = 6$
- Solution : forme de Horner

Exemple

Comment calculer $h(w) = \sum_{i=0}^{p-1} a_i \times 26^i \mod m$?

- Problème : éviter de calculer explicitement 26^i
- En Java, calcul de 26^i en entier OK jusqu'à $i = 6$
- Solution : forme de Horner
- $h(w) = a_0 + 26(a_1 + 26(a_2 + 26(\ldots)))$
- Calculé en Java avec une boucle

Sommaire

Motifs de conception

Principe :

- Réponse à des schémas classiques : modèles objets
- Solutions classiques : *design patterns*
- À appliquer si on est dans le cas considéré
- Souvent à adapter

Exemples de motifs de conception :

- Design Patterns – Tête la première
- Design Patterns : Elements of Reusable Object-Oriented Software (GOF)

Sommaire

Objectif

- Un objet possède une stratégie (comportement)
- Stratégie peut changer dynamiquement

Utilité :

- Plusieurs classes ne diffèrent que par leur comportement
- Plusieurs variantes du comportement

Exemple :

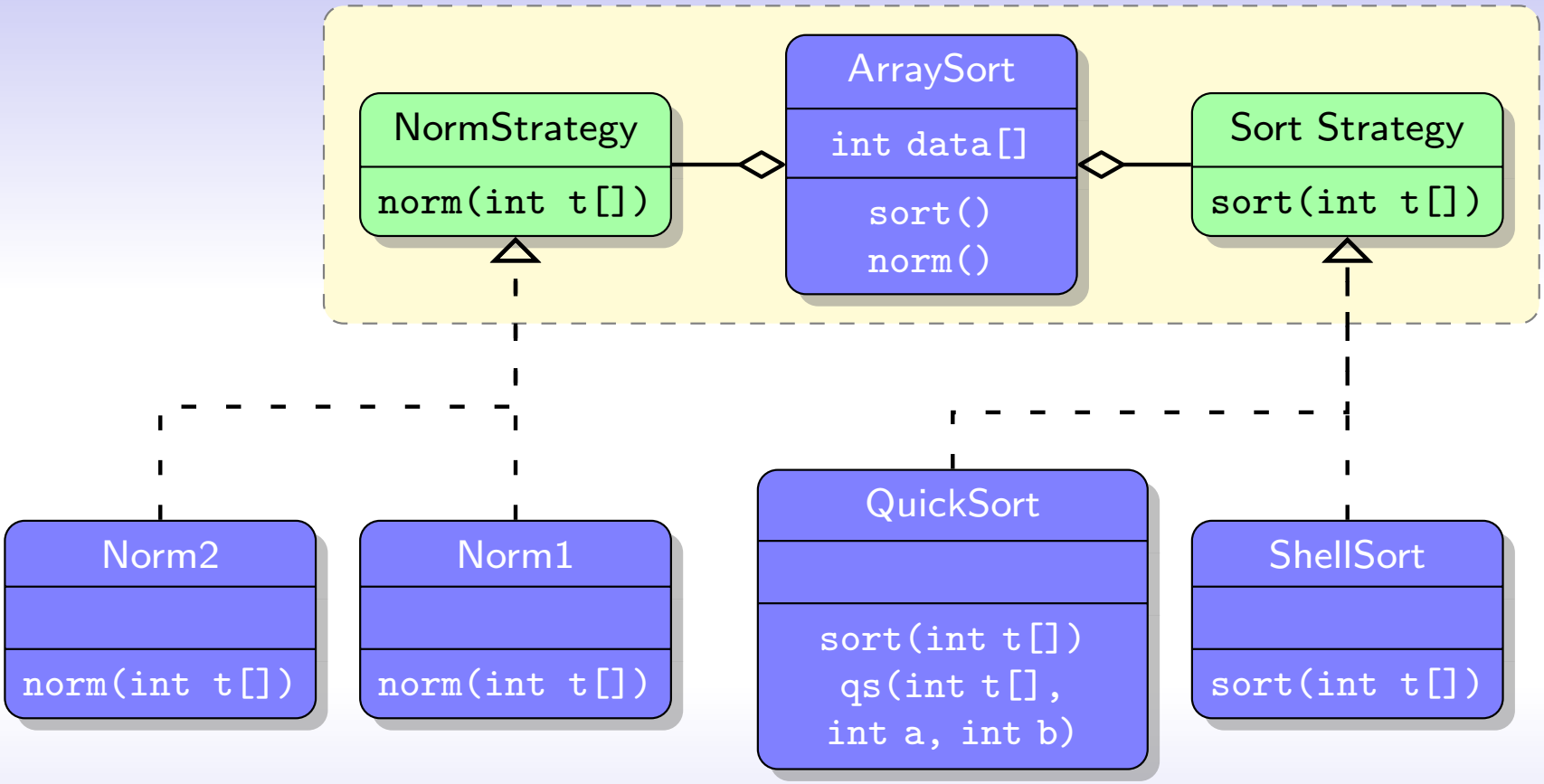
- Besoin de trier des vecteurs
- Plusieurs normes pour mesurer les vecteurs

Principe

- Classe possédant une stratégie
- Interface stratégie $\rightsquigarrow$ méthodes abstraites
- L'utilisateur écrit les classes utilisant la stratégie

$\Rightarrow$ obligation de redéfinir les méthodes

- Changement de stratégie : instancier une nouvelle classe


ArraySort
int data[]
sort()
norm()
NormStrategy
norm(int t[])
Sort Strategy
sort(int t[])
Norm2
norm(int t[])
Norm1
norm(int t[])
QuickSort
sort(int t[])
qs(int t[],
int a, int b)
ShellSort
sort(int t[])

Stratégie en Java

```
public static void main(String[] args) {
  // tableau de tris possibles
  SortStrategy[] alg = {new ShellSort(),
    new QuickSort() };
  // parcours de l'ensemble des tris
  for (SortStrategy str : alg) {
    // initialiser le ArraySort
    ArraySort as = new ArraySort(30, str,
            new Norm2());
    System.out.println(as);
    // tri utilisant la strategie
    as.sort();
  }
```

Avantages/inconvénients

Avantages

- Fait apparaître des familles d'algorithmes $\Rightarrow$ factorisation
- Possibilité de changer dynamiquement de stratégie
- Possibilité de posséder plusieurs stratégies
- Séparation des différents comportements

Inconvénients

- Il faut connaître les stratégies pour les utiliser
- Prototypes identiques
- Nombre de classes

Sommaire

Objectif

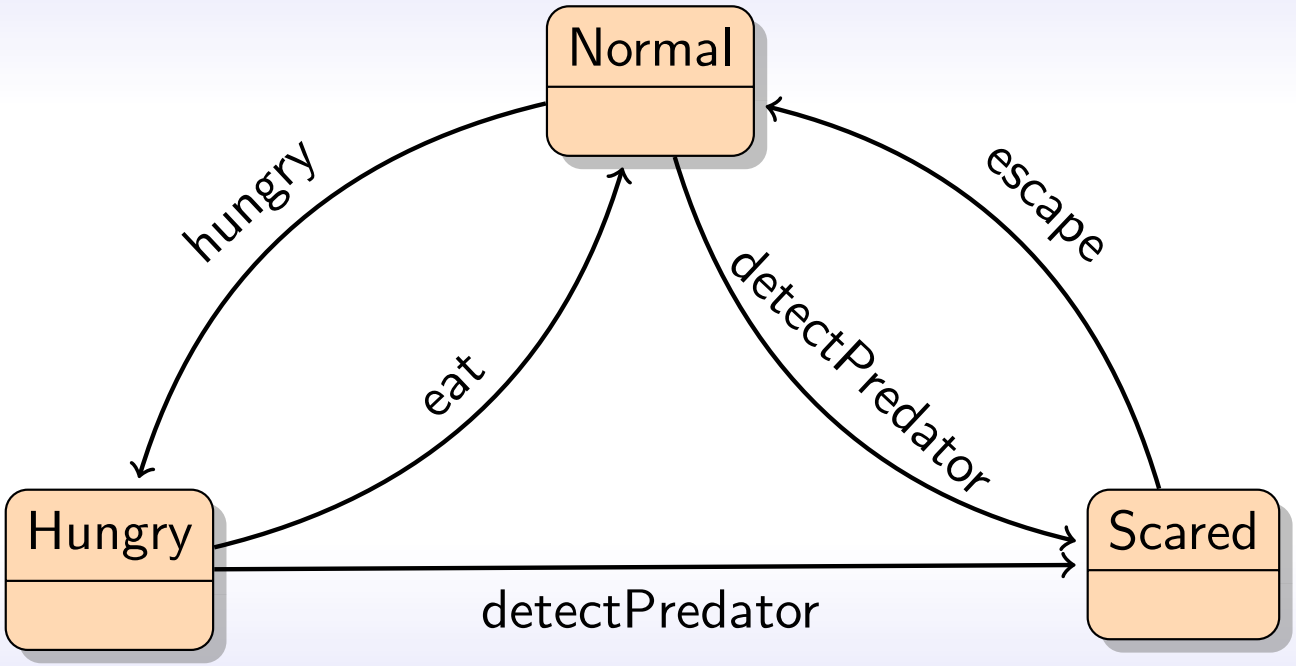
- Associer un comportement à un objet
- Comportement = états + transitions
- Dans chaque état : une action

⇝ Automate

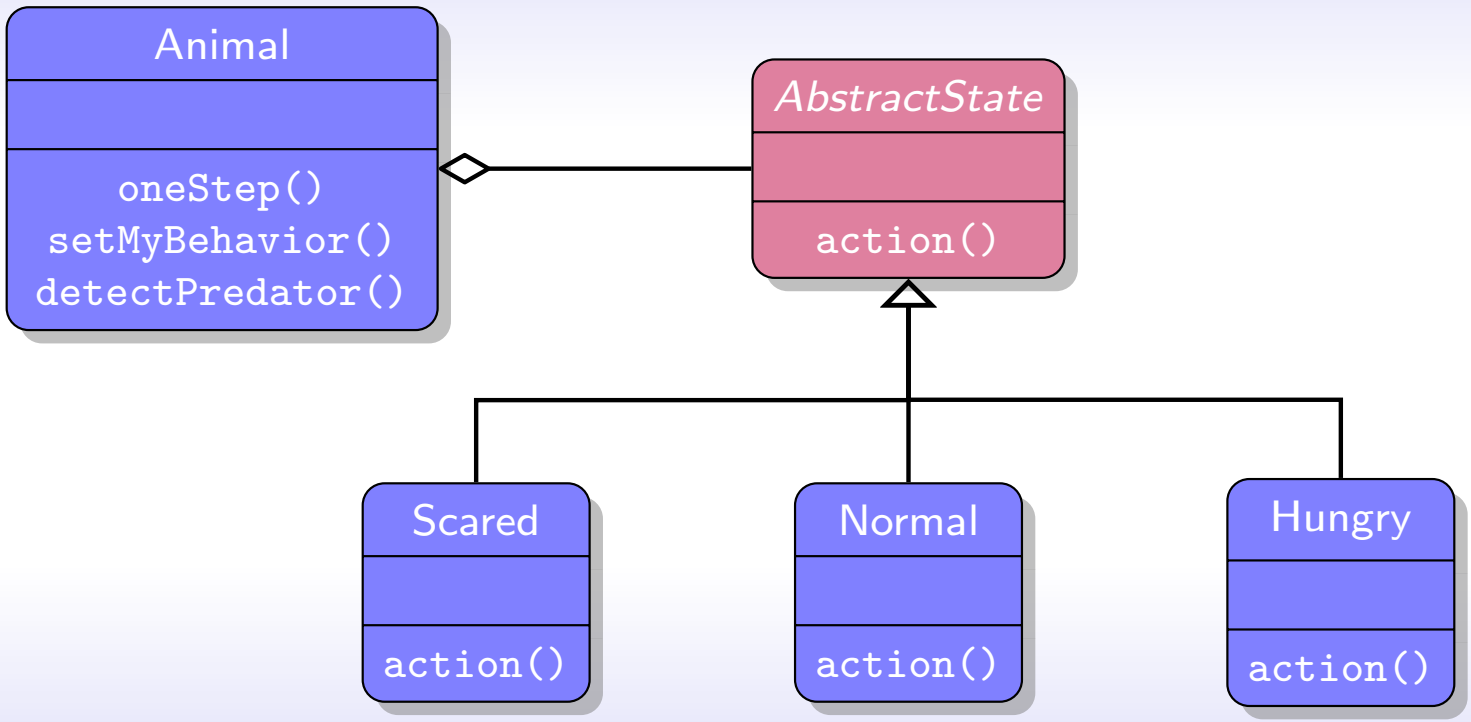
Codage :

- Classe abstraite État
- + sous-classes

Exemple


Normal
hungry
escape
eat
detectPredator
Hungry
Scared
detectPredator

Exemple


Animal
oneStep()
setMyBehavior()
detectPredator()
AbstractState
action()
Scared
action()
Normal
action()
Hungry
action()

État en Java

```
public class Animal {
  /** etat courant de l'animal */
  private AbstractState myBehavior;
  public Animal() {
    // initialement : etat normal
    myBehavior = new Normal();
  }
  /** effectue un tour pour l'animal */
  public void doOneStep() {
    this.myBehavior.action(this);
  }
}
```

État en Java

```
public class Normal extends AbstractState {
  public void action(Animal a) {
    if (a.detectPredator()) {
      // nouveau comportement : fuite
      a.setMyBehavior(new Scared());
    } else if (a.isHungry()) {
      // nouveau comportement : faim
      a.setMyBehavior(new Hungry());
    }
  }
}
```

Avantages/inconvénients

Avantages

- Rassemble tout le code traitant un état
- Ajout de nouvel état simple
- Transitions explicites

Inconvénients

- Classes à créer
- Création d'un objet à chaque transition

⇝ possibilité d'utiliser Singleton

Sommaire

Singleton

Objectif

Garantir qu'une classe ne pourra être instanciée qu'une seule fois.

Réalisation ?

- Classe **Singleton**
- Variable de classe de type **Singleton**
- Constructeur **privé**
- Méthode **de classe** d'instanciation

Singleton en Java

```
public class Singleton {
    /** singleton cree */
    private static Singleton s = null;
    /** constructeur prive */
    private Singleton ()  {
    }
    /** methode de classe de creation */
    public static Singleton getSingleton () {
      // creer le singleton si necessaire
      if (s == null)
        s = new Singleton();
      return s;
    }
  }
```

Utilisation du singleton en Java

```
public static void main (String[] args)  {
  Singleton s = Singleton.getSingleton();
  System.out.println(s);
  Singleton r = s.getSingleton(); // r = s
  System.out.println(r);
}
```

Sommaire

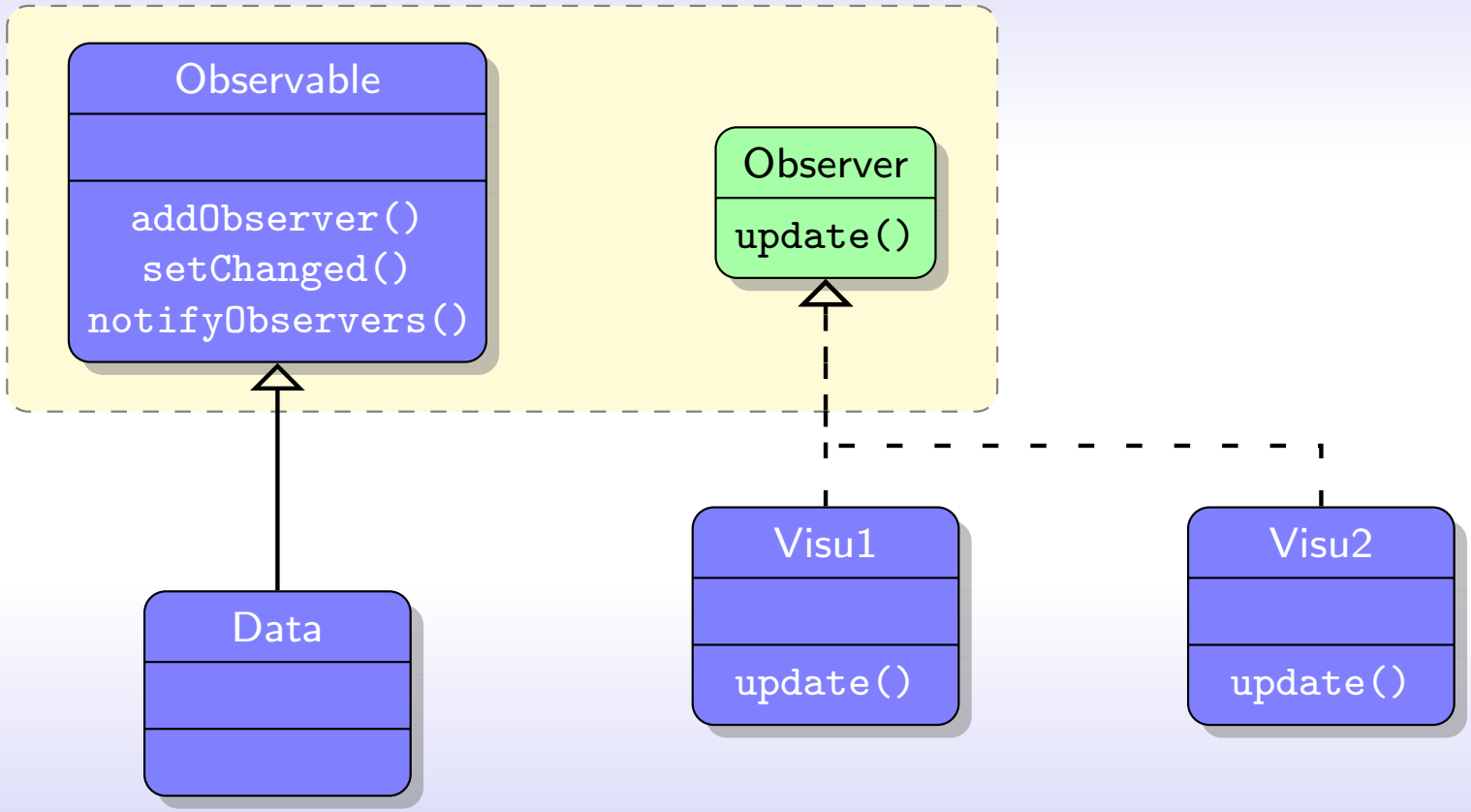
Objectif

- Simplifier la conception d'IHM
- Mécanisme de mise à jour d'interfaces observant des données
- Ajout de plusieurs vues très simple

Principe :

- Données :
  - hériter de **Observable**
  - signaler des mises à jour avec **setChanged** et **notifyObservers**
- Vues :
  - implanter **Observer**
  - réaliser les mises à jour dans **update**

Diagramme de classes



Code Java

```
public void setMyValue(double myValue) {
  this.myValue = myValue;
  setChanged();
  notifyObservers();
}
```

```
public void update(Observable o, Object arg) {
  // o est en fait un objet de type Data
  Data mesDonnees = (Data)o;
  // mise a jour l'interface
  // ...
}
```

Utilité

- une modification des données entraîne une mise à jour dans un nombre de vues non défini a priori
- Découplage fort données–affichage
- Mise à jour des visualisations par une interface commune