CENIN

Curso Java Básico

Uma introdução prática usando BlueJ

Técnicas de abstração adicionais

Interfaces e classes abstratas

Principais conceitos a serem abordados

- Classes abstratas
- Interfaces
- Herança múltipla

Simulações

- Programas são normalmente utilizados para simular atividades do mundo real, tais como:
  - tráfego de uma cidade;
  - previsão do tempo;
  - processos nucleares;
  - flutuações do mercado de ações; e
  - mudanças ambientais.

Simulações

- Frequentemente, são apenas simulações parciais.
- Em geral, elas envolvem simplificações.
  - Mais detalhes têm o potencial de fornecer mais precisão.
  - Na maioria das vezes, mais detalhes exigem mais recursos.
    - Capacidade de processamento.
    - Tempo de simulação.

Benefícios das simulações

- Suportam previsão útil.
  - Meteorologia.
- Permitem experimentação.
  - Mais segura, mais barata, mais rápida.
- Exemplo:
  - ‘Como a vida selvagem será afetada se dividirmos esse parque nacional com uma estrada?’

Simulações predador/presa

- Frequentemente, há um equilíbrio tênue entre as espécies.
  - Muitas presas significam muita comida.
  - Muita comida estimula um número maior de predadores.
  - Mais predadores comem mais presas.
  - Menos presas significam menos comida.
  - Menos comida significa ...

Simulação "raposas e coelhos"

- Modelaremos uma aplicação para monitorar populações de raposas e coelhos em uma área demarcada:
  - A aplicação tem uma área, uma coleção de coelhos e uma de raposas;
  - A cada passo, cada raposa e cada coelho executam as ações que caracterizam seus comportamentos;
  - Depois de cada passo é exibido o novo estado da simulação.

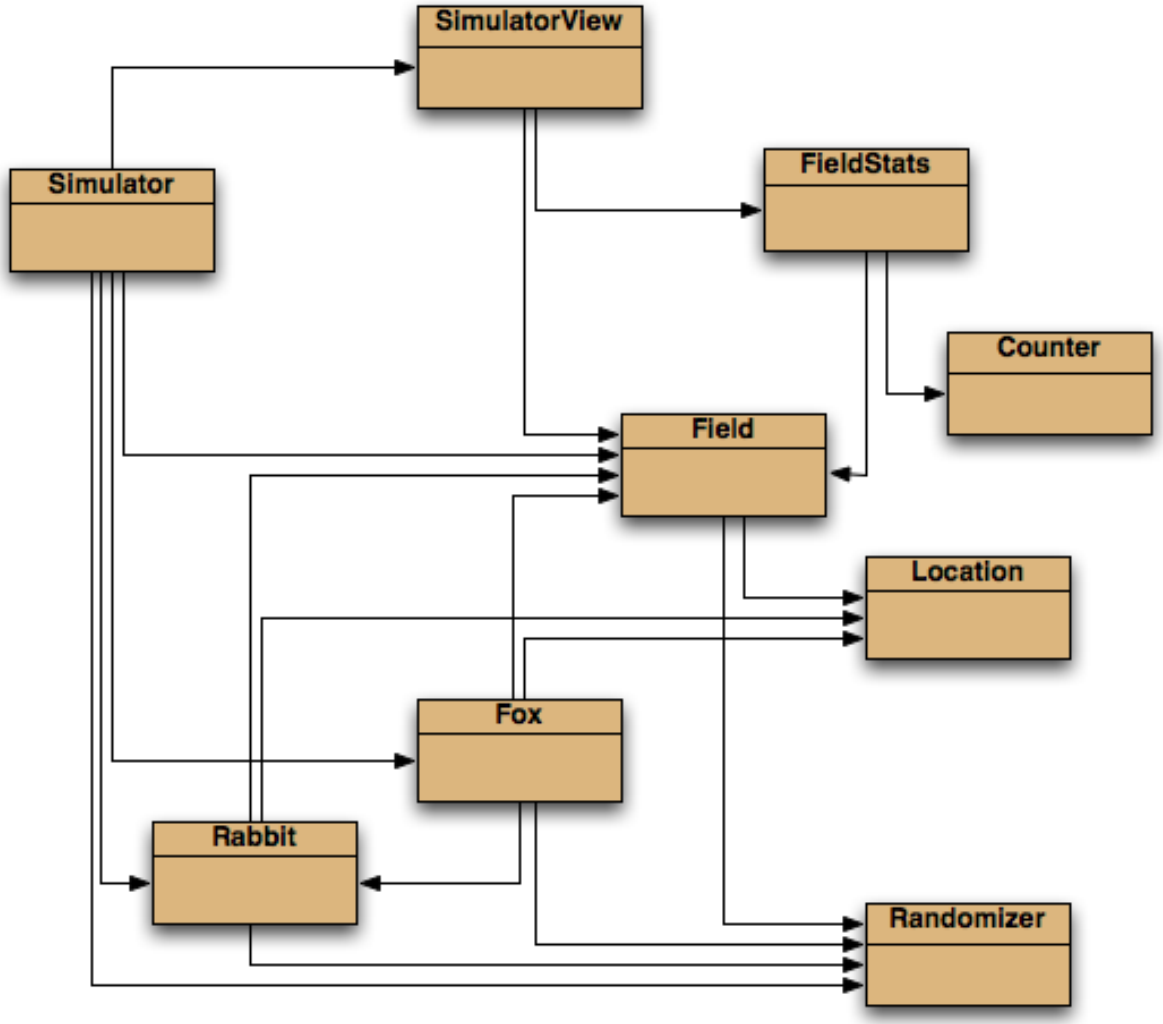
Exercício

- **Simulação raposas e coelhos**
  - Inicie o **BlueJ**, abra o projeto *foxes-and-rabbits-v1*, crie uma instância de *Simulator* (use o construtor sem parâmetros) e verifique a população de coelhos e a de raposas.
  - Chame o método *simulateOneStep*. A quantidade de raposas e coelhos muda ?

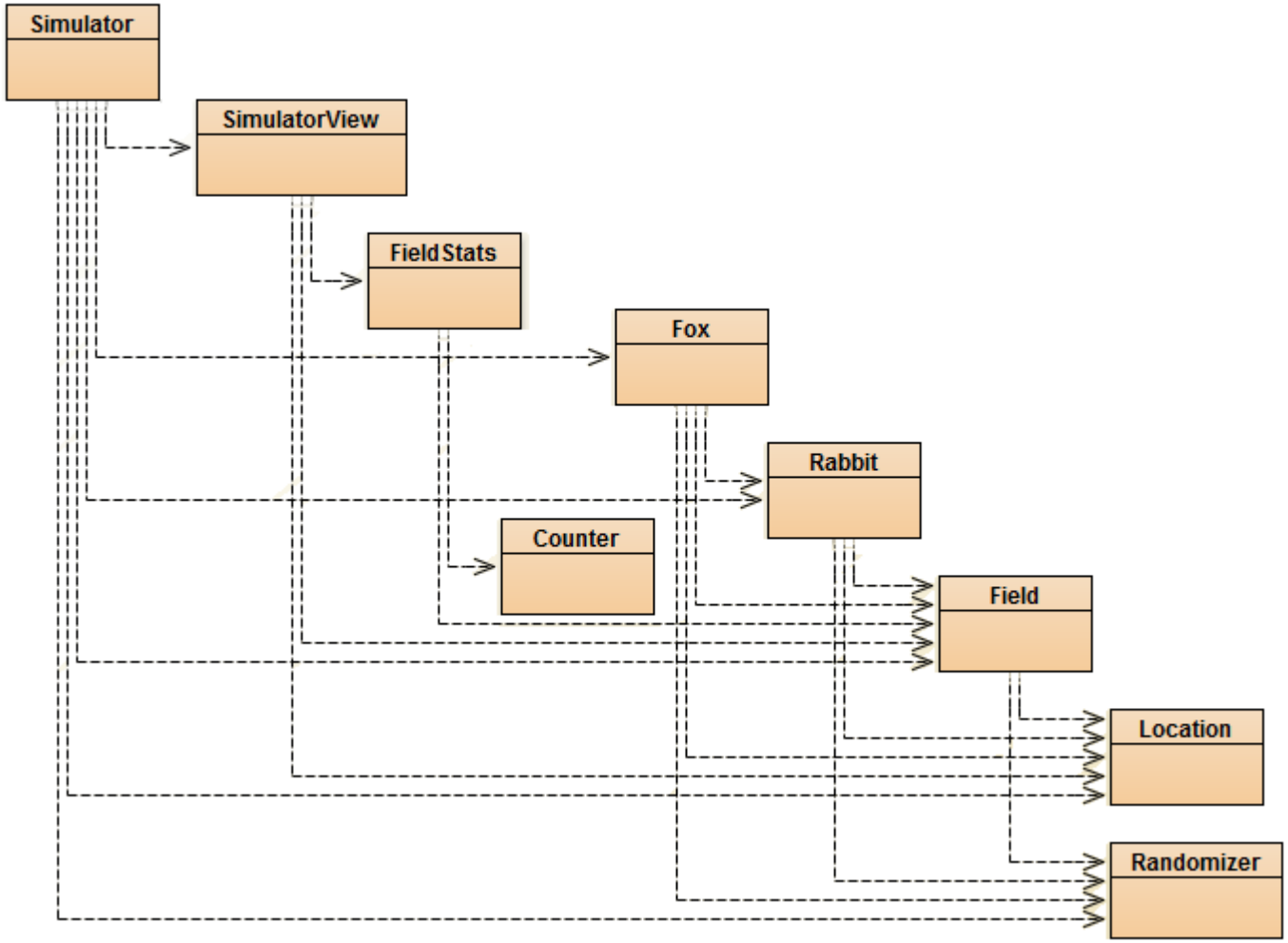
Exercício

- **Simulação raposas e coelhos**
  - Chame o método *simulate* para simular uma quantidade significativa de passos, como 50 ou 100. A quantidade de raposas e coelhos muda em taxas semelhantes ?
  - Quais alterações você percebe se simular uma quantidade muito grande de passos (use *runLongSimulation*) ?

O projeto “foxes-and-rabbits”



O projeto “foxes-and-rabbits”



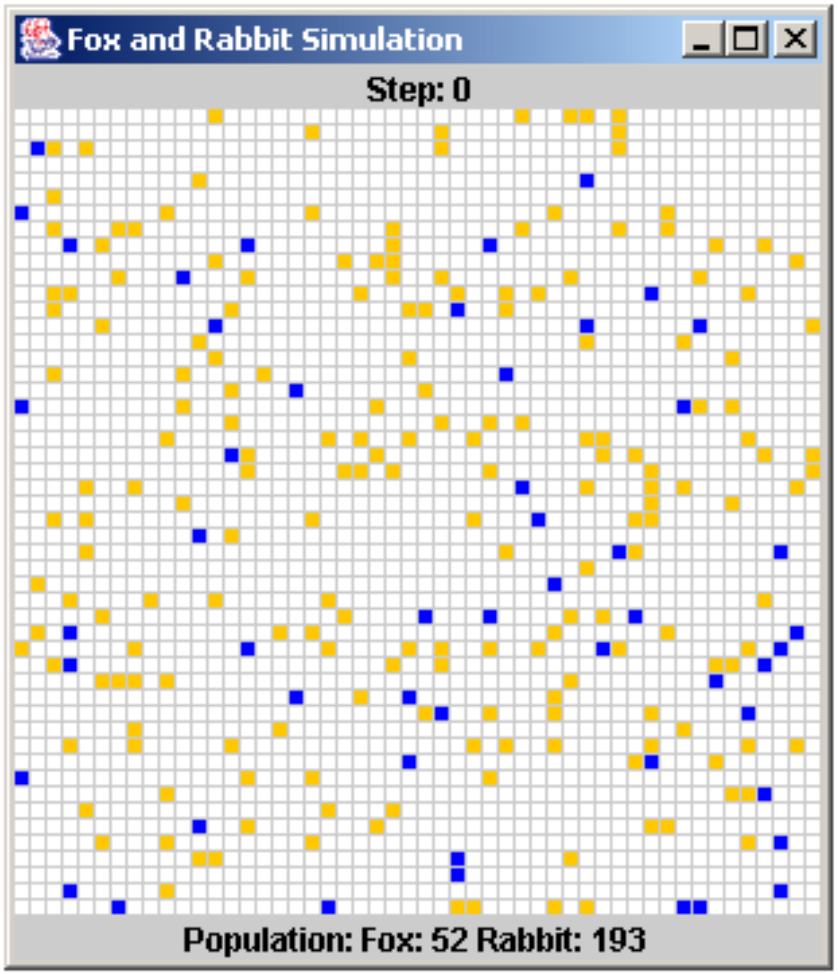
Principais classes de interesse

- `Fox`
  - Modelo simples de um tipo de predador.
- `Rabbit`
  - Modelo simples de um tipo de presa.
- `Simulator`
  - Gerencia a tarefa geral de simulação.
  - Armazena uma coleção de raposas e coelhos.

As classes remanescentes

- `Field`
  - Representa um campo em 2D.
- `Location`
  - Representa uma posição em 2D.
- `SimulatorView, FieldStats, Counter`
  - Armazenam as estatísticas e apresentam uma visualização do campo.

Exemplo da visualização



Estado de um coelho

```
public class Rabbit
{
    campos estáticos omitidos.

    // Características individuais (campos de instância).
    // A idade do coelho.
    private int age;
    // Se o coelho está vivo ou não.
    private boolean alive;
    // A posição do coelho.
    private Location location;
    // O campo ocupado.
    private Field field;

    métodos omitidos.
}
```

Comportamento de um coelho

- Gerenciado a partir do método `run`.
- Idade incrementada em cada "passo" da simulação.
  - Um coelho poderia morrer nesse ponto.
- Coelhos com idade suficiente poderiam procriar em cada passo.
  - Novos coelhos poderiam nascer nesse ponto.

Simplificações de coelho

- Coelhos não têm sexos diferentes.
  - Na verdade, todos são fêmeas.
- O mesmo coelho poderia procriar em cada passo.
- Todos os coelhos morrem com a mesma idade.
- Outros?

Exercício

- **Simulação raposas e coelhos**
  - Inspecione um objeto *Rabbit* a partir da instância de *Simulator*. Observe as variáveis de instância e de classe.
  - Experimente os efeitos de alterar algumas constantes da classe *Rabbit*. Você observa algum impacto sobre as populações de raposas e coelhos ?

Estado de uma raposa

```
public class Fox
{
    campos estáticos omitidos.

    // Características individuais (campos de instância).
    // A idade da raposa.
    private int age;
    // Se a raposa está viva ou não.
    private boolean alive;
    // A posição da raposa.
    private Location location;
    // O campo ocupado.
    private Field field;
    // O nível de alimentos da raposa,
    // que aumenta comendo coelhos.
    private int foodLevel;

    métodos omitidos.
}
```

Comportamento de uma raposa

- Gerenciado a partir do método `hunt`.
- Raposas também envelhecem e procriam.
- Elas têm fome.
- Elas caçam em locais adjacentes.

Configuração de uma raposa

- Simplificações semelhantes a coelhos.
- Caçar e comer poderiam ser modelados de diferentes maneiras.
  - O nível de alimentos deve ser cumulativo?
  - Qual é a probabilidade de uma raposa faminta caçar?
- As simplificações são aceitáveis?

Exercício

- **Simulação raposas e coelhos**
  - Inspecione um objeto *Fox* a partir da instância de *Simulator*. Observe as variáveis de instância e de classe.
  - Experimente os efeitos de alterar algumas constantes da classe *Fox*. Você observa algum impacto sobre as populações de raposas e coelhos ?

Exercício

- **Simulação raposas e coelhos**
  - Abra o projeto *foxes-and-rabbits-graph*, crie uma instância de *Simulator* e chame *runLongSimulation*. Observe o tamanho das populações de raposas e coelhos no gráfico apresentado.
  - Feche o projeto *foxes-and-rabbits-graph*.
  - Abra o projeto *foxes-and-rabbits-v1* e salve-o como *foxes-and-rabbits* (usaremos este projeto para diversas alterações).

A classe `Simulator`

- Três componentes-chave:
  - O construtor.
    - Constrói o campo, as coleções de raposas e de coelhos e a interface gráfica.
  - O método `populate`.
    - A cada animal é dada uma idade inicial aleatória.
  - O método `simulateOneStep`.
    - Itera pelas populações.

Código-fonte: Simulator

```
public void simulateOneStep()
{
    List<Rabbit> newRabbits = new ArrayList<Rabbit>();

    for(Iterator<Rabbit> it = rabbits.iterator();
            it.hasNext(); ) {
        Rabbit rabbit = it.next();
        rabbit.run(newRabbits);
        if(! rabbit.isAlive()) {
            it.remove();
        }
    }
    ...
```

Código-fonte: Simulator

```
…

List<Fox> newFoxes = new ArrayList<Fox>();

for(Iterator<Fox> it = foxes.iterator();
        it.hasNext(); ) {
    Fox fox = it.next();
    fox.hunt(newFoxes);
    if(! fox.isAlive()) {
        it.remove();
    }
}
…
```

Código-fonte: Simulator

```
        ...

        rabbits.addAll(newRabbits);
        foxes.addAll(newFoxes);

        step++;

        view.showStatus(step, field);
    }
```

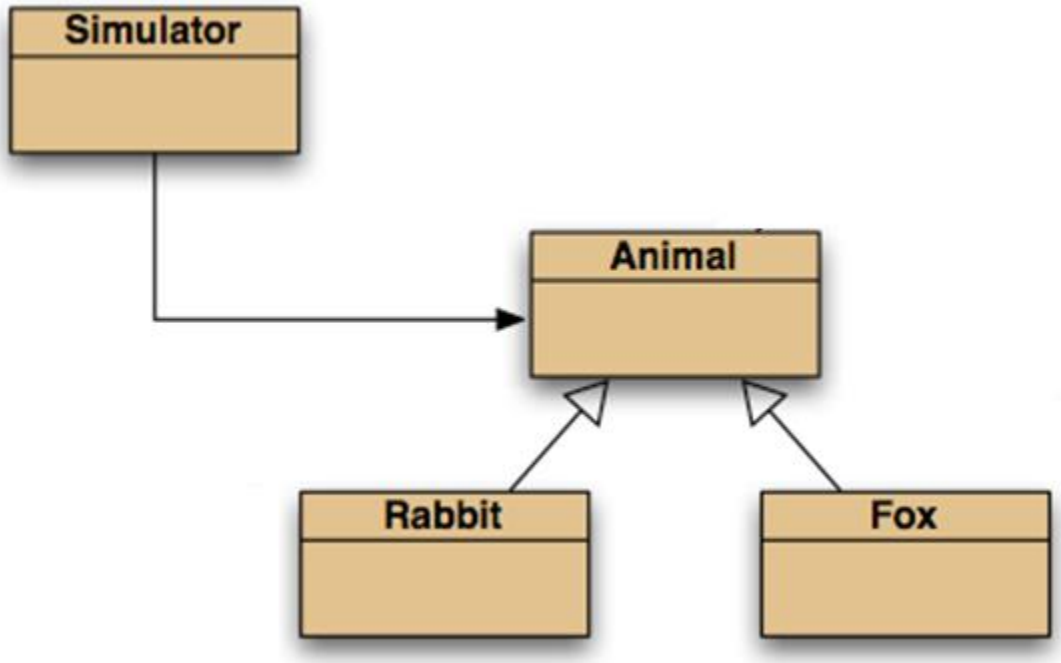
Oportunidade para aprimoramento

- `Fox` e `Rabbit` têm grandes semelhanças, mas não têm uma superclasse comum.
- `Simulator` está fortemente acoplada a classes específicas.
  - Ela 'conhece' bastante as diferenças de comportamento de raposas e coelhos.

Exercício

- **Simulação raposas e coelhos**
  - Identifique as semelhanças entre as classes *Fox* e *Rabbit*. Que membros poderiam ser movidos para uma superclasse comum ? Verifique as variáveis de classe e de instância, bem como os métodos.

Aprimorando a simulação: a superclasse `Animal`



A superclasse `Animal`

- Campos comuns em `Animal`:
  - `age`, `alive`, `field`, `location`
- Método remanescente para suportar ocultamento de informações:
  - `run` e `hunt` tornam-se `act`.
- `Simulator` agora pode ser significativamente desaclopada.

Código-fonte: Simulator

```
…

List<Animal> newAnimals = new ArrayList<Animal>();

for(Iterator<Animal> it = animals.iterator();
        it.hasNext(); ) {
    Animal animal = iter.next();
    animal.act(newAnimals);
    if(! animal.isAlive()) {
        it.remove();
    }
}
…
```

O método `act` em `Animal`

- Verificação de tipo estático requer um método `act` em `Animal`.
- Não há nenhuma implementação óbvia compartilhada.
- Definir `act` como abstrato:

```
abstract public void act(List<Animal> newAnimals);
```

Palavra-chave
`abstract`

Método abstrato não tem corpo,
apenas ponto-e-vírgula após o cabeçalho

Código-fonte: Animal

```
public abstract class Animal
{
   campos omitidos.

   /**
    * Faz o animal atuar - isto é: faz ele realizar
    * seja lá o que ele queira/precise realizar.
    */
  abstract public void act(List<Animal> newAnimals);

   outros métodos omitidos
}
```

Métodos e classes abstratas

- Métodos abstratos têm `abstract` no seu cabeçalho.
- Métodos abstratos não têm nenhum corpo.
- Se uma classe tiver métodos abstratos ela deve ser declarada abstrata.
- Classes abstratas podem ou não ter métodos abstratos.
- Classes abstratas não podem ser instanciadas.
- Subclasses concretas completam a implementação das classes abstratas.

Exercício

- **Simulação raposas e coelhos**
  - Crie a classe abstrata *Animal* e crie os relacionamentos para que ela seja superclasse de *Fox* e *Rabbit*.
  - Mova para *Animal* os campos *alive*, *location* e *field* e os métodos *isAlive*, *setDead*, *getLocation*, *setLocation* (verifique a visibilidade adequada).
  - Crie o método *getField* em *Animal*.
  - No construtor de *Animal*, configure *alive* como *true* e *location* e *field* com valores recebidos por parâmetros.
  - Compile *Animal*.

Exercício

- **Simulação raposas e coelhos**
  - Inclua nos construtores de *Fox* e *Rabbit* uma chamada ao construtor da superclasse.
  - Altere os métodos *hunt*, *run*, *findFood* e *giveBirth* de para que tenham acesso aos campos da superclasse.
  - Compile *Fox* e *Rabbit*.

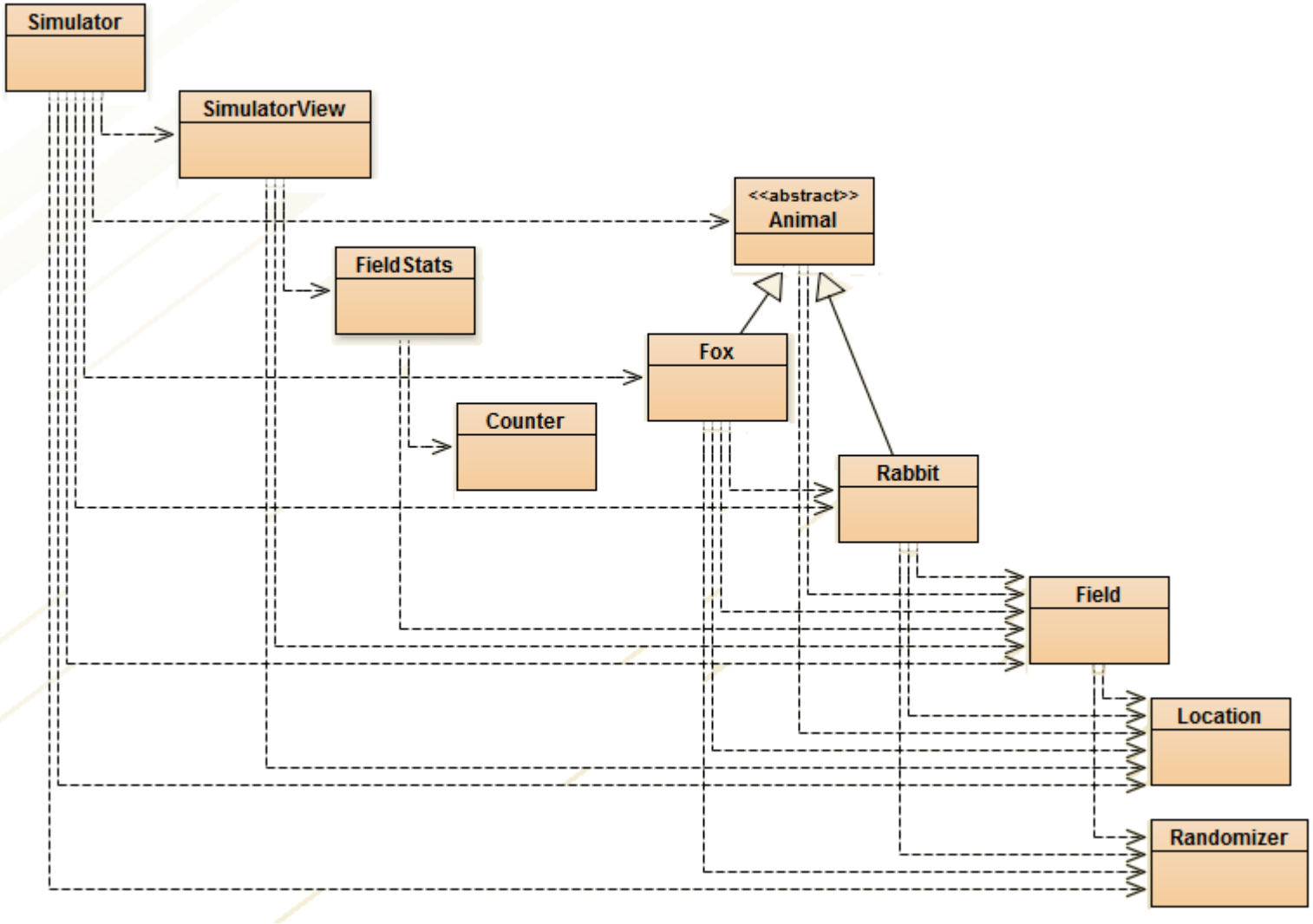
Exercício

- **Simulação raposas e coelhos**
  - Crie o método abstrato *act* em *Animal* e renomeie os métodos *hunt* e *run* para *act* em *Fox* e *Rabbit*, garantindo que eles tem a mesma assinatura de *act* (use coleções de *Animal*).
  - Altere o método *giveBirth* para que use a coleção de *Animal*.
  - Compile *Animal*, *Fox* e *Rabbit*.

Exercício

- **Simulação raposas e coelhos**
  - Substitua em Simulator as coleções de *Fox* e *Rabbit* por uma de *Animal*.
  - Altere o construtor e os métodos *simulateOneStep*, *reset* e *populate* para que usem a coleção de *Animal*.
  - Compile o projeto *foxes-and-rabbits* e compare o estágio atual do projeto com *foxes-and-rabbits-v2*.

O projeto “foxes-and-rabbits”



Mais métodos abstratos

- Poderíamos ter movido o campo `age` para `Animal` junto com os métodos que usam este campo: `incrementAge` e `canBreed`.
- Entretanto, estes métodos usam constantes (`MAX_AGE` e `BREEDING_AGE`) com valores diferentes em `Fox` e `Rabbit`; estas constantes com valores específicos devem permanecer nas subclasses.

Mais métodos abstratos

- Assim, para mover os métodos `incrementAge` e `canBreed` para `Animal`, é preciso que eles tenham acesso às constantes específicas das subclasses.
- Onde colocaremos métodos de acesso para `MAX_AGE` e `BREEDING_AGE` ?

Código-fonte: Animal

```
...
protected void incrementAge()
{
    age++;
    if(age > getMaxAge()) {
        setDead();
    }
}
abstract protected int getMaxAge();
...
protected boolean canBreed()
{
    return age >= getBreedingAge();
}
abstract protected int getBreedingAge();
...
```

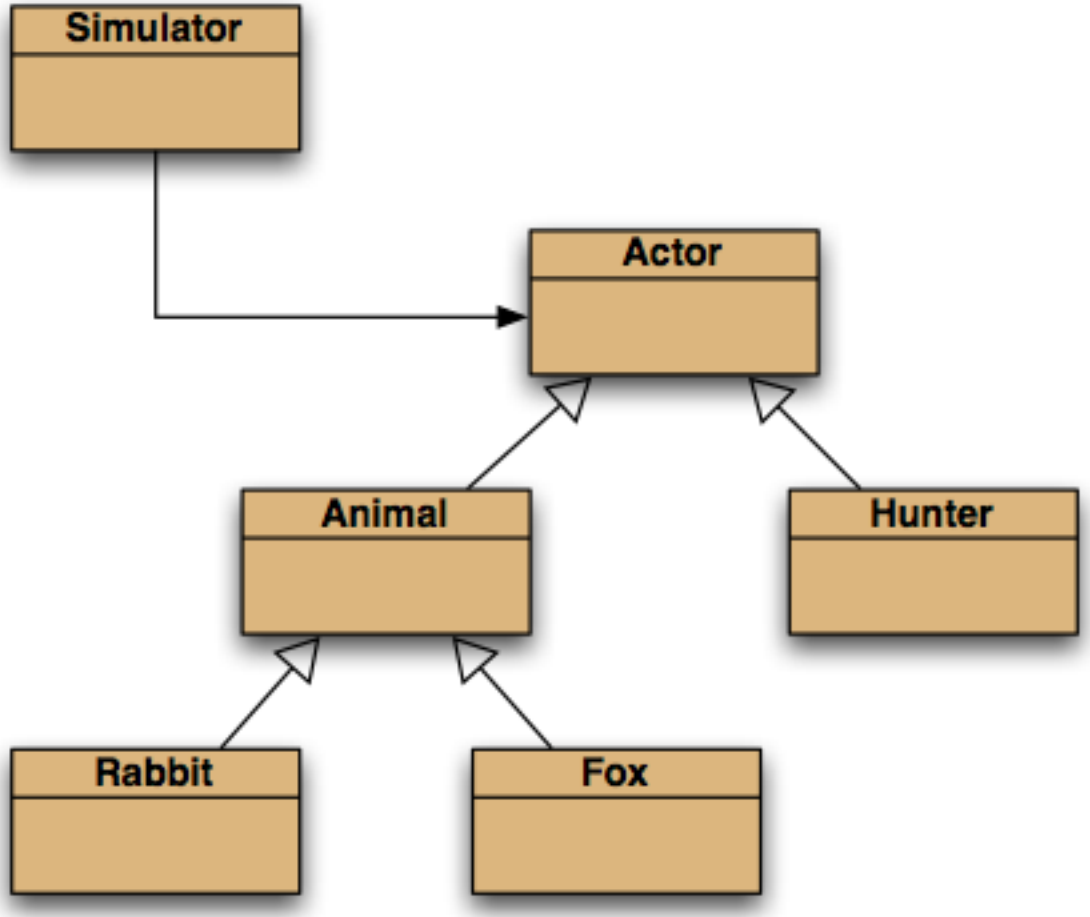
Exercício

- **Simulação raposas e coelhos**
  - Mova o campo *age* de *Fox* e *Rabbit* para *Animal*, configurando-o no construtor.
  - Mova o método *incrementAge* de *Fox* e *Rabbit* para *Animal*, inclua o método abstrato *getMaxAge* em *Animal* e versões concretas em *Fox* e *Rabbit*.
  - Mova o método *canBreed* de *Fox* e *Rabbit* para *Animal*, inclua o método abstrato *getBreedingAge* em *Animal* e versões concretas em *Fox* e *Rabbit*.

Exercício

- **Simulação raposas e coelhos**
  - Compile o projeto *foxes-and-rabbits* e compare o estágio atual do projeto com *foxes-and-rabbits-v3*.
  - O que é necessário para mover o método *breed* de *Fox* e *Rabbit* para *Animal* ? Faça essa alteração.
  - Compile o projeto *foxes-and-rabbits* e compare o estágio atual do projeto com *foxes-and-rabbits-v4*.

Mais abstração



Exercício

- **Simulação raposas e coelhos**
  - Crie a classe abstrata *Actor* e crie o relacionamento para que ela seja superclasse de *Animal*.
  - Mova de *Animal* para *Actor* o método abstrato *act* e defina o método abstrato *isActive* em *Actor*.
  - Implemente o método *isActive* em *Animal*, retornando o resultado de *isAlive*.
  - Altere para *Actor* o parâmetro das coleções usadas em *Fox*, *Rabbit*.
  - Compile *Animal*, *Fox* e *Rabbit*.

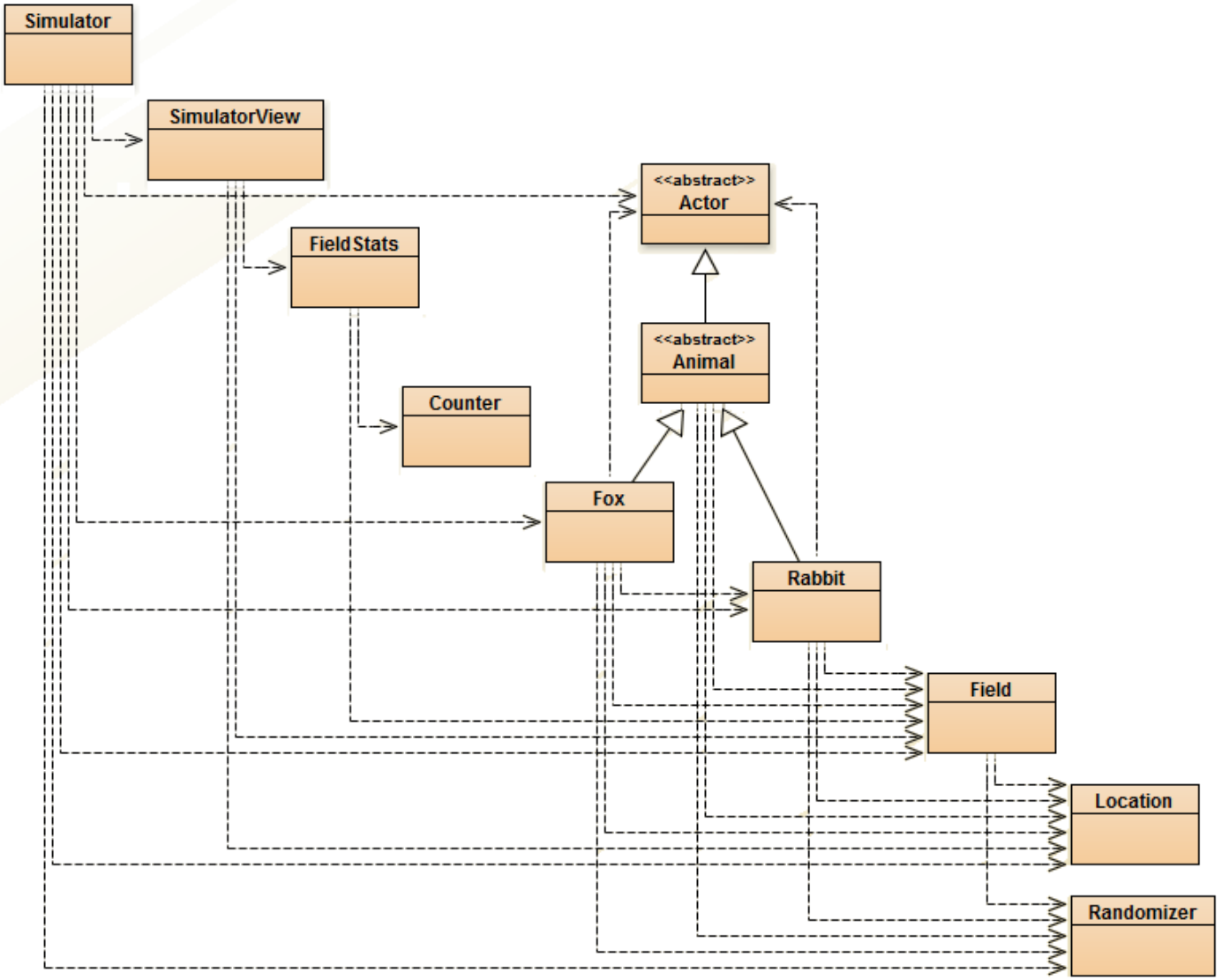
Exercício

- **Simulação raposas e coelhos**
  - Crie a classe abstrata *Actor* e crie o relacionamento para que ela seja superclasse de *Animal*.
  - *Defina em Actor* os métodos abstratos *act* (movido de Animal) e *isActive*.
  - Implemente o método *isActive* em *Animal*, retornando o resultado de *isAlive*.
  - Altere para *Actor* o parâmetro das coleções usadas em *Fox e Rabbit*.
  - Compile Actor, *Animal*, *Fox* e *Rabbit*.

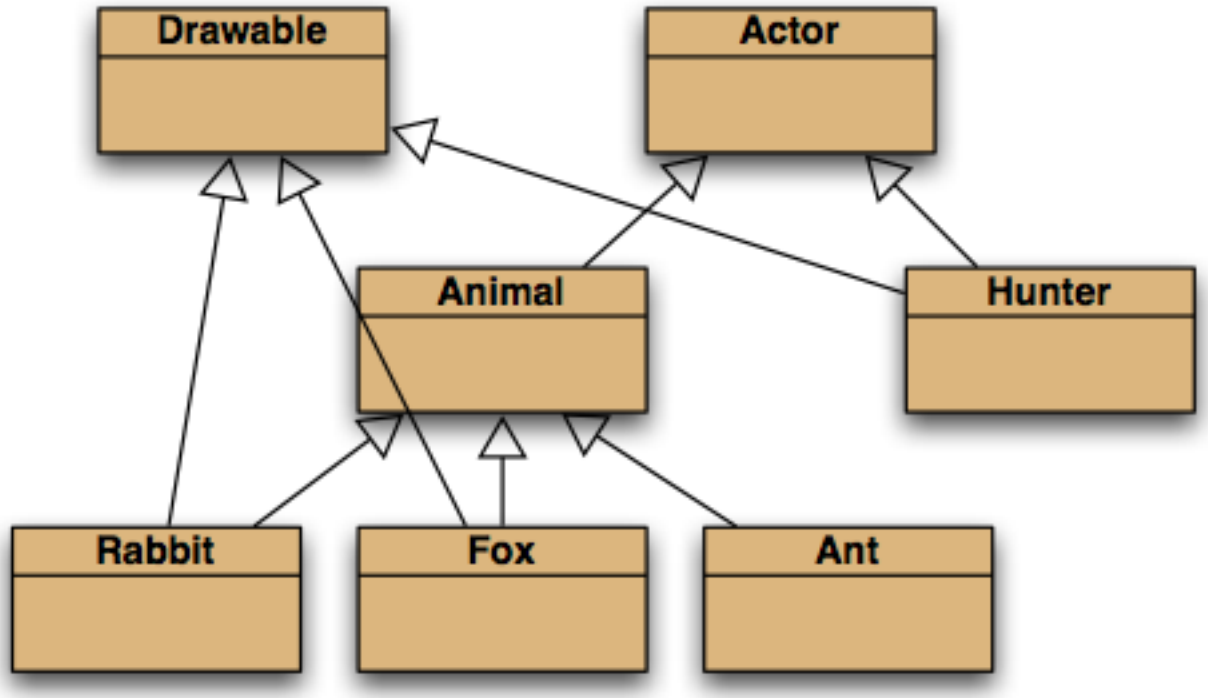
Exercício

- **Simulação raposas e coelhos**
  - Altere em *Simulator* o parâmetro das coleções de *Animal* para *Actor* e a chamada de *isAlive* para *isActive*.
  - Compile o projeto *foxes-and-rabbits* e compare o estágio atual do projeto com *foxes-and-rabbits-v5*.

O projeto “foxes-and-rabbits”



Desenho seletivo (herança múltipla)



Herança múltipla

- Faz com que uma classe herde diretamente de múltiplos ancestrais.
- Cada linguagem tem suas próprias regras.
  - Como resolver as definições de concorrência?
- O Java proíbe isso para classes.
- O Java permite isso para interfaces.
  - Nenhuma implementação concorrente.

A interface Ator

```
public interface Actor
{
    /**
     * Realiza o comportamento do ator.
     * @param newActors Lista para guardar atores criados.
     */
    void act(List<Actor> newActors);

    /**
     * O ator continua ativo?
     * @return true se contuna ativo, false se não.
     */
    boolean isActive();
}
```

Classes implementam interfaces

- Classes só podem herdar de uma superclasse.

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

- Entretanto, podem implementar diversas interfaces.

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

Interfaces

- Interfaces são semelhantes a classes, mas:
  - A palavra-chave `interface` é utilizada no cabeçalho ao invés de `class`.
  - Interfaces não possuem construtores.
  - Todos os métodos são abstratos e públicos

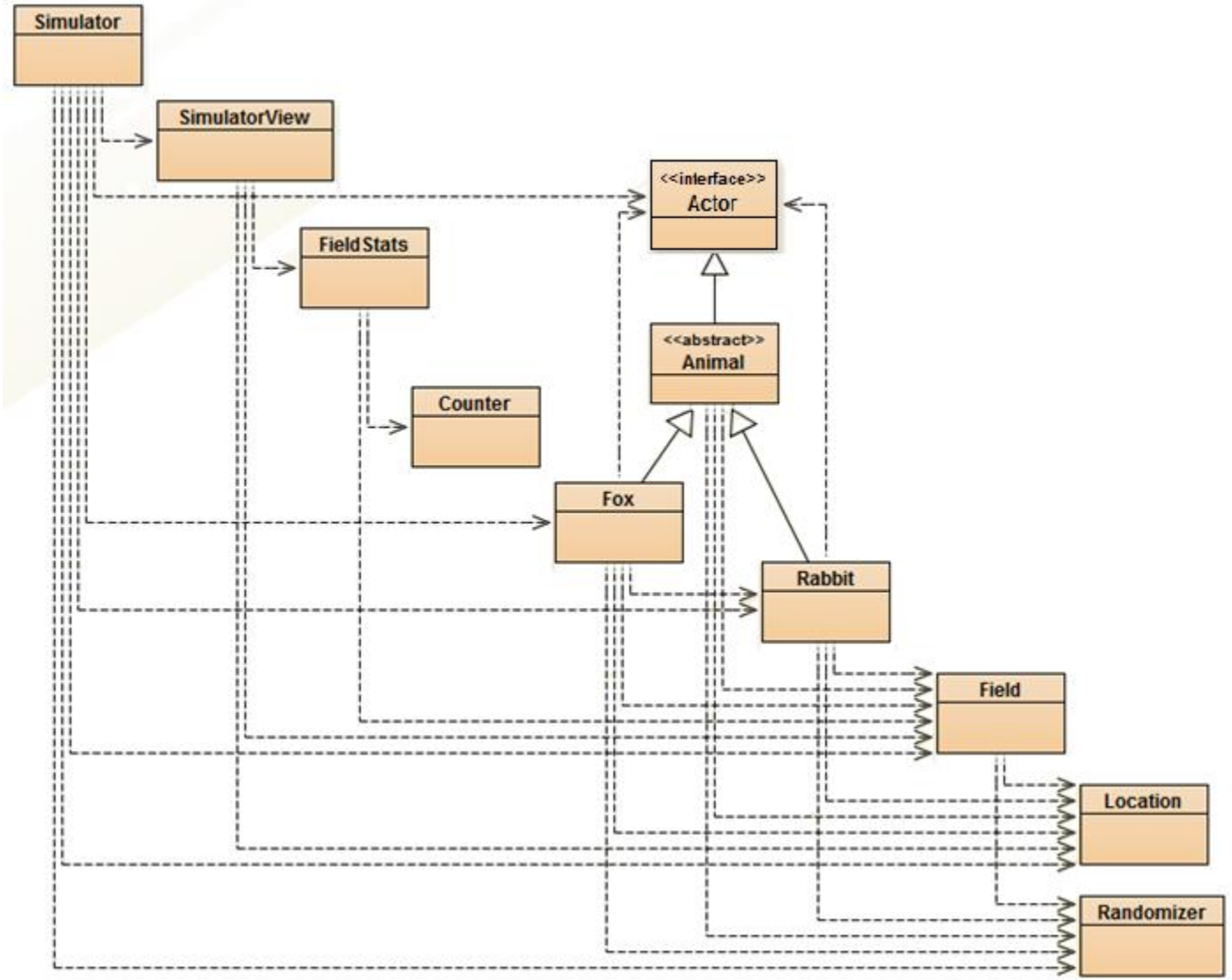
    (as palavras-chave `abstract` e `public` podem ser omitidas).
  - Apenas campos constantes são permitidos

    (as palavras-chave `public`, `static` e `final` podem ser omitidas).

Exercício

- **Simulação raposas e coelhos**
  - Redefina a classe abstrata *Actor* em seu projeto como uma interface. O projeto compila ? Faça as alterações necessárias para compilá-lo.
  - Compile o projeto *foxes-and-rabbits* e compare o estágio atual do projeto com *foxes-and-rabbits-v6*.

O projeto “foxes-and-rabbits”



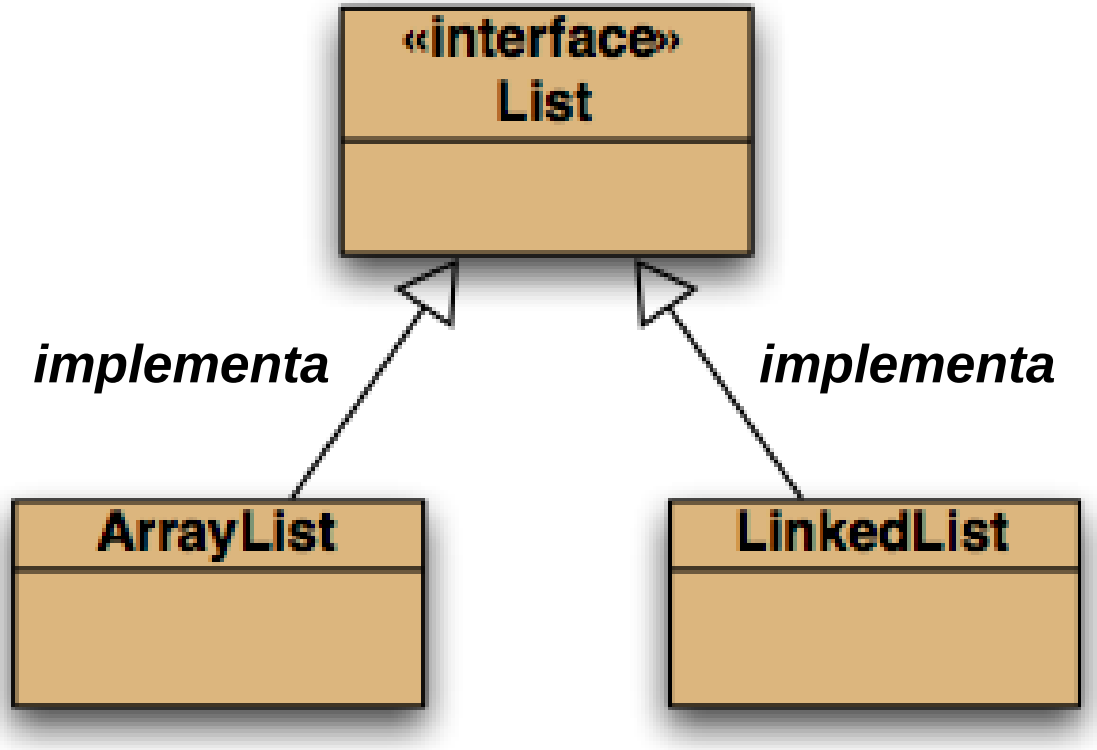
Interfaces como tipos

- Classes de implementação não herdam código da interface, mas ...
- ... classes de implementação são subtipos do tipo de interface.
- Portanto, o polimorfismo está disponível para interfaces e classes.

Interfaces como especificações

- Forte separação entre funcionalidade (interface pública) e implementação.
- Interação de clientes não depende da implementação.
  - Mas os clientes podem escolher implementações alternativas.

Implementações alternativas



Implementações alternativas

- A interface `List` especifica a funcionalidade de uma lista.
- As classes `ArrayList` e `LinkedList` fornecem implementações diferentes da interface `List`.
- Pode ser difícil julgar por antecipação qual implementação é mais adequada para uma aplicação.

Implementações alternativas

- Se usarmos a interface como tipo de variáveis e parâmetros, podemos variar a implementação alterando o código em apenas um lugar: na instanciação da classe concreta.

```
private List<Type> myList = new ArrayList<Type>;
```

- Isto é o princípio de design conhecido como "**Program to an interface, not an implementation**"

Revisão (1)

- Herança pode fornecer implementação compartilhada.
  - Classes concretas e abstratas.
- Herança fornece informações sobre o tipo compartilhado.
  - Classes e interfaces.

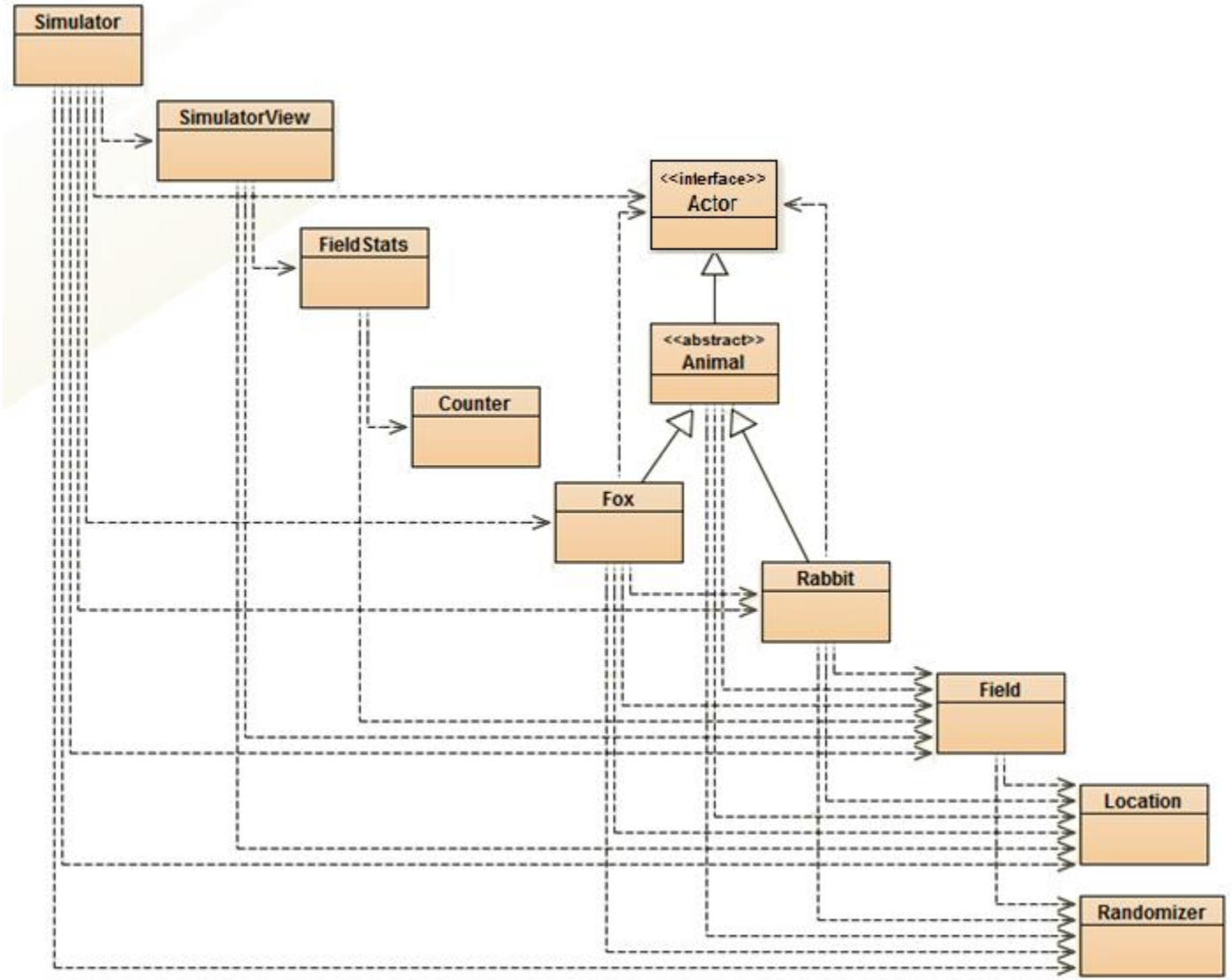
Revisão (2)

- Métodos abstratos permitem a verificação do tipo estático sem exigir uma implementação.
- Classes abstratas funcionam como superclasses incompletas.
  - Nenhuma instância.
- Classes abstratas suportam o polimorfismo.

Revisão (3)

- Interfaces fornecem uma especificação sem uma implementação.
  - Interfaces são totalmente abstratas.
- Interfaces suportam polimorfismo.
- Interfaces Java suportam herança múltipla.

O projeto “foxes-and-rabbits”



Oportunidade para aprimoramento

- `Fox` e `Rabbit` agora têm uma superclasse comum.
- `Simulator` ainda está fortemente acoplada a classes específicas.
  - Ela 'conhece' raposas e coelhos e deveria 'conhecer' apenas atores.

Qualidade do código

- Dois conceitos importantes quanto à qualidade do código:
  - Acoplamento
    - vinculação entre as unidades
    - bom design tem ***baixo acoplamento***
  - Coesão
    - afinidade de tarefas de uma unidade
    - bom design tem ***alta coesão***

Design baseado em responsabilidade

- Questão: onde (qual classe) devemos adicionar um novo método ?
- Cada classe deve ser responsável por manipular seus próprios dados.
- A classe que possui os dados deve ser responsável por processá-los.
- Design baseado em responsabilidade leva a uma alta coesão e a um baixo acoplamento.

Oportunidade para aprimoramento

- `Simulator` ainda está fortemente acoplada a classes específicas.
  - Ela 'conhece' raposas e coelhos e deveria 'conhecer' apenas atores.
  - Seu construtor conhece as cores de raposas e coelhos.
  - De quem (que classe) deveria ser a responsabilidade de conhecer a cor dos animais da simulação ?

Exercício

- **Simulação raposas e coelhos**
  - Defina em Fox e Rabbit a constante SIMULATION_COLOR, do tipo Color e inicializada com a cor de cada animal.
  - Implemente em Fox e Rabbit o método getSimulationColor().
  - Defina em Actor e em Animal o método abstrato getSimulationColor() com retorno do tipo Color.

Exercício

- **Simulação raposas e coelhos**
  - Elimine em SimulatorView o campo colors e os métodos setColor() e getColor().
  - Em showStatus() de SimulatorView, faça coerção do objeto retornado por field.getObjectAt(row, col) para Actor, de modo a poder chamar o método getSimulationColor() em drawMark(col, row, getColor(animal.getClass())).

Exercício

- **Simulação raposas e coelhos**
  - Crie uma classe
  - Defina em Fox e Rabbit a constante SIMULATION_COLOR, do tipo Color e inicializada com a cor de cada animal.
  - Implemente em Fox e Rabbit o método getSimulationColor().
  - Defina em Actor e em Animal o método abstrato getSimulationColor() com retorno do tipo Color.

Exercício

- **Simulação raposas e coelhos**
  - Defina em Fox e Rabbit a constante SIMULATION_COLOR, do tipo Color e inicializada com a cor de cada animal.
  - Implemente em Fox e Rabbit o método getSimulationColor().
  - Defina em Actor e em Animal o método abstrato getSimulationColor() com retorno do tipo Color.

Exercício

- **Simulação raposas e coelhos**
  - No construtor da classe *Simulator*, elimine as chamadas ao método *setColor()* de *SimulatorView*.
  - Compile o projeto *foxes-and-rabbits* e compare o estágio atual do projeto com *foxes-and-rabbits-v7*.

Oportunidade para aprimoramento

- `Simulator` ainda está fortemente acoplada a classes específicas.
  - Ela 'conhece' raposas e coelhos e deveria 'conhecer' apenas atores.
  - Seu método `populate` decide se uma localização ficará vazia ou com uma raposa ou um coelho.
  - De quem (que classe) deveria ser a responsabilidade de entregar um animal para uma localização não vazia?

Exercício

- **Simulação raposas e coelhos**
  - Defina em *Simulator* uma constante ACTOR_CREATION_PROBABILITY com a soma das probabilidades de criação de raposas e coelhos.
  - Crie uma classe ActorFactory e mova para ela as probabilidades de criação de raposas e coelhos. Altere seus valores de modo que mantenham a proporção mas sua soma seja 1.

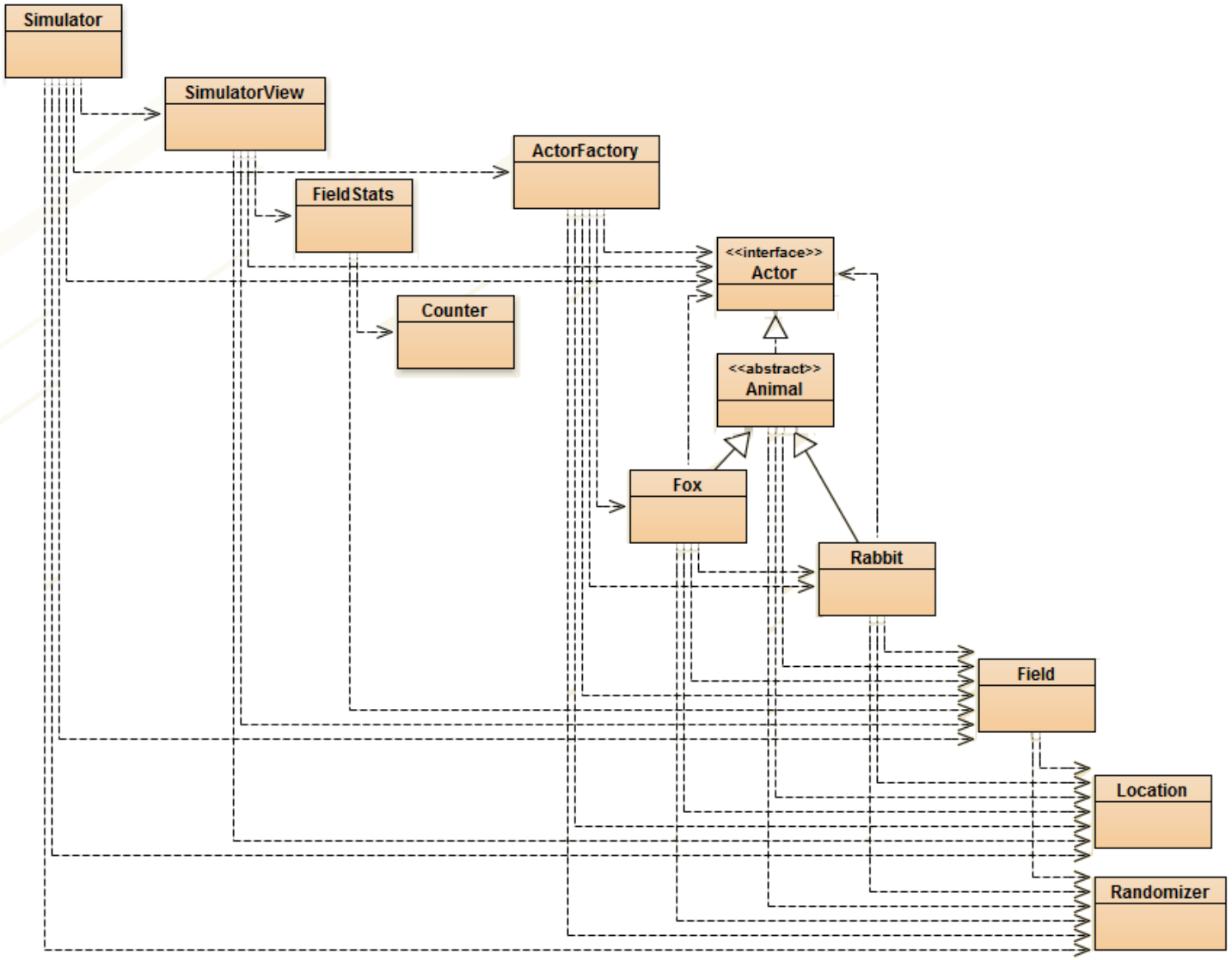
Exercício

- **Simulação raposas e coelhos**
  - Crie em *ActorFactory* o método *createActor(Field, Location)* que crie sempre um ator, decidindo se será raposa ou coelho conforme suas probabilidades de criação.
  - Altere em *Simulator* o método *populate* para que obtenha atores, conforme sua probabilidade de criação, chamando o método *createActor de ActorFactory*.

Exercício

- **Simulação raposas e coelhos**
  - Compile o projeto *foxes-and-rabbits* e compare o estágio atual do projeto com *foxes-and-rabbits-v8*.

O projeto “foxes-and-rabbits”



Contatos

Câmara dos Deputados

CENIN - Centro de Informática

Carlos Renato S. Ramos

carlosrenato.ramos@camara.gov.br