Conception objet en Java avec BlueJ
une approche interactive

10. Techniques d'abstraction avancées: classes abstraites et interfaces

David J. Barnes, Michael Kölling
version française: Patrice Moreaux

Principaux concepts abordés

- Classes abstraites
- Interfaces
- Héritage multiple

Simulations (1)

- Des programmes souvent employés pour simuler des activités du monde réel.
  - trafic urbain
  - climat, météorologie
  - phénomènes nucléaires
  - variations boursières
  - modifications de l'environnement

Simulations (2)

- Elles sont souvent seulement partielles.
- Elles sont une vue simplifiée.
  - plus de détail peut engendrer une meilleure précision…
  - … mais exige plus de ressources
    - traitements
    - temps de simulation

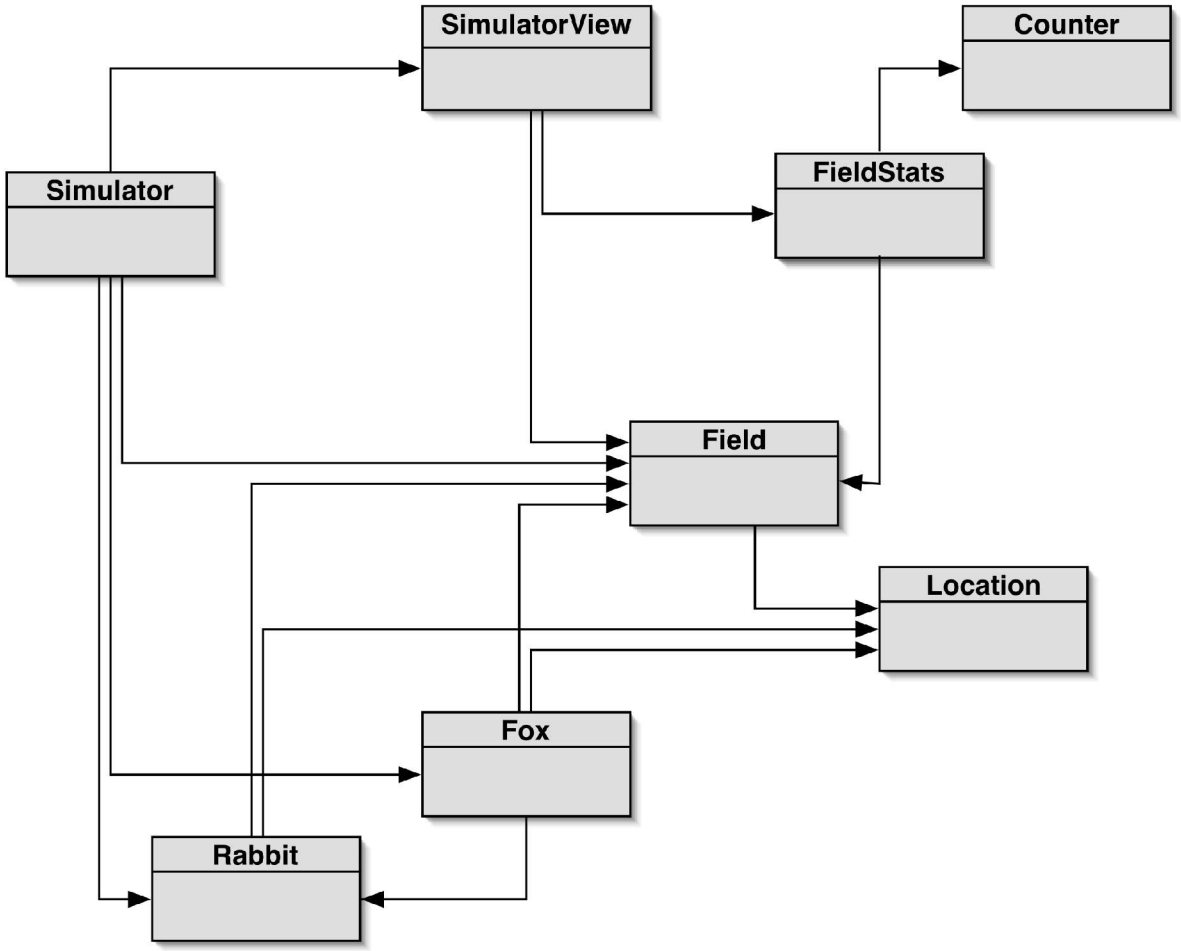
Intérêts des simulations

- Aide à la prévision
  - météorologie.
- Permet d'expérimenter
  - plus sûrement, moins cher, plus vite.
- Exemple:
  - "Comment la vie sauvage serait-elle affectée si une autoroute traversait ce parc national?"

Simulations prédateur-proie

- L'équilibre entre les espèces est souvent subtil
  - beaucoup de proies signifie beaucoup de nourriture.
  - beaucoup de nourriture entraîne plus de prédateurs.
  - plus de prédateurs mangent plus de proies
  - moins de proies signifie moins de nourriture.
  - moins de nourriture implique ...

Le projet *foxes-and-rabbits*



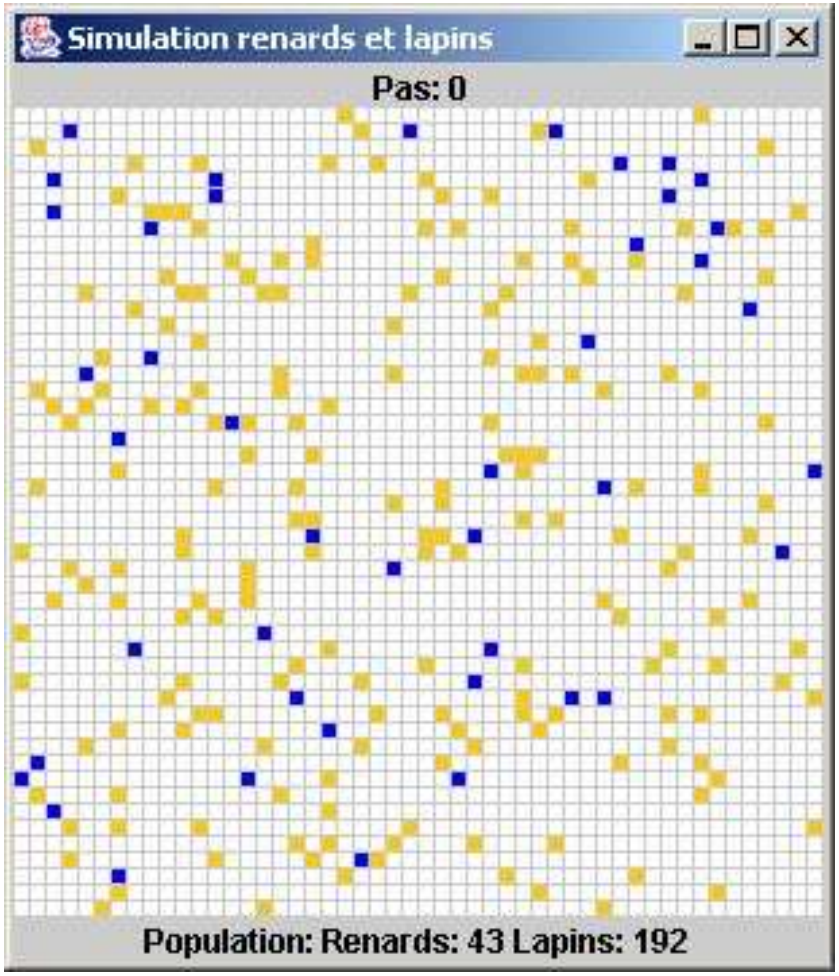
Principales classes

- **`Fox`** (renard)
  - modèle simple d'un type de prédateur.
- **`Rabbit`** (lapin)
  - modèle simple d'un type de proie.
- **`Simulator`**
  - gère la tâche globale de simulation.
  - maintient une collection de renards et de lapins.

Autres classes

- **`Field`** (terrain)
  - représente un terrain à 2 dimensions.
- **`Location`** (position)
  - représente une position 2D.
- **`SimulatorView`**, **`FieldStats`**, **`Counter`** (compteur)
  - gère des statistiques et présente une vue du terrain.

Exemple de vue



État d'un lapin

```
public class Rabbit
{
    // champs statiques non reproduits.

    // caracteristiques particulières
    // (champs d'instance).

    // L'âge du lapin.
    private int age;
    // Lapin vivant ou mort.
    private boolean alive;
    // Position du lapin
    private Location location;

    // Méthodes non reproduites.
}
```

Comportement d'un lapin

- Géré par la méthode **`run`**.
- L'âge est incrémenté à chaque "pas" de simulation.
  - un lapin peut mourir à ce moment là.
- Les lapin(e)s assez âgé(e)s peuvent mettre bas.
  - de nouveaux lapins peuvent naître à ce moment là.

Simplifications du comportement des lapins

- Un seul genre
  - en fait tous femelles.
- Un même lapin peut mettre bas à chaque pas.
- Tous les lapins meurent au même âge.
- Autres ?

État d'un renard

```
public class Fox
{
    // champs statiques non reproduits

    // âge du renard
    private int age;
    // renard vivant ou mort.
    private boolean alive;
    // Position du renard
    private Location location;
    // Niveau d'alimentation du renard,
    // augmente en mangeant des lapins.
    private int foodLevel;

    // Méthodes non reproduites.
}
```

Comportement d'un renard

- Géré par la méthode **`hunt`** (chasser).
- Les renards vieillissent et mettent bas aussi.
- Ils peuvent avoir faim.
- Ils chassent pour de la nourriture dans les positions adjacentes.

Configuration des renards

- Simplifications identiques à celles des lapins.
- Chasser et manger peuvent être modélisés de différentes manières
  - le niveau de nourriture doit-il être additif?
  - un renard affamé chasse-t-il plus qu'un autre?
- Ces simplifications sont-elles toujours acceptables?

La classe **`Simulator`**

- Trois composants fondamentaux:
  - configuration dans le constructeur
  - la méthode **`populate`**
    - chaque animal reçoit un âge initial aléatoire
  - la méthode **`simulateOneStep`**
    - itération sur toute la population
    - deux objets **`Field`** sont utilisés: **`field`** et **`updatedField`**.

Le pas de mise à jour

```
if(animal instanceof Rabbit) {
    Rabbit rabbit = (Rabbit)animal;
    if(rabbit.isAlive()) {
        rabbit.run(updatedField, newAnimals);
    }
    else {
        iter.remove();
    }
}
else if(animal instanceof Fox) {
    Fox fox = (Fox)animal;
    if(fox.isAlive()) {
        fox.hunt(field, updatedField,
newAnimals);
    }
    else {
        iter.remove();
    }
}
```

Ce qu'il faudrait améliorer

- **`Fox`** et **`Rabbit`** ont de fortes ressemblances mais n'ont pas de superclasse commune.
- **`Simulator`** est fortement couplée à des classes spécifiques
  - elle "en sait beaucoup" sur le comportement des renards et des lapins.

La superclasse **`Animal`**

- Placer les champs communs dans **`Animal`**:
  - **`age`**, **`alive`**, **`location`**
- Renommage de méthode pour gérer le masquage d'information:
  - **`run`** et **`hunt`** deviennent **`act`**.
- **`Simulator`** peut être maintenant découplé de manière importante.

Itération modifiée (découplée)

```
for(Iterator iter = animals.iterator();
                                    iter.hasNext(); ) {
    Animal animal = (Animal)iter.next();
    if(animal.isAlive()) {
        animal.act(field, updatedField, newAnimals);
    }
    else {
        iter.remove();
    }
}
```

La méthode **act** de **Animal**

- Le contrôle de type statique impose une méthode **act** dans **Animal**.
- Il n'y a pas d'implantation commune évidente.
- Définir alors **act** comme abstraite:

```
abstract public void act(Field currentField,
                         Field updatedField,
                         List newAnimals);
```

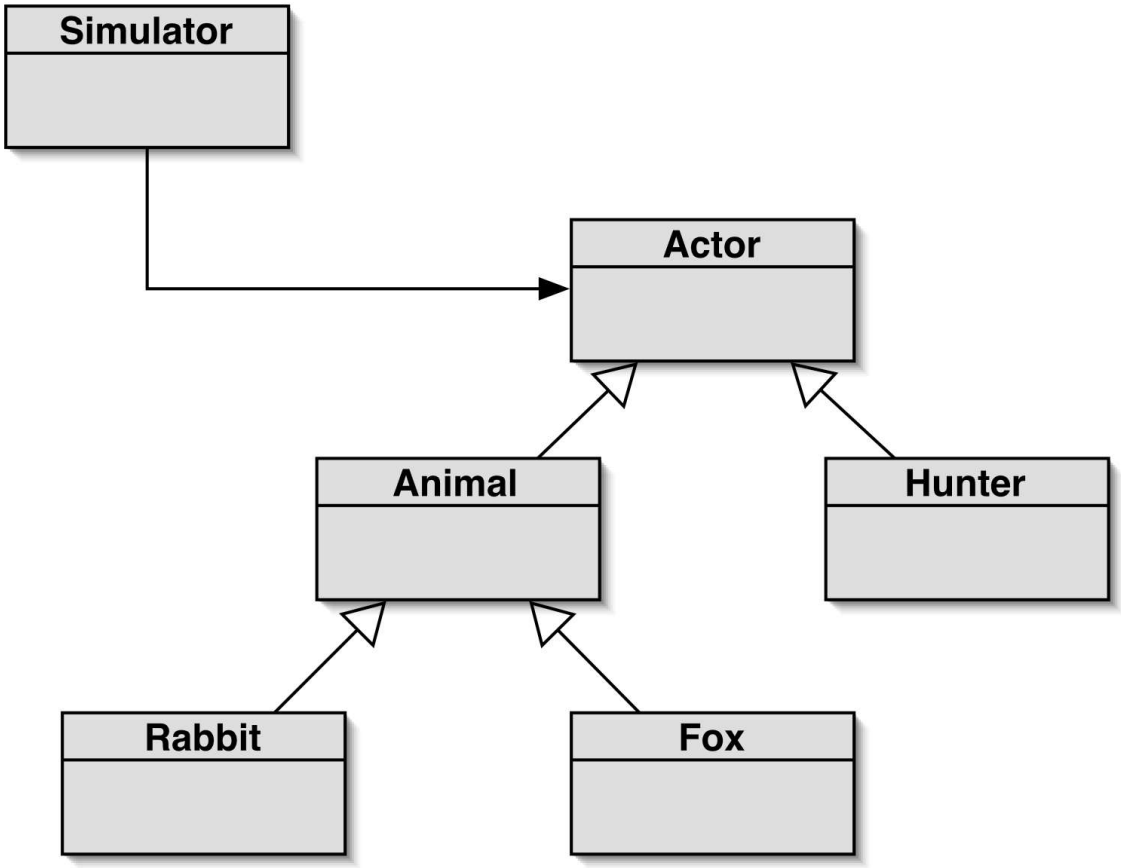
Classes et méthodes abstraites

- Les méthodes abstraites comportent le mot **`abstract`** dans leur signature.
- Elles n'ont pas de corps.
- Elles rendent leur classe abstraite.
- Les classes abstraites ne peuvent être instanciées.
- Les sous-classes concrètes complètent l'implantation.

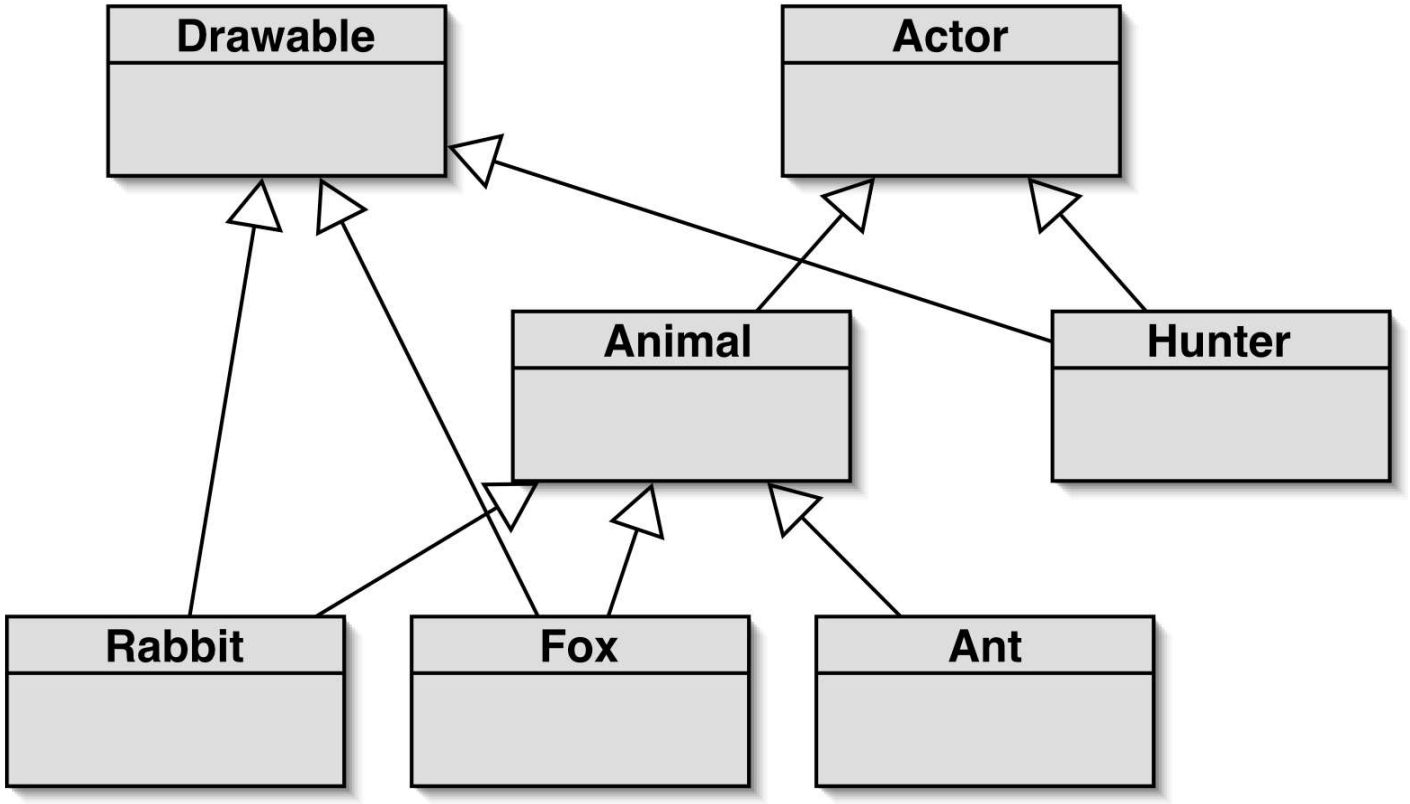
La classe **Animal**

```
public abstract class Animal
{
    // champs non reproduits

    /**
     * faire agir l'animal - c'est à dire:
     * lui faire faire ce qu'il veut/doit faire.
     */
    abstract public void act(Field currentField,
                             Field updatedField,
                             List newAnimals);

    // autres méthodes non reproduites
}
```

Encore plus d'abstraction



Dessin sélectif (héritage multiple)



Héritage multiple

- Situation: une classe hérite directement de plusieurs ancêtres.
- Chaque langage possède sa solution au problème
  - comment résoudre les conflits de définitions?
- Java interdit l'héritage multiple de classes.
- Java autorise l'héritage multiple dans les interfaces.
  - pas de conflit d'implantation.

Une interface **Actor**

```
public interface Actor
{
    /**
     * Réalise le comportement quotidien de l'acteur.
     * Transfert l'acteur dans le nouveau terrain
     * s'il doit participer aux pas suivants de la sim.

     * @param currentField l'état courant du terrain
     * @param location la position actuelle de l'acteur
     * @param updatedField l'état mis à jour du terrain
     */
     void act(Field currentField, Location location,
              Field updatedField);
}
```

Les classes implantent une interface

```
public class Fox extends Animal implements Drawable
{
    ...
}

public class Hunter implements Actor, Drawable
{
    ...
}
```

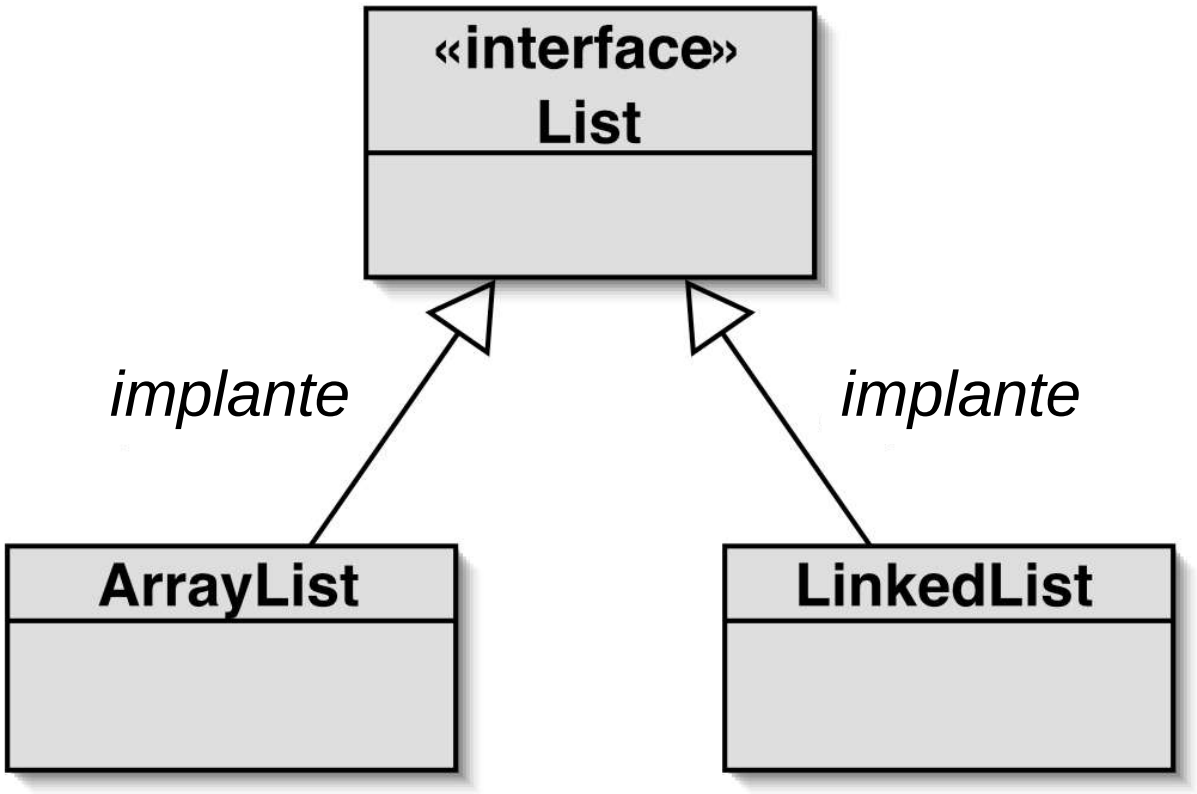
Interfaces comme types

- Les classes qui implantent n'héritent pas du code, mais ...
- ... elles deviennent sous-type du type interface.
- Donc le polymorphisme est disponible pour les interfaces comme pour les classes.

Interfaces comme spécifications

- Forte séparation entre fonctionnalités et implantation
  - mais les types des paramètres et du retour sont fixés.
- Les clients interagissent indépendamment de l'implantation.
  - mais les clients peuvent choisir entre plusieurs implantations.

Autres implantations



Résumé (1)

- L’héritage peut fournir une implantation partagée
  - classes abstraites et concrètes.
- L’héritage permet le partage de type
  - classes et interfaces.

Résumé (2)

- Les méthodes abstraites permettent le contrôle statique de type sans exiger d'implantation.
- les classes abstraites fonctionnent comme des superclasses incomplètes.
  - pas d'instance.
- Les classes abstraites gèrent le polymorphisme.

Interfaces

- Les interfaces fournissent la spécification sans l'implantation.
  - les interfaces sont totalement abstraites.
- Les interfaces gèrent le polymorphisme.
- Les interfaces Java autorisent l'héritage multiple.

Sommaire général