Další techniky abstrakce

Abstraktní třídy a interfejsy

Založeno na originální prezentaci ke Kapitole 10
„Further abstraction techniques“ z učebnice
Objects First with Java - A Practical Introduction using BlueJ, © David J. Barnes, Michael Kölling

Hlavní pojmy

- Abstraktní třídy
- Interfejsy
- Vícenásobná dědičnost

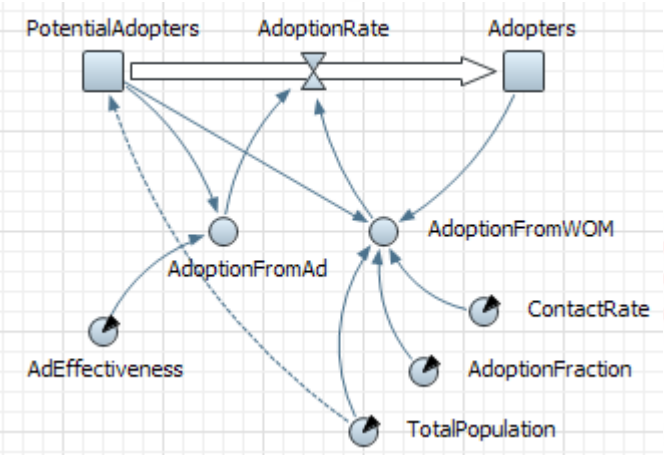
Simulace

- Programy, které se používají zpravidla pro simulaci skutečných činností (dějů).
  - městský provoz
  - počasí
  - jaderné procesy
  - pohyby na trhu cenných papírů
  - změny životního prostředí

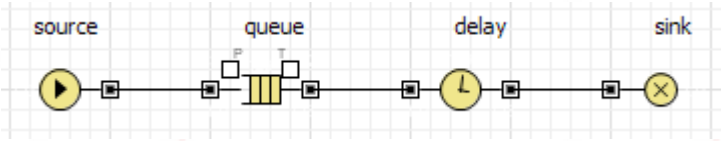
Simulace

- Často se jedná pouze o částečné simulace.
- Často obsahují zjednodušení.
  - Více detailů má potenciál poskytnout větší přesnost.
  - Více detailů vyžaduje typicky více zdrojů.
    - výpočetní výkon
    - doba simulace

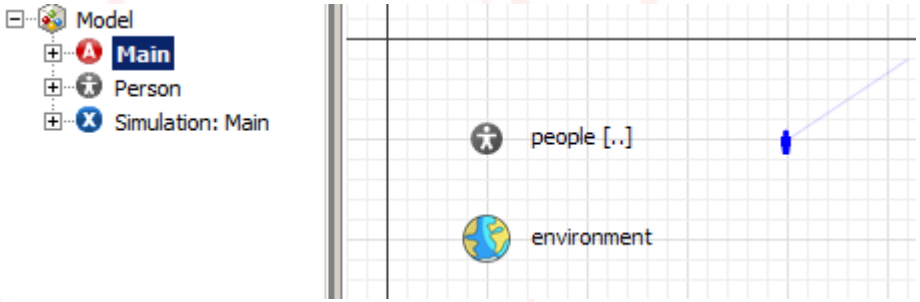
Přístupy k simulaci

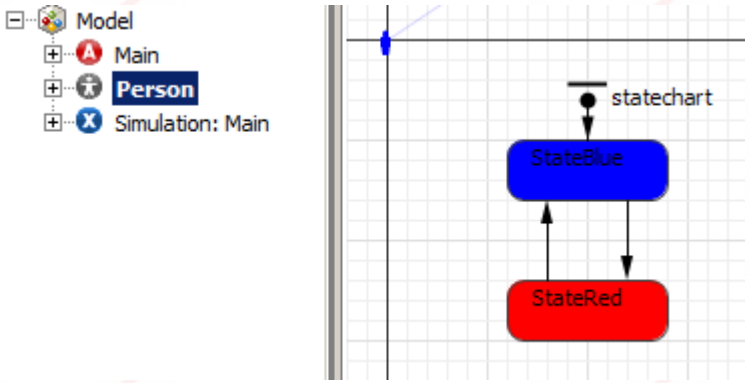


SDM



DEM





ABM

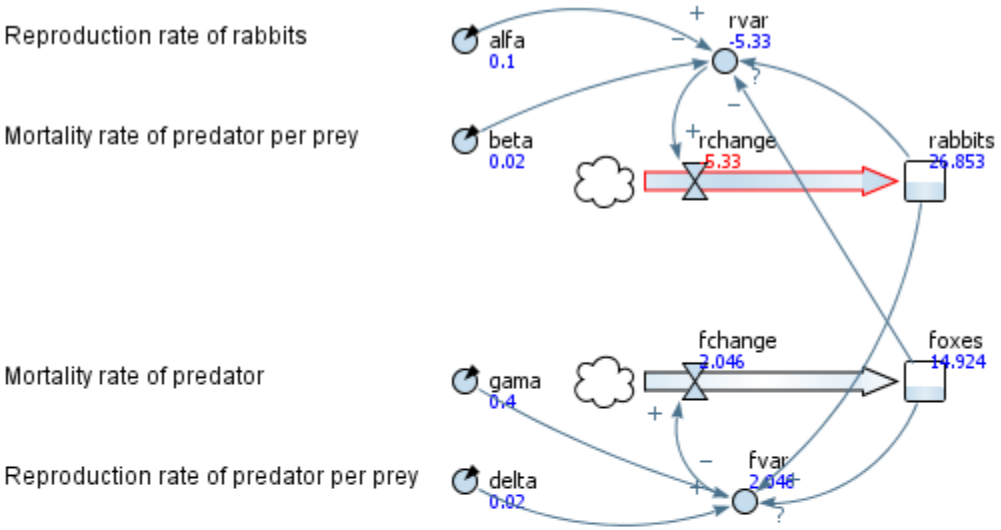
Výhody simulací

- Podporují tvorbu předpovědí.
  - např. počasí.
- Umožňují experimentování.
  - bezpečnější, levnější, rychlejší.
- Příklad:
  - „Jak bude život v přírodě ovlivněn tím, že dálnice povede středem národního parku?“

Simulace dravec - kořist

- Mezi druhy existuje často křehká rovnováha.
  - Mnoho kořisti znamená mnoho potravy pro dravce.
  - Mnoho potravy podporuje zvýšení počtu dravců.
  - Více dravců žere více kořisti.
  - Méně kořisti znamená méně potravy.
  - Méně potravy znamená...
- Lotka –Volterra equation
  - http://en.wikipedia.org/wiki/Lotka–Volterra_equation

Ukázka řešení

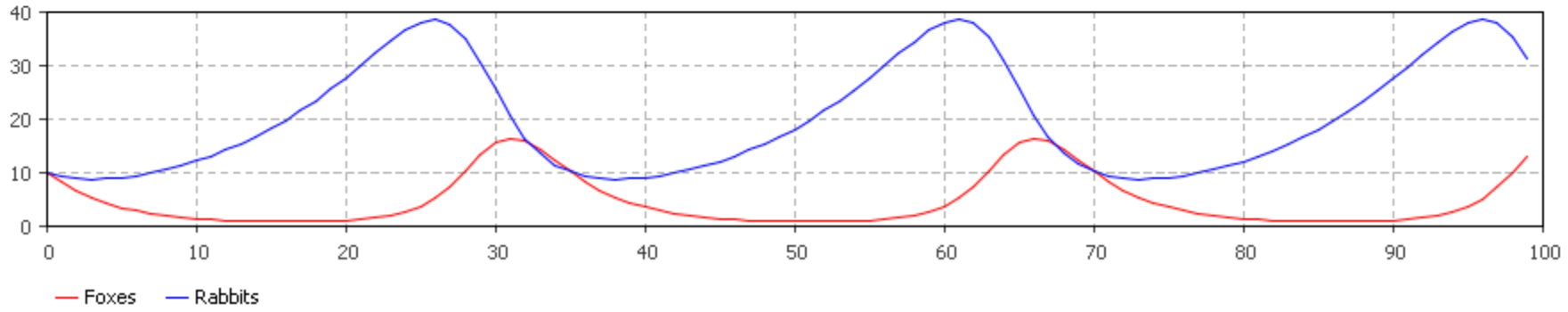


$$\frac{dx}{dt} = \alpha x - \beta xy$$

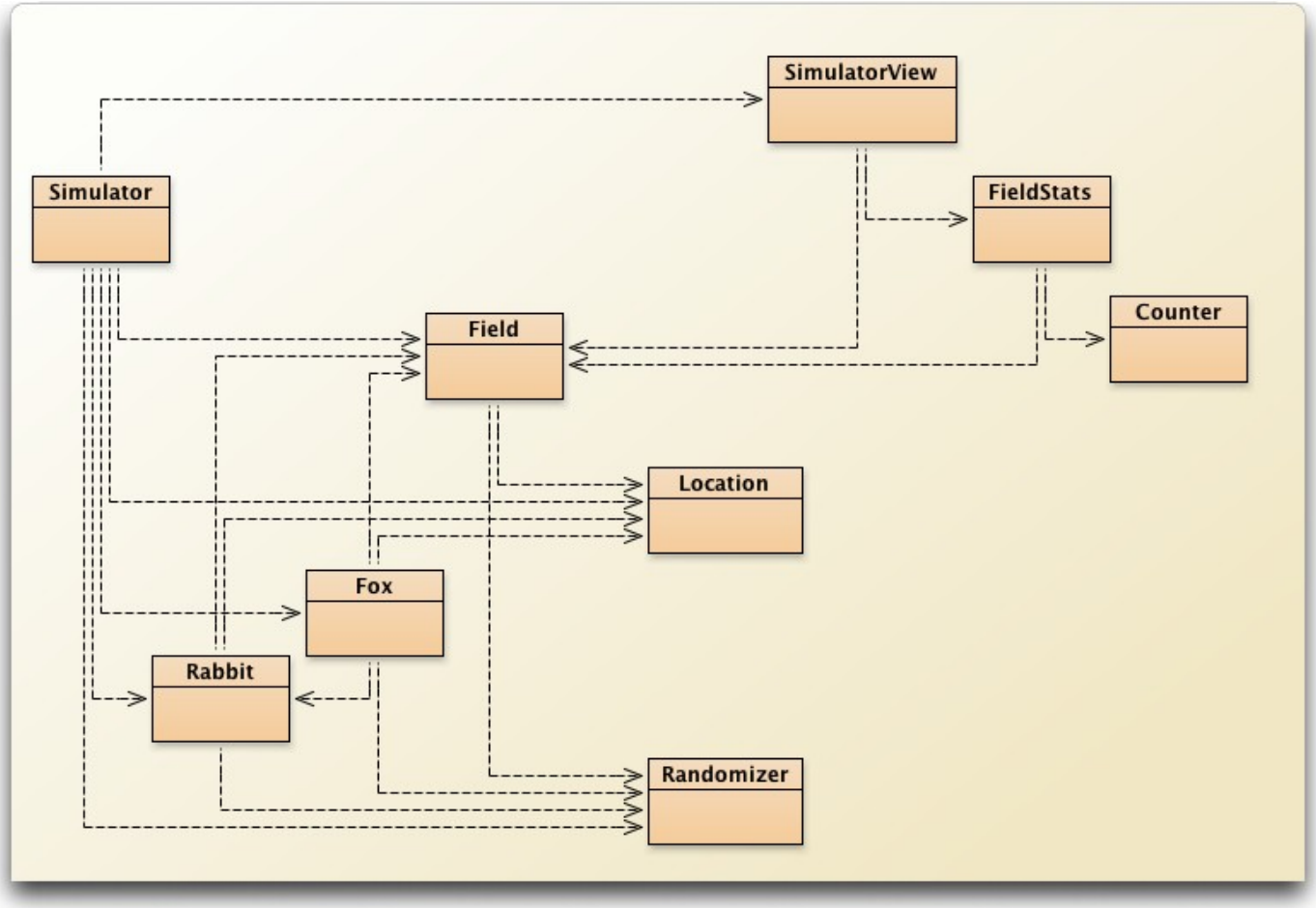
$$\frac{dy}{dt} = \delta xy - \gamma y$$

where

- x is the number of prey (for example, rabbits);
- y is the number of some predator (for example, foxes);
- $\frac{dy}{dt}$ and $\frac{dx}{dt}$ represent the growth rates of the two populations over time;
- t represents time; and
- $\alpha, \beta, \gamma, \delta$ are positive real parameters describing the interaction of the two species.



Projekt foxes and rabbits



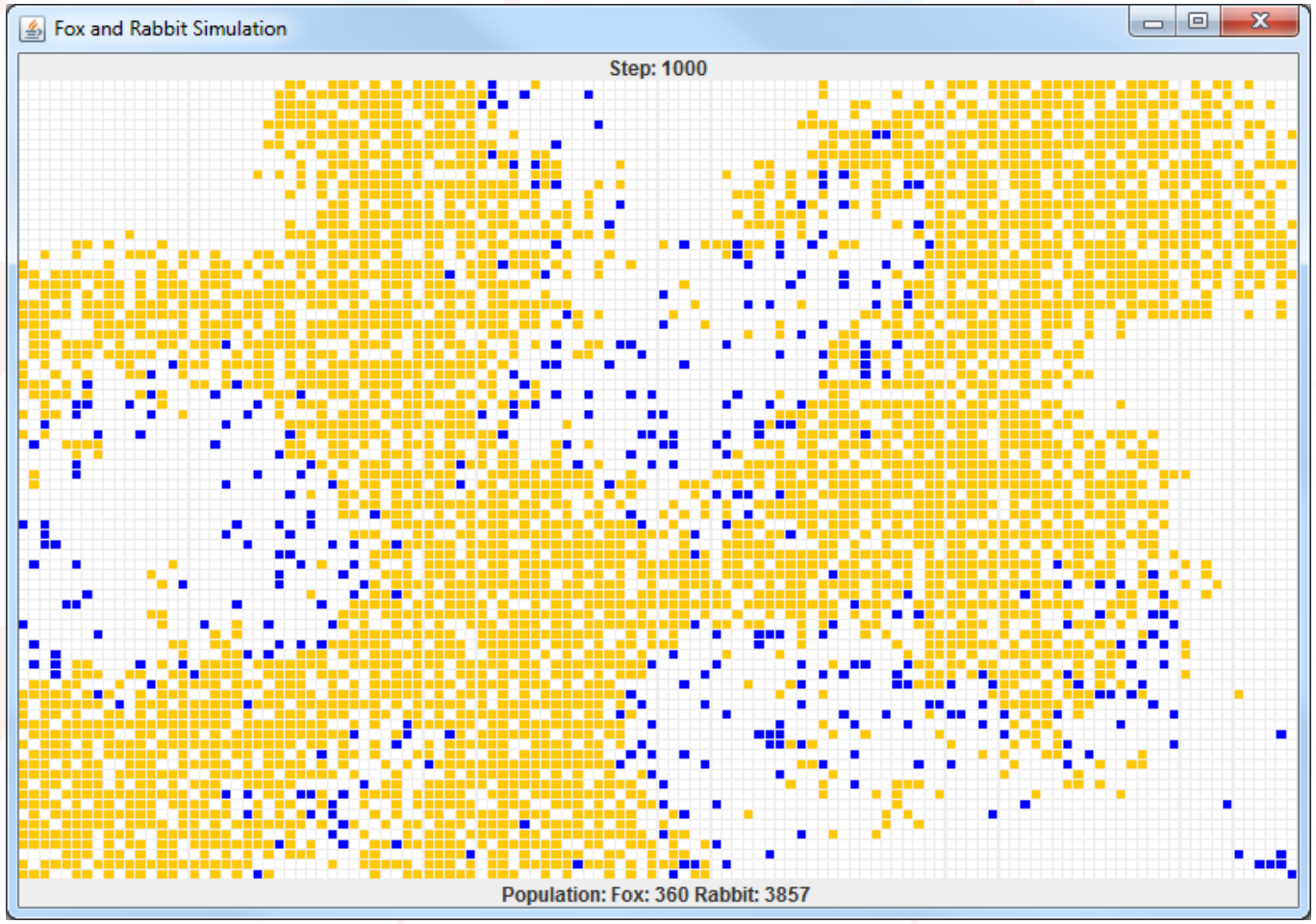
Hlavní třídy

- **`Fox`**
  - Jednoduchý model typu dravce.
- **`Rabbit`**
  - Jednoduchý model typu kořisti.
- **`Simulator`**
  - Řídí celkovou simulační úlohu.
  - Uchovává kolekci lišek a králíků.

Zbývající třídy

- **`Field`**
  - Reprezentuje 2D pole.
- **`Location`**
  - Reprezentuje 2D pozici v poli **`Field`**.
- **`SimulatorView`**, **`FieldStats`**, **`Counter`**
  - Udržují statistiku a poskytují pohled na pole **`Field`**.
- **`Randomizer`**
  - Řídí náhodnost prováděné simulace.

Příklad vizualizace



Stav králíka

```
public class Rabbit
{
    statické proměnné jsou vynechané

    // individuální vlastnosti(instanční proměnné).

    // věk králíka
    private int age;
    // zda je králík naživu či nikoliv
    private boolean alive;
    // pozice králíka
    private Location location;
    // obývané pole
    private Field field;

    metody jsou vynechány.
}
```

Chování králíka

- Řízeno metodou **`run`**. Králík se pohybuje v okolí aktuální pozice.
- Věk je navyšován při každém „kroku“ simulace.
  - Králík může umřít v tomto okamžiku.
- Králíci, kteří jsou dostatečně staří, mohou mít mladé v každém kroku.
  - Noví králíci se mohou narodit v tomto okamžiku.

Zjednodušení v modelu králíka

- U králíků se nerozlišuje pohlaví.
  - Ve skutečnosti jsou všichni samičky.
- Stejný králík může rodit v každém simulačním kroku.
- Všichni králíci umírají ve stejném věku.
- Ostatní?
  - Např. se nepočítá se stravou pro králíky.
  - …

Stav lišky

```
public class Fox
{
    statické proměnné jsou vynechané

 // individuální vlastnosti(instanční proměnné).
    // věk lišky
    private int age;
    // zda je liška naživu či nikoliv
    private boolean alive;
    // pozice lišky
    private Location location;
    // obývané pole
    private Field field;
    // úroveň nasycení lišky, která se zvyšuje pojídáním
    // králíků. Max. počet kroků simulace, než musí zase žrát.
    private int foodLevel;

    metody jsou vynechány.
}
```

Chování lišky

- Řízeno metodou **`hunt`.**
- Lišky rovněž stárnou a rodí mláďata.
- V každém kroku simulace se zvyšuje úroveň hladu, pokud se jim nezdaří ulovit králíka.
- Loví potravu v nejbližším okolí aktuální pozice (na sousedních pozicích).

Konfigurace lišek

- Podobná zjednodušení jako u králíků.
- Lov a stravování by mohlo být modelováno mnoha jinými způsoby.
  - Měla by se úroveň nasycení měnit aditivně?
  - Je více či méně pravděpodobné, že hladová liška loví?
- Jsou zjednodušení vůbec přijatelná?

Třída Simulator

- Tři klíčové části:
  - Nastavení v konstruktoru.
  - Metoda **`populate`.**
    - Každé zvíře má při zahájení simulace náhodný věk.
  - Metoda **`simulateOneStep`**.
    - Iteruje přes oddělené populace lišek a králíků.
    - Pořadí kroků?

Iterator<E>

- Iterator<E> je generický interfejs, který umožňuje popsat postupné zpracování objektů typu E.
- Typické použití pro iterativní zpracování kolekcí objektů. Předpokládejme, že **`collection`** je kolekce objektů typu **`Rabbit`**.

```
Iterator<Rabbit> it = collection.iterator();
while(it.hasNext()){
        Rabbit rabbit = it.next();
        zpracování prvku rabbit
}
```

Aktualizační krok

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

Prostor pro zlepšení

- Třídy **`Fox`** a **`Rabbit`** jsou si velmi podobné, ale nemají společnou nadtřídu.
- Aktualizační krok se týká podobně vypadajícího kódu.
- Třída **`Simulator`** je těsně svázána se specifickými třídami aktérů simulace.
  - „Zná“ mnoho o chování lišek a králíků.

Nadtřída Animal

- Umístíme společné položky do třídy **`Animal`**:
  - **`BREEDING_AGE`**, **`alive`**, **`location`**, **`field`**.
  - Pozor na mechanický přístup k návrhu společné nadtřídy.
- Přejmenování metod podporuje ukrývání informací:
  - z **`run`** a **`hunt`** se stane **`act`**.
- Třída **`Simulator`** může být nyní výrazně oddělena od konkrétních tříd aktérů simulace.

Upravená iterace (zrušení vazby)

```
for(Iterator<Animal> it = animals.iterator();
  it.hasNext(); ) {
        Animal animal = iter.next();
        animal.act(newAnimals);
        // Odstraň mrtvá zvířata ze simulace
        if(!animal.isAlive()) {
            it.remove();
        }
  }
```

Metoda `act` ve třídě `Animal`

- Kontrola na základě statických typů vyžaduje, aby metoda **`act`** byla ve třídě **`Animal`**.
- Neexistuje ovšem žádná zřejmá sdílená implementace.
- Deklarujeme **`act`** jako abstraktní:

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

nebo

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

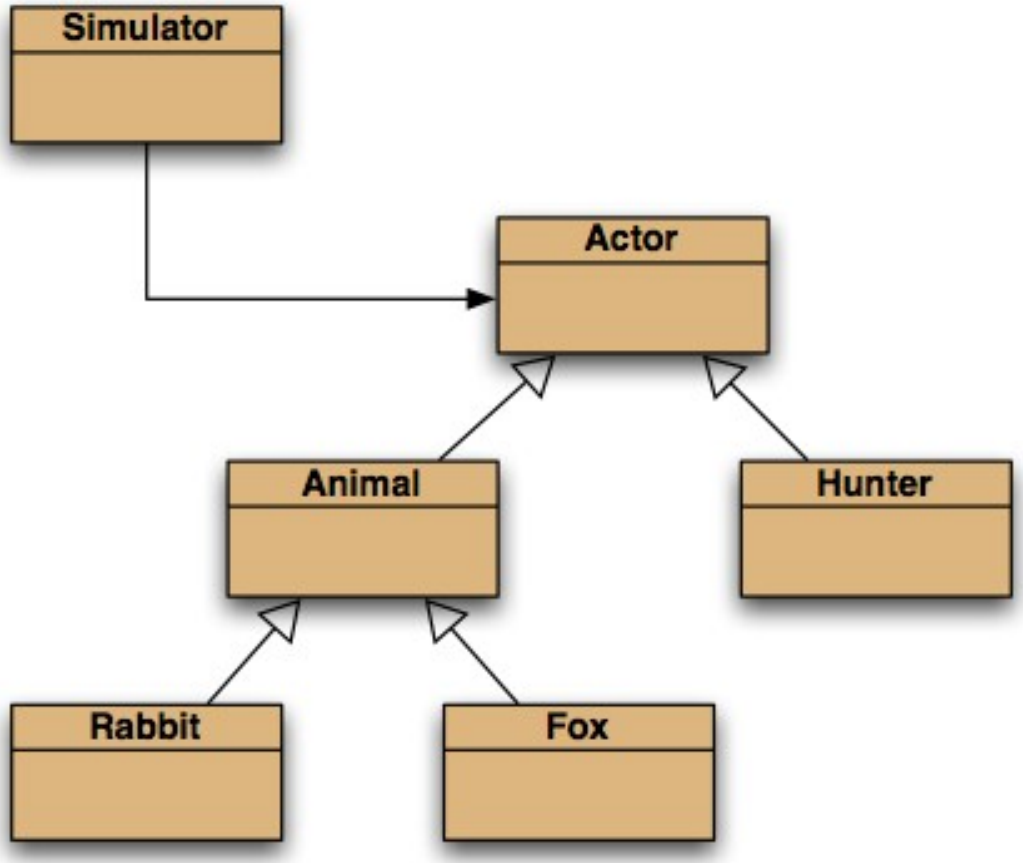
Abstraktní třídy a metody

- Abstraktní metody mají v hlavičce klíčové slovo **`abstract`**.
- Abstraktní metody nemají žádné tělo.
- Abstraktní metody způsobí, že se třída stane abstraktní.
- Abstraktní třídy nemohou vytvářet instance.
- Konkrétní podtřídy doplní implementaci.

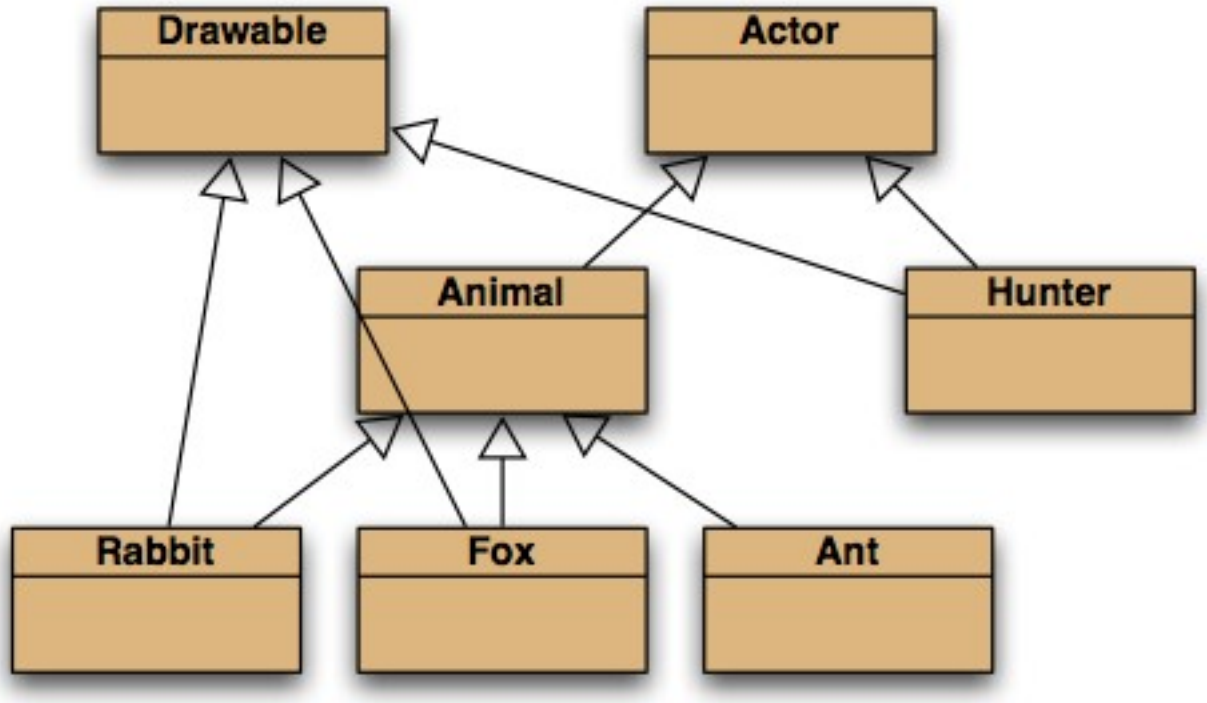
Třída Animal

```
public abstract class Animal
{
    instanční proměnné jsou vynechány

    /**
     * Způsobí, že zvíře udělá to, co chce/potřebuje
     * udělat.
     */
    abstract public void act(List<Animal> newAnimals);

    ostatní metody jsou vynechány
}
```

Další abstrakce



Selektivní zobrazování (vícenásobná dědičnost)



Vícenásobná dědičnost

- Vzniká, když máme třídu, která dědí přímo z více předků.
- Každý programovací jazyk má svá vlastní pravidla.
  - Jak řešit konkurující si definice?
- Java to zakazuje pro třídy.
- Java to povoluje pro interfejsy.

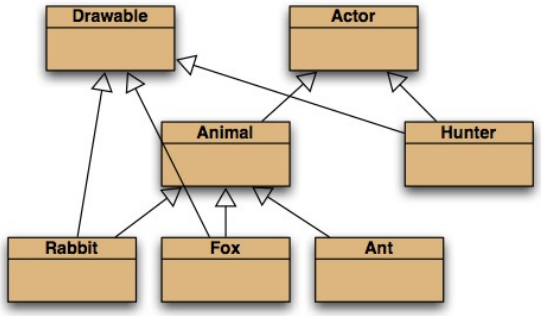
Interfejs Actor

```
public interface Actor
{
    /**
     * Provádí pravidelnou činnost aktéra.
     * @param newActors Seznam pro uložení nově
     *  vytvořených aktérů.
     */
    void act(List<Actor> newActors);

    /**
     * Je aktér ještě aktivní?
     * @return true jestliže je ještě aktivní, jinak
     * false.
     */
    boolean isActive();
}
```

Třídy implementují interfejs

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

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



Interfejsy v Javě

- Interfejs definuje objektový typ, podobně jako třída.
- Deklarace interfejsu může obsahovat pouze:
  - statické konstantní proměnné
  - abstraktní metody
  - statické metody
  - defaultní metody
- Interfejsy nemohou tvořit instance.
- Interfejsy lze implementovat v třídách.
- Interfejs lze deklarovat jako rozšíření několika jiných interfejsů (užívá se klíčové slovo extends).
- Další informace

Interfejsy v Javě

- Všechny abstraktní, defaultní a statické metody jsou implicitně **`public`** a je možné tento modifikátor vynechat.
- Všechny proměnné v interfejsu jsou implicitně **`public`**, **`static`** a **`final`**. Tyto modifikátory lze vynechat.

Interfejsy v Javě

- V případě, že přidáme do interfejsu další abstraktní metody, bude nutné upravit všechny třídy, které implementují původní verzi.
- Jak řešit tento nepříjemný problém?
- Buď navrhneme novou verzi interfejsu jako podinterfejs původní verze nebo užijeme defaultní či statické metody.

Defaultní metody

- Defaultní metody umožňují přidat do interfejsu novou funkcionalitu a zajistí binární kompatibilitu s původní verzí.
- Defaultní metody jsou deklarované s použitím klíčového slova **`default`** a jsou plně implementované.

Defaultní metody

- V případě rozšiřování interfejsu s defaultními metodami můžeme:
  - si nevšímat těchto metod a zdědit je.
  - je znovu implementovat a tak je překrýt.
  - je deklarovat jako abstraktní.

Statické metody v interfejsu

- Statické metody lze deklarovat i v interfejsu.
- Pro statické metody není nutné tedy vytvářet zvláštní třídy a je tak možné snáze navrhovat strukturu knihoven s těmito statickými metodami.

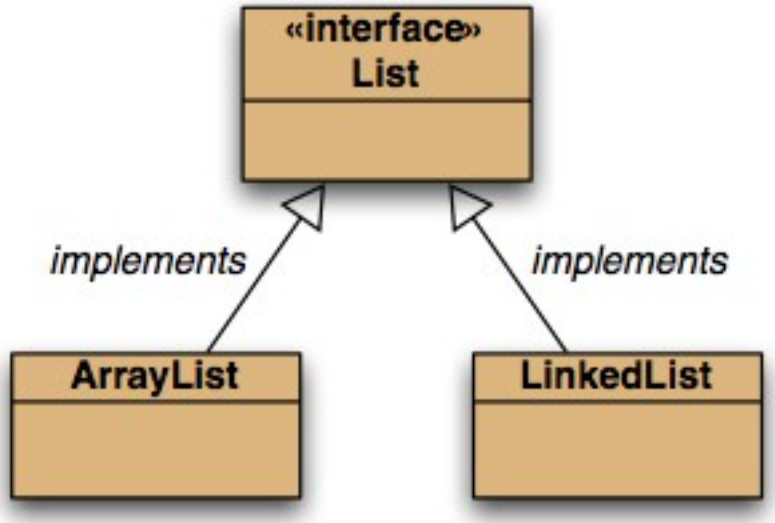
Interfejsy jako typy

- Implementující třídy nedědí přímo kód, ale...
- ... implementující třídy určují podtypy typu interfejsu.
- Takže polymorfizmus je použitelný s interfejsy stejně jako se třídami.

Interfejsy jako specifikace

- Interfejsy umožňují oddělit funkcionalitu od implementace.
  - Nicméně typy parametrů a návratové typy jsou povinné.
- Klienti interagují nezávisle na implementaci.
  - Ale klienti si mohou vybrat z alternativních implementací.

Alternativní implementace




```
List<E> seznam = new ArrayList<>();
```

Třída Class<T>

- Metoda **`getClass()`** ze třídy **`Object`** vrací instanci třídy **`Class`**, která reprezentuje runtime třídu objektu.
- Literál ze třídy **`Class<T>`** zapisujeme pomocí suffixu **`.class`** (př. **`Fox.class`**).
- Použito ve třídě **`SimulatorView_V2`**
  - **`Map<Class<? extends Animal>, Color> colors;`**
  - **`view.setColor(Rabbit.class,Color.ORANGE);`**
- Metoda **`getName()`** ze třídy **`Class<T>`** vrací jméno třídy.

Přehled

- Dědičnost poskytuje sdílenou implementaci.
  - Konkrétní a abstraktní třídy.
- Dědičnost poskytuje sdílenou typovou informaci.
  - Třídy a interfejsy.

Přehled

- Abstraktní metody umožňují provádět kontrolu založenou na statických typech bez požadavku na implementaci.
- Abstraktní třídy fungují jako neúplné nadtřídy.
  - Žádné instance.
- Abstraktní třídy podporují polymorfizmus.

Přehled

- Interfejsy poskytují specifikaci bez implementace (kromě defaultních a statických metod).
- Interfejsy podporují polymorfizmus.
- Interfejsy v jazyce Java podporují vícenásobnou dědičnost.