

# Programação Orientada por Objetos

---

## **Classes Abstratas e Interfaces**

Prof. José Cordeiro,

Prof. Cédric Grueau,

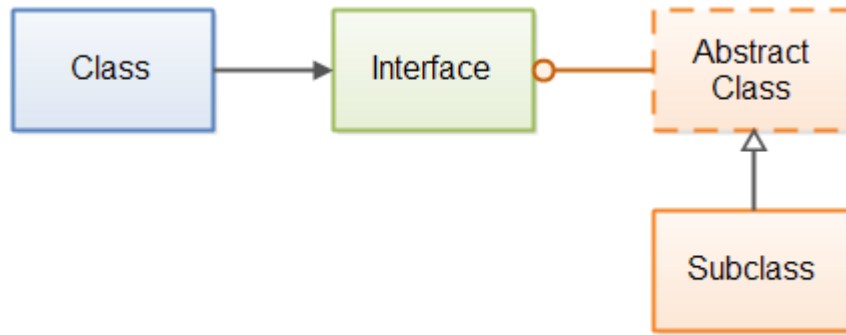
Prof. Laercio Júnior

Departamento de Sistemas e Informática

Escola Superior de Tecnologia de Setúbal – Instituto Politécnico de Setúbal

2019/2020

- Sessão 1: Exemplo Raposas e Coelhos
- Sessão 2: Código da Aplicação
- Sessão 3: Classes e Métodos Abstratos
- Sessão 4: Interfaces



Módulo 3 – Classes Abstratas e Interfaces

## SESSÃO 1 – EXEMPLO RAPOSAS E COELHOS

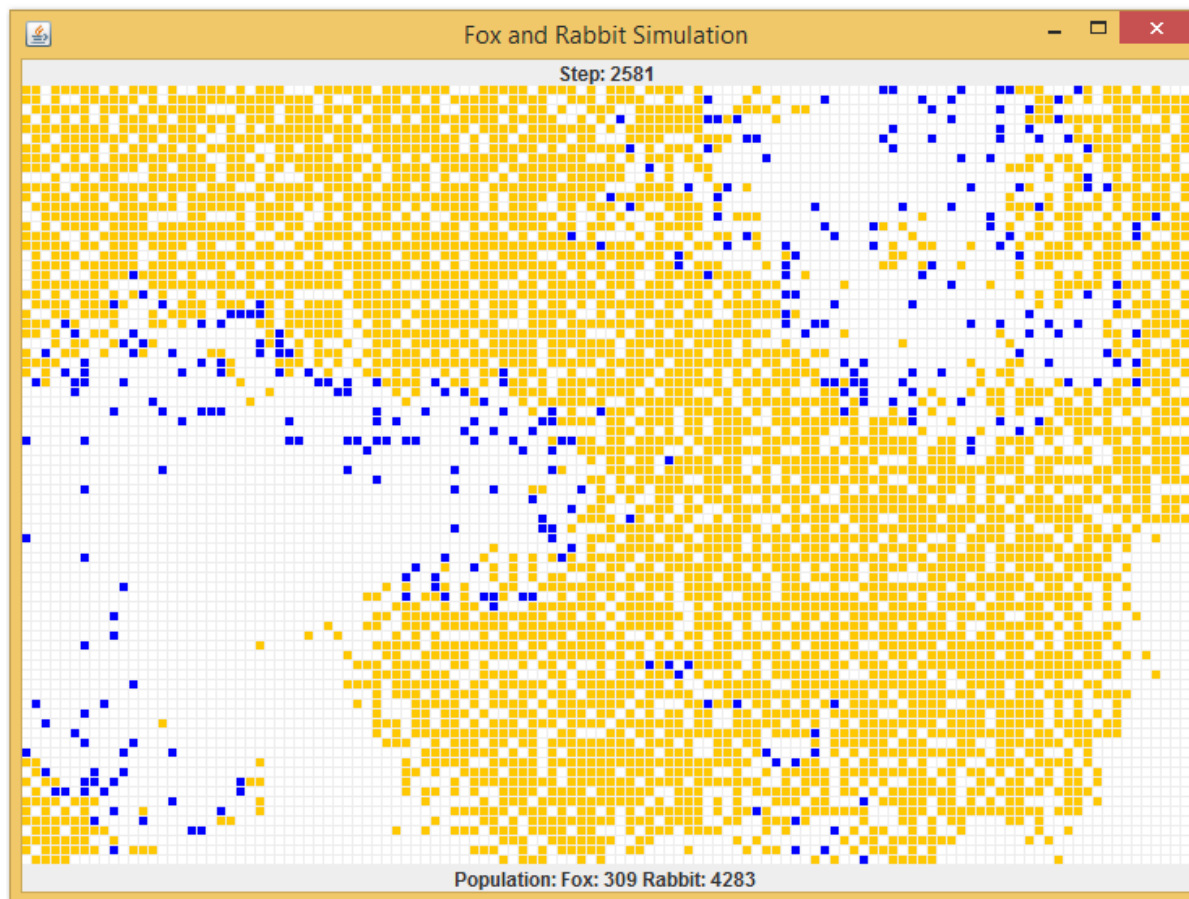
# Exemplo — Foxes and Rabbits

- Requisitos da aplicação:
  - Um simulador da evolução de populações de raposas e coelhos num campo limitado.
  - Na simulação os coelhos andam livremente pelo campo enquanto as raposas os caçam.
  - O equilíbrio das populações é obtido da seguinte forma:
    - Quando existem muitos coelhos não falta alimento às raposas e a sua população cresce rapidamente.
    - Com muitas raposas a caçar a população de coelhos começa a diminuir levando à diminuição do alimento das raposas e consequentemente à redução da sua população.
    - Com poucas raposas a população de coelhos começa a crescer e o ciclo repete-se



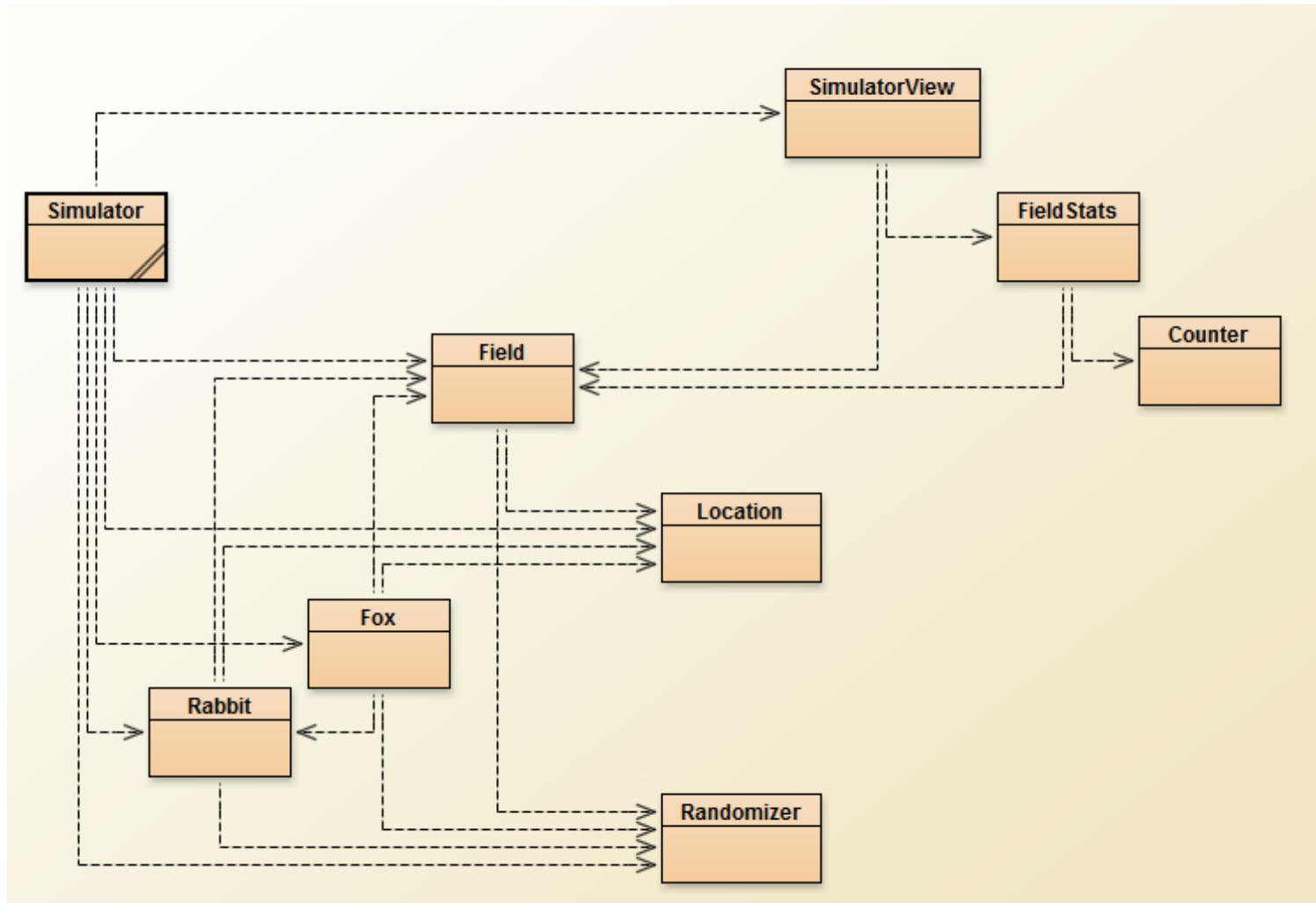
# Exemplo — Foxes and Rabbits

## □ Simulação:



# Exemplo — Foxes and Rabbits

- Diagrama de classes da aplicação **Foxes and Rabbits**:

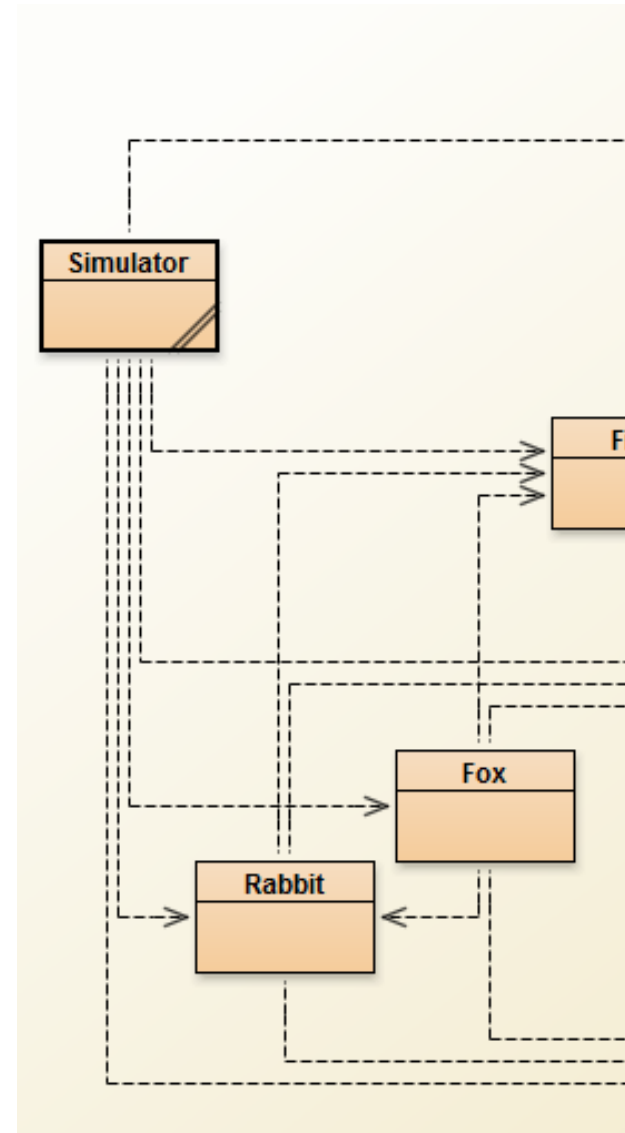


## Exemplo — Foxes and Rabbits

- ## □ Classes principais da aplicação

## Foxes and Rabbits:

- **Fox** – Representa as raposas.
- **Rabbit** – representa os coelhos
- **Simulator** – representa o motor da simulação.

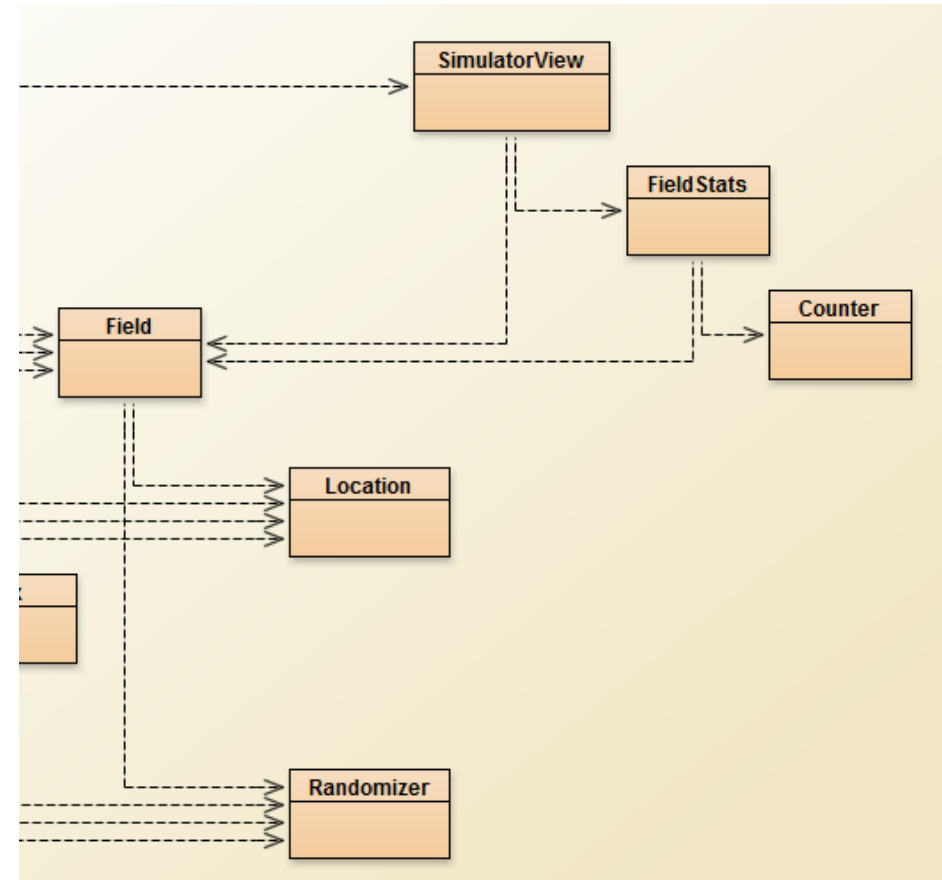


# Exemplo — Foxes and Rabbits

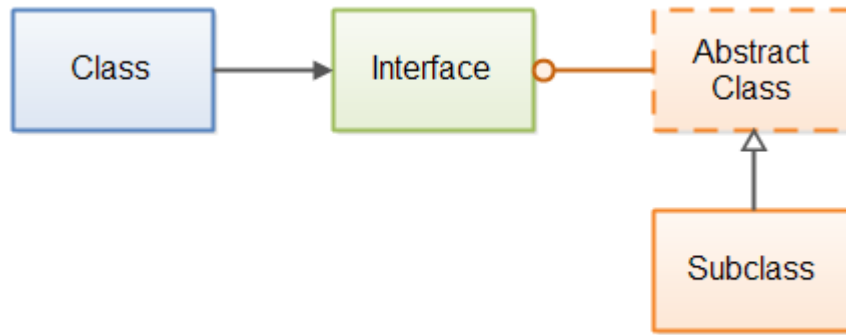
- Restantes classes da aplicação

## **Foxes and Rabbits:**

- **Field** – Representa o campo 2D onde se desenrola a simulação.
- **Location** – uma posição 2D dentro do campo.
- **Randomizer** – fornece a aleatoriedade da simulação
- **SimulatorView**, **FieldStats** e **Counter** – fornecem uma visualização gráfica da simulação





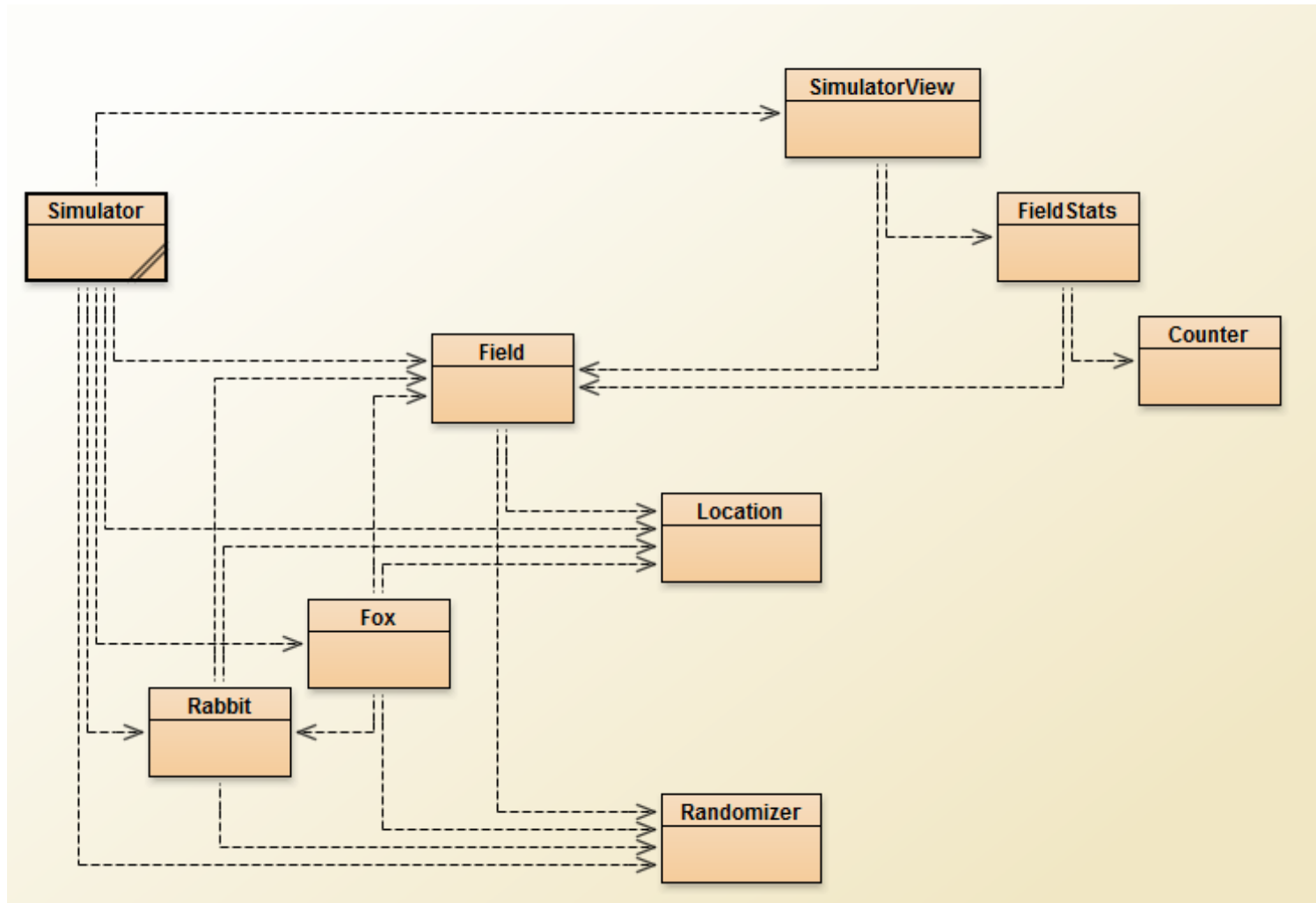


Módulo 3 – Classes Abstratas e Interfaces

## SESSÃO 2 – CÓDIGO DA APLICAÇÃO

# Exemplo — Foxes and Rabbits

- Diagrama de classes da aplicação **Foxes and Rabbits**:



# Exemplo — Foxes and Rabbits

## □ Classe **Rabbit**

```
public class Rabbit {
```

Idade mínima de reprodução

```
    private static final int BREEDING_AGE = 5;
```

```
    private static final int MAX_AGE = 40;
```

Idade máxima atingida

```
    private static final double BREEDING_PROBABILITY = 0.12;
```

```
    private static final int MAX_LITTER_SIZE = 4;
```

Número máximo de filhos

```
    private static final Random rand = Randomizer.getRandom();
```

```
    private int age;
```

```
    private boolean alive;
```

```
    private Location location;
```

```
    private Field field;
```

```
// continua...
```

# Exemplo — Foxes and Rabbits

## □ Classe **Rabbit** – **construtor** e método **run**

```
public Rabbit(boolean randomAge, Field field, Location location) {  
    age = 0;  
    alive = true;  
    this.field = field;  
    setLocation(location);  
    if(randomAge) {  
        age = rand.nextInt(MAX_AGE);  
    }  
}
```

**Ação principal dos coelhos: executada a cada passo da simulação para cada coelho**

```
public void run(List<Rabbit> newRabbits) {  
    incrementAge();  
    if(alive) {  
        giveBirth(newRabbits);  
        Location newLocation = field.freeAdjacentLocation(location);  
        if(newLocation != null) {  
            setLocation(newLocation);  
        }  
        else {  
            setDead();  
        }  
    }  
}
```

A lista **newRabbits** recebida é preenchida com os coelhos que nasceram

# Exemplo — Foxes and Rabbits

## □ Classe **Rabbit** – método **run**

```
public void run(List<Rabbit> newRabbits) {  
    incrementAge();  
    if(alive) {  
        giveBirth(newRabbits);  
        Location newLocation = field.freeAdjacentLocation(location);  
        if(newLocation != null) {  
            setLocation(newLocation);  
        }  
        else {  
            setDead();  
        }  
    }  
}
```

## □ ○ **comportamento dos coelhos** é definido pelo método **run**:

- A idade é incrementada a cada passo da simulação
  - Nesta altura o coelho pode morrer
- Os coelhos que já tiverem idade podem ter filhos em cada passo da simulação
  - Podem nascer coelhos nesta altura

# Exemplo — Foxes and Rabbits

- ❑ Classe **Rabbit** – métodos **isAlive**, **setDead**, **getLocation** e **setLocation**

```
public boolean isAlive() {
    return alive;
}

public void setDead() {
    alive = false;
    if(location != null) {
        field.clear(location);
        location = null;
        field = null;
    }
}

public Location getLocation() {
    return location;
}

private void setLocation(Location newLocation) {
    if(location != null) {
        field.clear(location);
    }
    location = newLocation;
    field.place(this, newLocation);
}
```

# Exemplo — Foxes and Rabbits

- Classe **Rabbit** – métodos **incrementAge** e **giveBirth**

```
private void incrementAge() {  
    age++;  
    if(age > MAX_AGE) {  
        setDead();  
    }  
}
```

```
private void giveBirth(List<Rabbit> newRabbits) {  
    List<Location> free = field.getFreeAdjacentLocations(location);  
    int births = breed();  
    for(int b = 0; b < births && free.size() > 0; b++) {  
        Location loc = free.remove(0);  
        Rabbit young = new Rabbit(false, field, loc);  
        newRabbits.add(young);  
    }  
}
```

# Exemplo — Foxes and Rabbits

- Classe **Rabbit** – métodos **breed**, e **canBreed**

```
private int breed() {  
    int births = 0;  
    if(canBreed() && rand.nextDouble() <= BREEDING_PROBABILITY) {  
        births = rand.nextInt(MAX_LITTER_SIZE) + 1;  
    }  
    return births;  
}  
  
private boolean canBreed() {  
    return age >= BREEDING_AGE;  
}
```



# Exemplo — Foxes and Rabbits

## □ Classe **Fox**

```
public class Fox {
```

Idade mínima de reprodução

```
    private static final int BREEDING_AGE = 15;
```

```
    private static final int MAX_AGE = 150;
```

Idade máxima atingida

```
    private static final double BREEDING_PROBABILITY = 0.08;
```

```
    private static final int MAX_LITTER_SIZE = 2;
```

```
    private static final int RABBIT_FOOD_VALUE = 9;
```

Número máximo de filhos

Número de passos necessários para raposa ter de comer novamente

```
    private static final Random rand = Randomizer.getRandom();
```

```
    private int age;
```

```
    private boolean alive;
```

```
    private Location location;
```

```
    private Field field;
```

```
    private int foodLevel;
```

```
// continua...
```

# Exemplo — Foxes and Rabbits

## □ Classe **Fox** – **construtor**

```
public Fox(boolean randomAge, Field field, Location location) {  
    age = 0;  
    alive = true;  
    this.field = field;  
    setLocation(location);  
    if(randomAge) {  
        age = rand.nextInt(MAX_AGE);  
        foodLevel = rand.nextInt(RABBIT_FOOD_VALUE);  
    }  
    else {  
        foodLevel = rand.nextInt(RABBIT_FOOD_VALUE);  
    }  
}
```

# Exemplo — Foxes and Rabbits

## □ Classe **Fox** — método **hunt**

```
public void hunt(List<Fox> newFoxes) {  
    incrementAge();  
    incrementHunger();  
    if(alive) {  
        giveBirth(newFoxes);  
        Location newLocation = findFood();  
        if(newLocation == null) {  
            newLocation = field.freeAdjacentLocation(location);  
        }  
        if(newLocation != null) {  
            setLocation(newLocation);  
        }  
        else {  
            setDead();  
        }  
    }  
}
```

A lista **newFoxes** recebida é preenchida com as raposas que nasceram

**Ação principal das raposas: executada a cada passo da simulação para cada raposa**

# Exemplo — Foxes and Rabbits

## □ Classe **FOX** — método **hunt**

```
public void hunt(List<Fox> newFoxes) {  
    incrementAge();  
    incrementHunger();  
    if(alive) {  
        giveBirth(newFoxes);  
        Location newLocation = findFood();  
        if(newLocation == null) {  
            newLocation = field.freeAdjacentLocation(location);  
        }  
        if(newLocation != null) {  
            setLocation(newLocation);  
        }  
        else {  
            setDead();  
        }  
    }  
}
```

## □ ○ **comportamento das raposas** é definido pelo método **hunt**:

- As raposas, como os coelhos, podem morrer ou reproduzirem-se
- As raposas ficam com fome
- As raposas procuram comida nas posições adjacentes

# Exemplo — Foxes and Rabbits

## □ Classe **Fox** – método **findFood**

```
private Location findFood() {
    List<Location> adjacent = field.adjacentLocations(location);
    Iterator<Location> it = adjacent.iterator();
    while(it.hasNext()) {
        Location where = it.next();
        Object animal = field.getObjectAt(where);
        if(animal instanceof Rabbit) {
            Rabbit rabbit = (Rabbit) animal;
            if(rabbit.isAlive()) {
                rabbit.setDead();
                foodLevel = RABBIT_FOOD_VALUE;
                return where;
            }
        }
    }
    return null;
}
```

# Exemplo — Foxes and Rabbits

- Classe **FOX** — métodos **isAlive**, **setDead**, **getLocation** e **setLocation**

```
public boolean isAlive() {
    return alive;
}

private void setDead() {
    alive = false;
    if(location != null) {
        field.clear(location);
        location = null;
        field = null;
    }
}

public Location getLocation() {
    return location;
}

private void setLocation(Location newLocation) {
    if(location != null) {
        field.clear(location);
    }
    location = newLocation;
    field.place(this, newLocation);
}
```

# Exemplo — Foxes and Rabbits

- Classe **FOX** — métodos **incrementAge**, **incrementHunger** e **giveBirth**

```
private void incrementAge() {  
    age++;  
    if(age > MAX_AGE) {  
        setDead();  
    }  
}
```

```
private void incrementHunger() {  
    foodLevel--;  
    if(foodLevel <= 0) {  
        setDead();  
    }  
}
```

```
private void giveBirth(List<Fox> newFoxes) {  
    List<Location> free = field.getFreeAdjacentLocations(location);  
    int births = breed();  
    for(int b = 0; b < births && free.size() > 0; b++) {  
        Location loc = free.remove(0);  
        Fox young = new Fox(false, field, loc);  
        newFoxes.add(young);  
    }  
}
```

# Exemplo — Foxes and Rabbits

- Classe **FOX** – métodos **breed**, e **canBreed**

```
private int breed() {  
    int births = 0;  
    if(canBreed() && rand.nextDouble() <= BREEDING_PROBABILITY) {  
        births = rand.nextInt(MAX_LITTER_SIZE) + 1;  
    }  
    return births;  
}  
  
private boolean canBreed() {  
    return age >= BREEDING_AGE;  
}
```



# Exemplo — Foxes and Rabbits

## □ Classe **Simulator**

```
public class Simulator {  
  
    private static final int DEFAULT_WIDTH = 120;  
    private static final int DEFAULT_DEPTH = 80;  
    private static final double FOX_CREATION_PROBABILITY = 0.02;  
    private static final double RABBIT_CREATION_PROBABILITY = 0.08;  
  
    private List<Rabbit> rabbits;  
    private List<Fox> foxes;  
  
    private Field field;  
    private int step;  
    private SimulatorView view;  
  
    // continua...
```

# Exemplo — Foxes and Rabbits

## ❑ Classe **Simulator** – **construtores**

```
public Simulator() {  
    this(DEFAULT_DEPTH, DEFAULT_WIDTH);  
}  
  
public Simulator(int depth, int width) {  
    if(width <= 0 || depth <= 0) {  
        System.out.println("The dimensions must be greater than zero.");  
        System.out.println("Using default values.");  
        depth = DEFAULT_DEPTH;  
        width = DEFAULT_WIDTH;  
    }  
  
    rabbits = new ArrayList<Rabbit>();  
    foxes = new ArrayList<Fox>();  
    field = new Field(depth, width);  
  
    view = new SimulatorView(depth, width);  
    view.setColor(Rabbit.class, Color.ORANGE);  
    view.setColor(Fox.class, Color.BLUE);  
  
    reset();  
}
```

**Utiliza `this()` para evitar a duplicação de código nos construtores**

# Exemplo — Foxes and Rabbits

- Classe **Simulator** – métodos **reset** e **populate**

```
public void reset() {
    step = 0;
    rabbits.clear();
    foxes.clear();
    populate();

    view.showStatus(step, field);
}

private void populate() {
    Random rand = Randomizer.getRandom();
    field.clear();
    for(int row = 0; row < field.getDepth(); row++) {
        for(int col = 0; col < field.getWidth(); col++) {
            if(rand.nextDouble() <= FOX_CREATION_PROBABILITY) {
                Location location = new Location(row, col);
                Fox fox = new Fox(true, field, location);
                foxes.add(fox);
            }
            else if(rand.nextDouble() <= RABBIT_CREATION_PROBABILITY) {
                Location location = new Location(row, col);
                Rabbit rabbit = new Rabbit(true, field, location);
                rabbits.add(rabbit);
            }
        }
    }
}
```

**Distribuição inicial das raposas e coelhos**

# Exemplo — Foxes and Rabbits

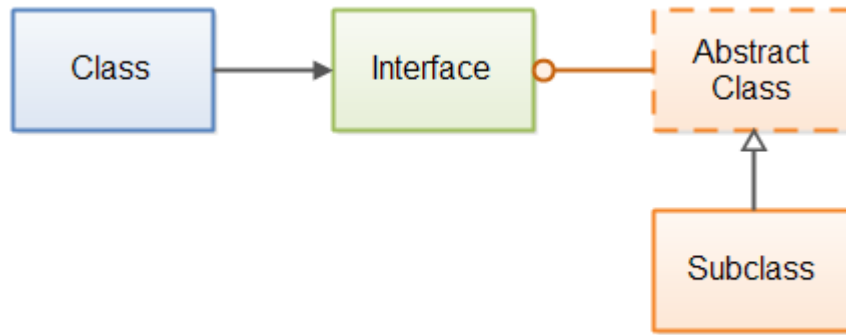
- Classe **Simulator** – método **simulateOneStep**

```
public void simulateOneStep() {
    step++;

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

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

**Passo da simulação:  
onde se executam as  
ações principais**



Módulo 3 – Classes Abstratas e Interfaces

## SESSÃO 3 – CLASSES E MÉTODOS ABSTRATOS

# Exemplo — Foxes and Rabbits

- **Análise e alterações à aplicação:**
  - Mais uma vez existem classes com vários atributos e métodos idênticos onde se pode aplicar a herança. É o caso das classes **Fox** e **Rabbit**
    - Criação da superclasse **Animal** com os atributos e métodos comuns que fizerem sentido.
  - Na simulação temos listas separadas para raposas e coelhos, com a herança e usando o principio da substituição podemos ter apenas uma lista de **Animal**
  - A cada passo da simulação as raposas caçam e os coelhos correm. Neste caso poderíamos tirar partido do polimorfismo onde cada um deles faz a sua ação.

# Exemplo — Foxes and Rabbits

## □ Análise e alterações à aplicação: criação da classe **Animal**

```
public class Fox {  
  
    private static final int BREEDING_AGE = 15;  
    private static final int MAX_AGE = 150;  
    private static final double BREEDING_PROBABILITY = 0.08;  
    private static final int MAX_LITTER_SIZE = 2;  
    private static final int RABBIT_FOOD_VALUE = 9;  
  
    private static final Random rand = Randomizer.getRandom();  
  
    private int age;  
    private boolean alive;  
    private Location location;  
    private Field field;  
    private int foodLevel;  
  
    // continua...
```



```
public class Rabbit {  
  
    private static final int BREEDING_AGE = 5;  
    private static final int MAX_AGE = 40;  
    private static final double BREEDING_PROBABILITY = 0.12;  
    private static final int MAX_LITTER_SIZE = 4;  
  
    private static final Random rand = Randomizer.getRandom();  
  
    private int age;  
    private boolean alive;  
    private Location location;  
    private Field field;  
  
    // continua...
```

- Os atributos **alive**, **location** e **field** podem passar para a classe **Animal**
  - A inicialização destes atributos será feita no construtor da classe **Animal**
- O atributo **age** não pode ainda passar porque depende da constante **MAX\_AGE** que é diferente nas classes.

# Exemplo — Foxes and Rabbits

## □ Análise e alterações à aplicação: criação da classe **Animal**

```
public class Fox {  
  
    private static final int BREEDING_AGE = 15;  
    private static final int MAX_AGE = 150;  
    private static final double BREEDING_PROBABILITY = 0.08;  
    private static final int MAX_LITTER_SIZE = 2;  
    private static final int RABBIT_FOOD_VALUE = 9;  
  
    private static final Random rand = Randomizer.getRandom();  
  
    private int age;  
    private boolean alive;  
    private Location location;  
    private Field field;  
    private int foodLevel;  
  
    public boolean isAlive() { ... }  
    private void setDead() { ... }  
    public Location getLocation() { ... }  
    private void setLocation(Location newLocation) { ... }  
  
    // continua...
```

```
public class Rabbit {  
  
    private static final int BREEDING_AGE = 5;  
    private static final int MAX_AGE = 40;  
    private static final double BREEDING_PROBABILITY = 0.12;  
    private static final int MAX_LITTER_SIZE = 4;  
  
    private static final Random rand = Randomizer.getRandom();  
  
    private int age;  
    private boolean alive;  
    private Location location;  
    private Field field;  
  
    public boolean isAlive() { ... }  
    public void setDead() { ... }  
    public Location getLocation() { ... }  
    private void setLocation(Location newLocation) { ... }  
  
    // continua...
```

- Os métodos seletores e modificadores associados aos atributos **alive**, **location** e **field** passam também para a classe **Animal**
- Vai ser necessário acrescentar um método **getField** na classe **Animal** porque esse atributo é acedido dentro de outros métodos das classes **Fox** e **Rabbit**



# Exemplo — Foxes and Rabbits

## □ Análise e alterações à aplicação: criação da classe **Animal**

```
public class Fox {  
  
    // código omitido  
  
    private int age;  
    private boolean alive;  
    private Location location;  
    private Field field;  
    private int foodLevel;  
  
    public boolean isAlive() { ... }  
    private void setDead() { ... }  
    public Location getLocation() { ... }  
    private void setLocation(Location newLocation) { ... }  
  
    // continua...
```

```
public class Rabbit {  
  
    // código omitido  
  
    private int age;  
    private boolean alive;  
    private Location location;  
    private Field field;  
  
    public boolean isAlive() { ... }  
    public void setDead() { ... }  
    public Location getLocation() { ... }  
    private void setLocation(Location newLocation) { ... }  
  
    // continua...
```

- A visibilidade do método **setLocation** é **private**, se se quiser ter acesso a este método nas subclasses tem de se alterar para **protected** pelo menos na classe **Animal**.
- O método **setDead** é **public** na classe **Rabbit** e **private** na classe **Fox** porque na classe **Fox** era necessário aceder a esse método da classe **Rabbit**. Neste caso pode-se passar também a **protected** tendo em conta que não se pretende que o método faça parte da interface da classe

# Exemplo — Foxes and Rabbits

- Análise e alterações à aplicação: classe **Simulator**

```
public class Simulator {  
  
    private static final int DEFAULT_WIDTH = 120;  
    private static final int DEFAULT_DEPTH = 80;  
    private static final double FOX_CREATION_PROBABILITY = 0.02;  
    private static final double RABBIT_CREATION_PROBABILITY = 0.08;  
  
    private List<Rabbit> rabbits;  
    private List<Fox> foxes;  
  
    private Field field;  
    private int step;  
    private SimulatorView view;  
  
    // continua...
```



Listas separadas para  
Fox e Rabbit

- Com a utilização da herança podemos ter apenas um tipo de lista: uma lista de **Animal**
  - `private List<Animal> animals;`

# Exemplo — Foxes and Rabbits

## □ Análise e alterações à aplicação: classe **Simulator**

```
public class Simulator {  
    public void simulateOneStep() {  
        // código omitido  
        for(Iterator<Animal> it = animals.iterator(); it.hasNext(); ) {  
            Animal animal = it.next();  
            if(animal instanceof Rabbit) {  
                Rabbit rabbit = (Rabbit) animal;  
                rabbit.run(newAnimals);  
            }  
            else if(animal instanceof Fox) {  
                Fox fox = (Fox) animal;  
                fox.hunt(newFoxes);  
            }  
            else {  
                System.out.println("found unknown animal");  
            }  
            if(!animal.isAlive()) {  
                it.remove();  
            }  
        }  
    }  
}
```



É necessário identificar o tipo de animal com **instanceOf**

□ Não conseguimos utilizar o polimorfismo porque as ações da raposa (**hunt**) e do coelho (**run**) são diferentes.

- **Solução:** criar o método **act** na classe **Animal** que na classe **Fox** chama o método **hunt** e na classe **Rabbit** chama o método **run**.

# Exemplo — Foxes and Rabbits

## □ Análise e alterações à aplicação: classe **Simulator**

```
public class Simulator {  
    public void simulateOneStep() {  
        // código omitido  
  
        for(Iterator<Animal> it = animals.iterator(); it.hasNext(); ) {  
            Animal animal = it.next();  
            animal.act(newAnimals);  
            if(! animal.isAlive()) {  
                it.remove();  
            }  
        }  
    }  
}
```

- **Solução:** criar o método **act** na classe **Animal** que na classe **Fox** chama o método **hunt** e na classe **Rabbit** chama o método **run**.
- Neste caso volta a acontecer o problema: “que código colocar no método **act** da classe **Animal**”?
  - Na realidade a classe **Animal** é abstrata, não existem “animais” em abstrato mas sim instâncias particulares de animais como é o caso da raposa ou do coelho.

# Exemplo — Foxes and Rabbits

- Análise e alterações à aplicação: classe **Simulator**

```
public class Simulator {  
    public void simulateOneStep() {  
        // código omitido  
  
        for(Iterator<Animal> it = animals.iterator(); it.hasNext(); ) {  
            Animal animal = it.next();  
            animal.act(newAnimals);  
            if(! animal.isAlive()) {  
                it.remove();  
            }  
        }  
    }  
}
```

- Quando se tem um método para o qual não se pretende definir o código dizemos que é um **método abstrato**
  - Em Java é possível criar um método abstrato

# Métodos abstratos

## □ Métodos abstratos em Java

modificador **abstract**

sem código e a terminar em ;



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

- Os **métodos abstratos** têm o modificador **abstract** e em vez do código dentro de chavetas têm em sua substituição um ponto e vírgula
  - Nestes casos o método abstrato pode ser usado com o polimorfismo

# Classes abstratas

## □ Classes abstratas em Java

modificador **abstract**



```
public abstract class Animal {
```

- Tal como os métodos as classes também podem ser abstratas.
- Para uma classe se tornar abstrata tem que se colocar o modificador **abstract** antes da palavra **class**.
- As classes abstratas não têm objetos
  - Neste caso não será possível criar instâncias duma classe abstrata.

# Classes e métodos abstratos

- Regras das classes e métodos abstratos:
  - Um **método abstrato** não tem código
  - Uma **classe abstrata** não tem instâncias
  - Se uma classe tiver pelo menos um método abstrato ela terá de ser obrigatoriamente abstrata
    - Caso contrário existirá um erro de compilação
- As **classes abstratas** são usadas para:
  - Definir uma classe base da qual não se pretende criar objetos
  - Definir uma classe incompleta que inclui um conjunto de métodos (concretos) que servem de base à implementação de classes semelhantes.
    - Neste caso criam-se métodos abstratos que devem ser definidos nas classes derivadas e assim concluir a construção da classe desse tipo
- Os **métodos abstratos** são usados para:
  - Definir comportamentos que se pretende que sejam implementados numa hierarquia de classes



# Classes e métodos abstratos

- Os **métodos abstratos** criados numa classe são herdados pelas classes derivadas.
  - Neste caso devem ser redefinidos nas classes derivadas
  - Se não forem definidos numa classe derivada essa classe passa a ter um método abstrato e consequentemente deve ser obrigatoriamente abstrata.

# Exemplo — Foxes and Rabbits

- Análise e alterações à aplicação: novo método **act**

```
public abstract class Animal {  
    // código omitido  
  
    public abstract void act(List<Animal> newAnimals);  
}
```

classe **Animal**  
abstrata

método **act** abstrato

```
public class Simulator {  
    public void simulateOneStep() {  
        // código omitido
```

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

Polimorfismo do método  
**act** a funcionar

```
public class Fox extends Animal {  
    // código omitido  
  
    @Override  
    public void act(List<Animal> newFoxes) {  
        // código do método hunt  
    }  
}
```

métodos **act** redefinidos  
nas subclasses

```
public class Rabbit extends Animal {  
    // código omitido  
  
    @Override  
    public void act(List<Animal> newRabbits) {  
        // código do método run  
    }  
}
```

# Exemplo — Foxes and Rabbits

- Análise e alterações à aplicação: criação da classe **Animal**

```
public class Fox {  
  
    // código omitido  
    private static final int BREEDING_AGE = 15;  
  
    private int age;  
  
    private boolean canBreed() {  
        return age >= BREEDING_AGE;  
    }  
  
    // continua...
```

```
public class Rabbit {  
  
    // código omitido  
    private static final int BREEDING_AGE = 5;  
  
    private int age;  
  
    private boolean canBreed() {  
        return age >= BREEDING_AGE;  
    }  
  
    // continua...
```

- Alguns métodos não foram colocados na classe **Animal** porque dependiam de constantes diferentes nas classes derivadas, é o caso, por exemplo do método **canBreed**
  - Neste caso as constantes (e os atributos), ao contrário dos métodos não podem ser redefinidos nas classes derivadas pelo que não faz sentido serem colocadas na classe base.
- Com os **métodos abstratos** e o **polimorfismo** pode-se alterar a situação...

# Exemplo — Foxes and Rabbits

- Análise e alterações à aplicação: método **canBreed**

método **getBreedingAge** abstrato

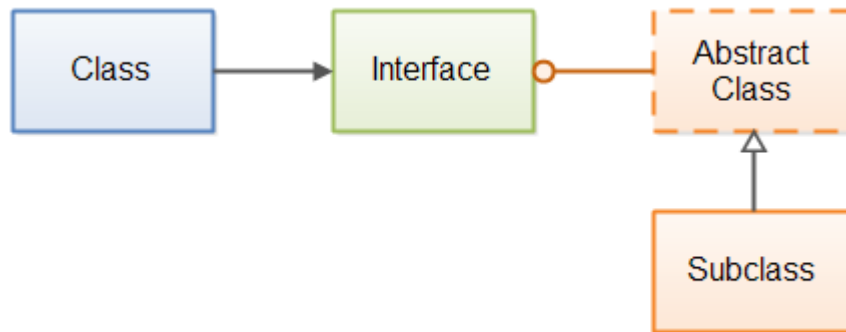
```
public abstract class Animal {  
    // código omitido  
    private int age;  
  
    private boolean canBreed() {  
        return age >= getBreedingAge();  
    }  
  
    protected abstract int getBreedingAge();  
}
```

Polimorfismo do método  
**getBreedingAge** a funcionar

```
public class Fox extends Animal {  
    // código omitido  
    private static final int BREEDING_AGE = 15;  
  
    @Override  
    protected int getBreedingAge() {  
        return BREEDING_AGE;  
    }  
}
```

métodos **getBreedingAge**  
redefinidos nas subclasses

```
public class Rabbit extends Animal {  
    private static final int BREEDING_AGE = 5;  
  
    @Override  
    protected int getBreedingAge() {  
        return BREEDING_AGE;  
    }  
}
```



Módulo 3 – Classes Abstratas e Interfaces

## SESSÃO 4 – INTERFACES

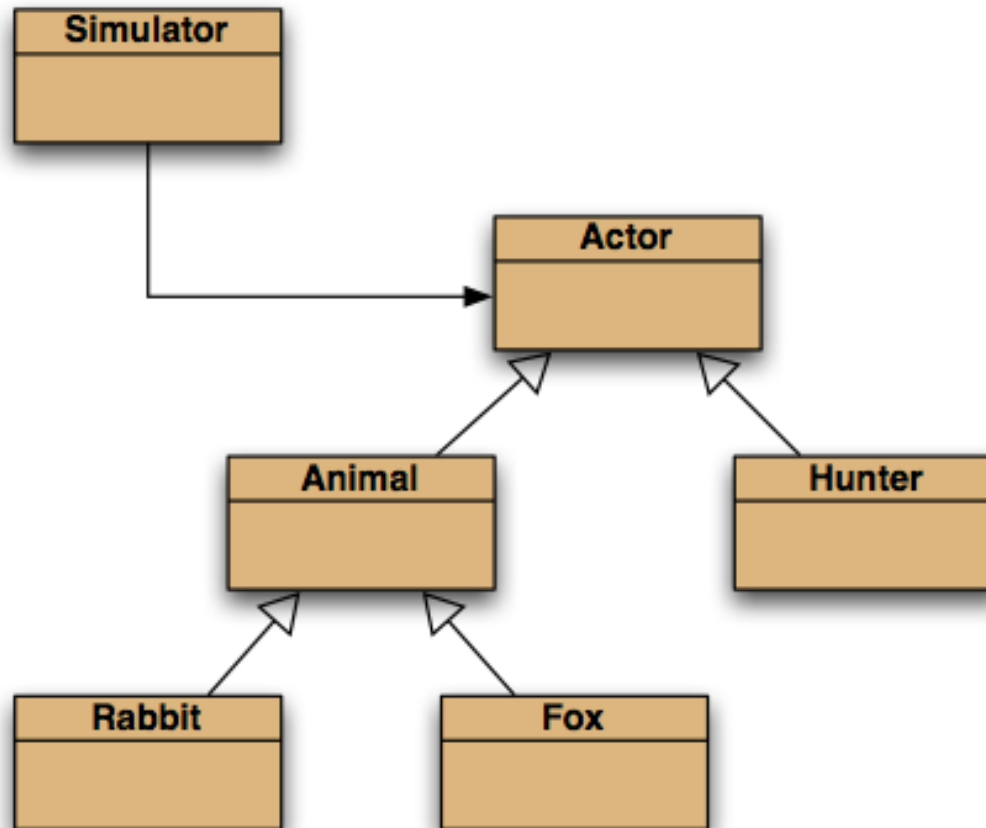
# Exemplo — Foxes and Rabbits

## □ Novas funcionalidades da simulação:

- Uma nova funcionalidade poderá ser a adição de mais animais.
  - Simples de implementar criando uma nova subclasses da classe **Animal**.
- Outra funcionalidade seria a inclusão de **predadores humanos**.
  - Podiam ser, por exemplo, caçadores ou apenas colocarem armadilhas
  - Neste caso os atores da simulação não seriam apenas animais.
- Outros atores a incluir podiam ser **plantas** ou mesmo as **condições meteorológicas**.
  - As plantas influenciariam a população de coelhos e o crescimento das plantas seria influenciado pelas condições atmosféricas
- Nos casos anteriores temos novos atores na simulação mais gerais. Uma solução para poder continuar a tirar partido do polimorfismo do método **act** seria a criação de uma classe **Actor** que servisse de base às classes referidas e que incluísse o método referido como abstrato.

# Exemplo — Foxes and Rabbits

- Análise e alterações à aplicação: novas classes **Actor** e **Hunter**



# Exemplo — Foxes and Rabbits

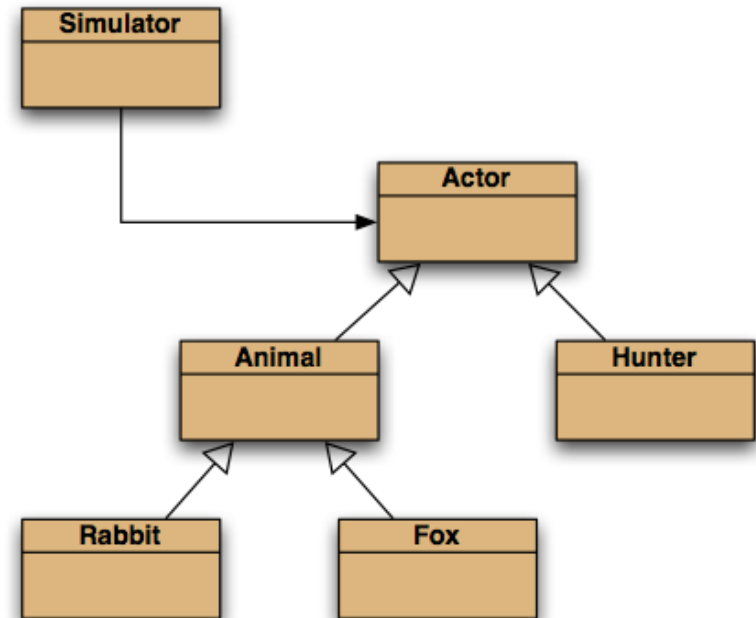
- ❑ Análise e alterações à aplicação: nova classe **Actor** e **Hunter**

- ❑ Neste caso a classe **Actor** poderia ser simplesmente:

```
public abstract class Actor {  
    public abstract void act(List<Actor> newActors);  
    public abstract boolean isActive();  
}
```

- ❑ Esta estrutura permitiria adicionar facilmente outro tipo de atores

Mas uma classe que apenas tem métodos abstratos pode ser definida usando um **novo tipo de dados** em Java: as **interfaces**






# Interfaces

## □ Interfaces em Java

Palavra reservada **interface**

Lista de métodos



```
public interface Actor {  
    void act(List<Actor> newActors);  
    boolean isActive();  
}
```

- As interfaces são definidas utilizando-se a palavra chave **interface**
- As interfaces incluem um conjunto de métodos.
- Todos os métodos das interfaces são abstratos e públicos
  - Não é **necessário** neste caso utilizar qualquer dos modificadores **abstract** e **public**
- É ainda possível incluir constantes públicas
  - Também neste caso podem-se omitir as palavras **public**, **static** e **final**

# Interfaces

## □ Interfaces em Java

```
public interface Actor {  
    void act(List<Actor> newActors);  
    boolean isActive();  
}
```

Palavra reservada **implements**

```
public class Fox extends Animal implements Actor {  
    //...
```

- As classes podem *herdar* das interfaces da mesma forma que herdam duma classe.
  - Neste caso usam a palavra **implements** em vez de **extends**
  - Diz-se que uma classe *implementa* uma interface porque, neste caso, ela terá de implementar todos os seus métodos. Caso contrário passará a ser uma classe abstrata.

# Interfaces

## □ Interfaces em Java

```
public interface Actor {  
    void act(List<Actor> newActors);  
    boolean isActive();  
}
```

Lista de interfaces



```
public class Fox extends Animal implements Actor, Drawable {  
    //...
```

## □ Uma classe pode implementar (*herdar de*) **várias interfaces**.

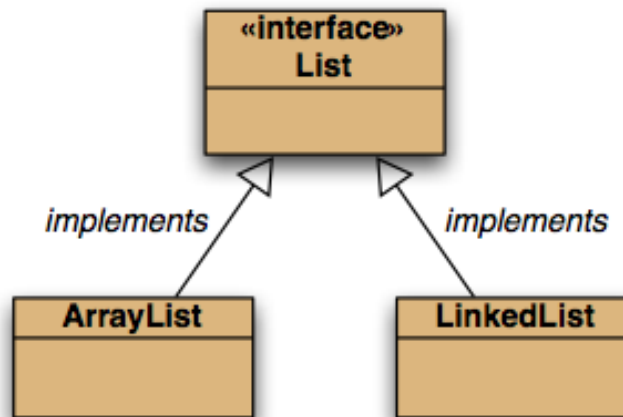
- Neste caso colocam-se as várias interfaces a seguir à palavra **implements** separadas por vírgulas
- As classes que implementam várias interfaces na prática terão de implementar todos os métodos que existem nessas interfaces.

# Interfaces

- As **interfaces** definem um novo tipo de dados
  - É possível criar variáveis dum tipo interface:  
Exemplo: **Actor actor =**
  - Apesar de ser possível criar variáveis do tipo interface não é possível criar valores do tipo interface.
    - Neste caso uma variável deste tipo apenas pode guardar um objeto duma classe que implementa essa interface. Uma classe que implementa uma interface é um subtipo dessa interface.  
Exemplo: **Actor actor = new Fox();**
  - O polimorfismo está disponível para interfaces da mesma forma que está para classes

# Interfaces

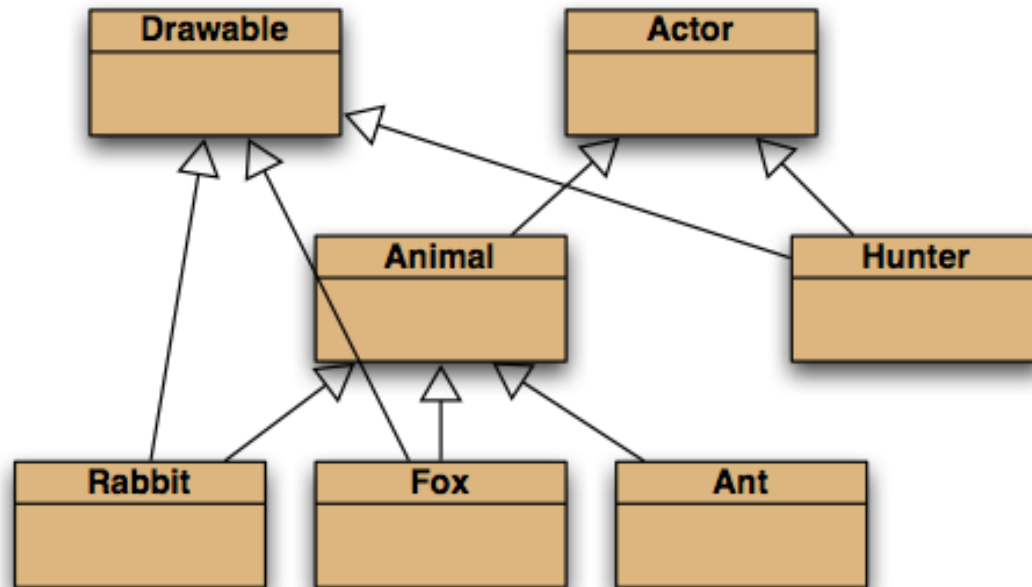
- As **interfaces** são especificações de comportamentos
  - Separam as funcionalidades (os seus métodos abstratos) da sua implementação (nas classes que as implementam)
  - As classes que implementa essas especificações podem escolher a forma de o fazer.



- As **interfaces** podem herdar de outras interfaces
  - Neste caso é como se adicionassem métodos aos métodos que são *herdados*

# Interfaces

## ❑ Herança múltipla de interfaces



# Classes abstratas e Interfaces

## □ Classes abstratas

- Definem uma entidade abstrata
- Representam entidades abstratas das quais nunca serão instanciados objetos.
- Podem ter métodos abstratos e métodos concretos
- Os métodos abstratos definem um comportamento que deve ser implementado na hierarquia de classes

## □ Interfaces

- Definem um comportamento
- Representam conjuntos de funcionalidades sem implementação.
- Apenas têm métodos abstratos
- Os métodos das interfaces definem um comportamento que pode ser implementado por quaisquer classes (pertencendo ou não a uma hierarquia)

# Bibliografia

- Objects First with Java (6th Edition), David Barnes & Michael Kölling, Pearson Education Limited, 2016
  - Capítulo 12

