

به نام پروردگار دانایی

طراحی سیستم‌های شی گرا

برنامه‌نویسی شی گرا

درس ششم: کلاس‌های abstract

سید کاوه احمدی

شبیه‌سازی

■ برنامه‌های کامپیوتری خیلی از مواقع برای شبیه‌سازی فعالیت‌های پیچیده مورد استفاده قرار می‌گیرد.

— ترافیک شهری

— آب و هوا

— پروسه‌های اتمی

— نواسانات بازار سهام

— تغییرات زیست محیطی

شبیه‌سازی

- غالباً شبیه‌سازی‌ها جزئی هستند.
- معمولاً در آنها ساده‌سازی صورت می‌گیرد.
 - جزئیات بیشتر می‌تواند باعث بالا رفتن دقت شود.
 - جزئیات بیشتر معمولاً نیاز به منابع بیشتری دارد.
- توان‌پردازی
- زمان‌شبیه‌سازی

مزایای شبیه‌سازی

- پشتیبانی از پیش‌بینی‌های مفید.

 - آب و هوا

- اجازه آزمایش به ما می‌دهد.

 - ایمن‌تر، ارزانتر و سریعتر

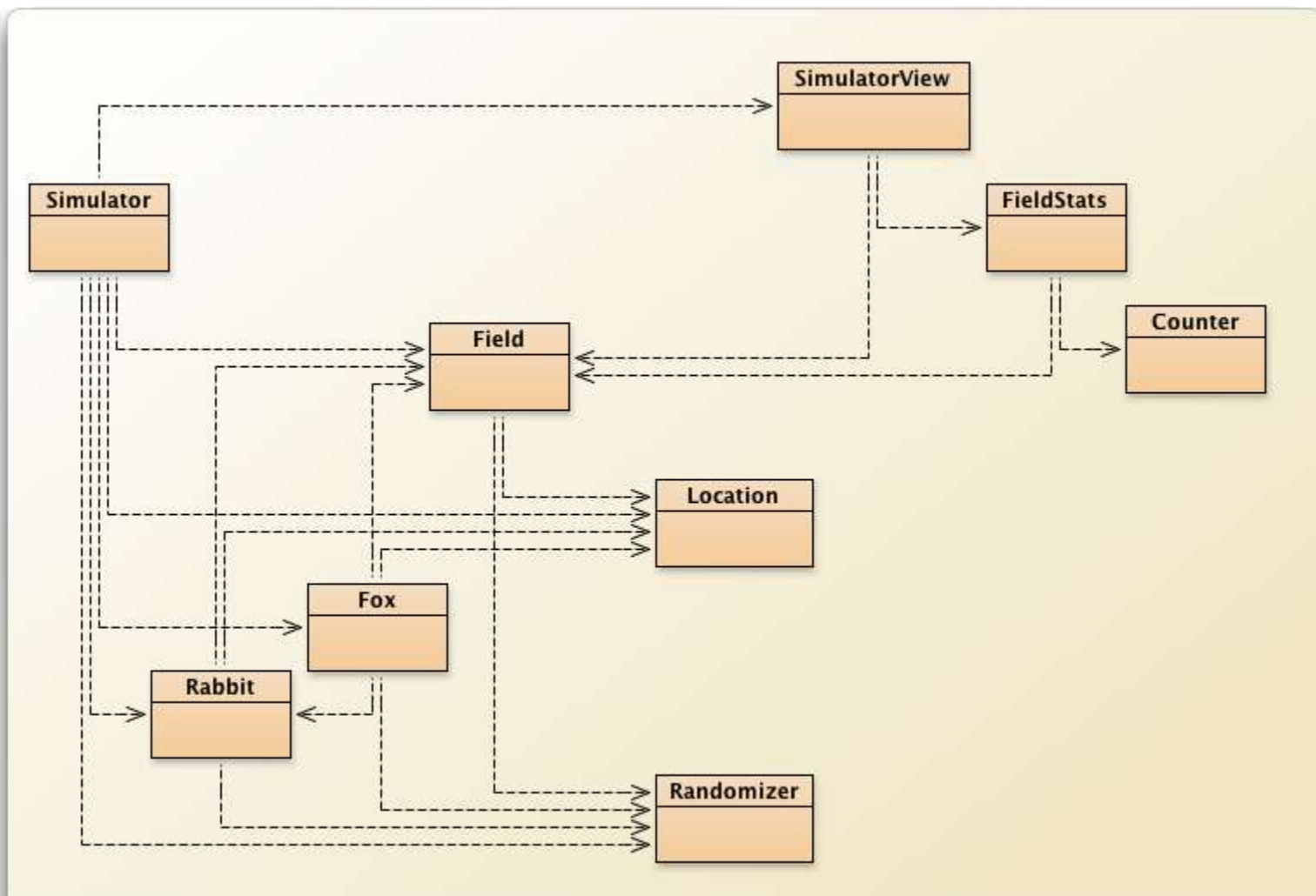
- مثال:

 - “حیات وحش چگونه تحت تاثیر قرار می‌گیرد اگر یک اتوبان از وسط پارک ملی عبور کند؟”

شبیه‌سازی شکار و شکارچی

- غالباً بین گونه‌های مختلف حیوانات یک تعادل برقرار است.
 - طعمه (شکار) زیاد به معنی غذای زیاد است.
 - زیاد بودن طعمه تعداد شکارچیان را بالا می‌برد.
 - شکارچیان زیاد غذای بیشتری می‌خورند.
 - کم شدن طعمه به معنی کم شدن غذا است.
 - کم شدن غذا به معنی...

پروژه روباه و خرگوش (foxes-and-rabbits)



کلاس‌های اصلی مورد بحث

FOX ■

— یک مدل ساده از شکارچی.

Rabbit ■

— یک مدل ساده از شکار

Simulator ■

— کار اصلی شبیه‌سازی را پیاده‌سازی می‌کند.

— یک مجموعه از روباه‌ها و خرگوش‌ها را نگهداری می‌کند.

سایر کلاس‌ها

Field ■

— یک عرصه دو بعدی را نمایش می‌دهد.

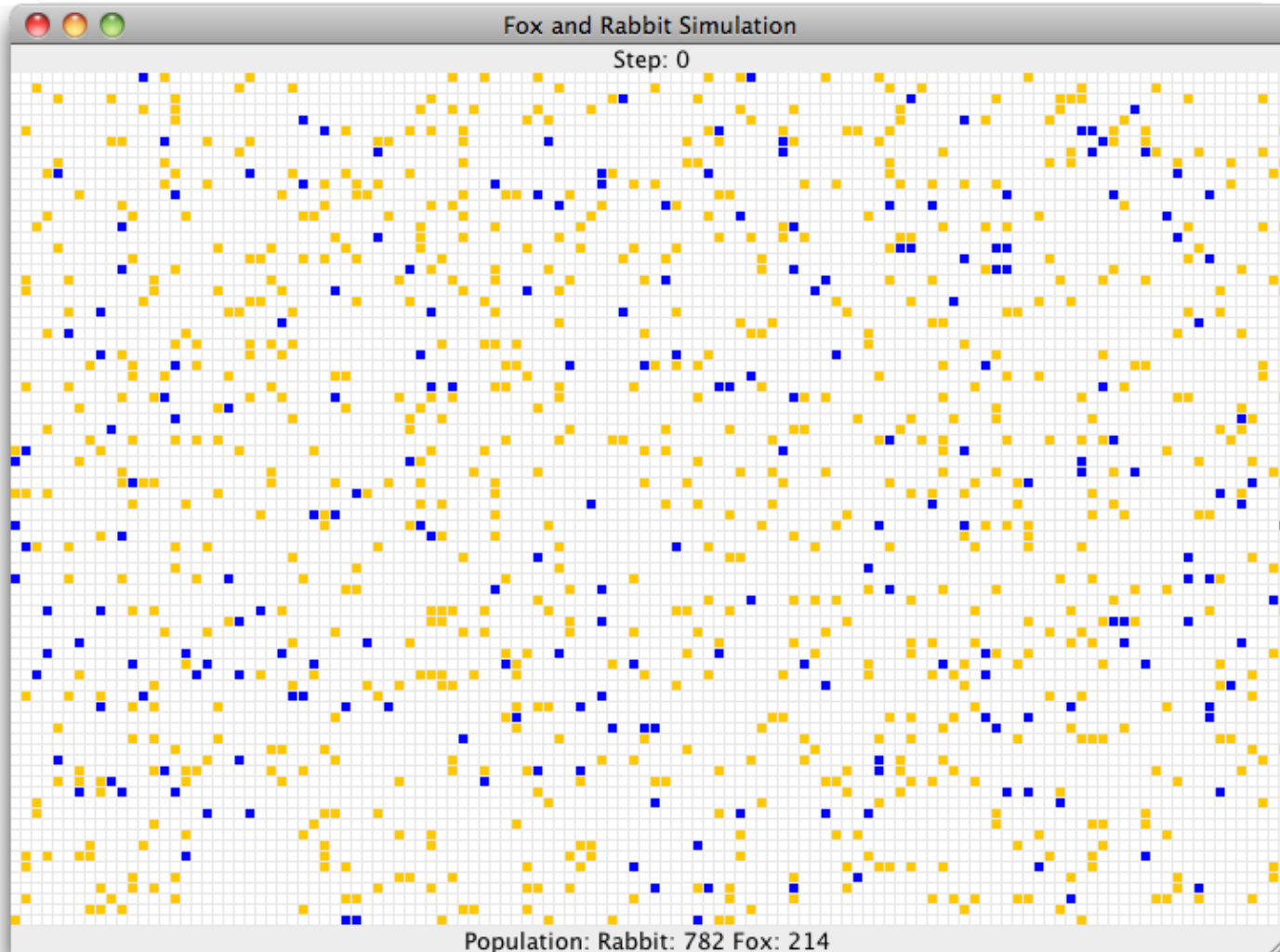
Location ■

— نمایش یک موقعیت دو بعدی.

SimulatorView, FieldStats, Counter ■

— حفظ آمار و ارائه‌نمایی از عرصه

نمایی از شبیه‌سازی



وضعیت یک خرگوش

```
public class Rabbit
{
    Static fields omitted.

    // Individual characteristics (instance fields).

    // The rabbit's age.
    private int age;
    // Whether the rabbit is alive or not.
    private boolean alive;
    // The rabbit's position
    private Location location;
    // The field occupied
    private Field field;

    Methods omitted.
}
```

رفتار یک خرگوش

- به وسیله متد **run** مدیریت می شود.
- در هر مرحله از شبیه سازی سن افزایش می یابد
 - خرگوش ممکن است در این مرحله بمیرد.
- خرگوش هایی که به اندازه کافی بزرگ شده اند می توانند زاد و ولد کنند.
 - خرگوش های جدید ممکن است در این مرحله متولد شوند.

ساده‌سازی خرگوش

- خرگوش‌ها جنسیت‌های مختلف ندارند.

— در عمل همه مونث هستند

- خرگوش‌ها می‌توانند در هر مرحله زاد و ولد کنند.

- همه خرگوش‌ها در سن مشابه می‌میرند.

- دیگر؟

```
public class Fox
{
    Static fields omitted

    // The fox's age.
    private int age;
    // Whether the fox is alive or not.
    private boolean alive;
    // The fox's position
    private Location location;
    // The field occupied
    private Field field;
    // The fox's food level, which is increased
    // by eating rabbits.
    private int foodLevel;

    Methods omitted.
}
```

- توسط متد **hunt** مدیریت می شود.
- روباهها هم سن شان بالا می رود و زاد و ولد می کنند.
- آنها گرسنه می شوند.
- در نقاط مجاور برای بدست آوردن غذا شکار می کنند.

پیکربندی روباه

- ساده‌سازی مشابه خرگوش.
- شکار کردن و خوردن می‌تواند به روش‌های متفاوتی مدل شوند.
 - آیا میزان غذا باید زیاد شود؟
 - آیا یک روباه گرسنه باید بیشتر شکار کند؟
- آیا ساده‌سازی همیشه قابل قبول است؟

کلاس Simulator

- سه جزء کلیدی

- راه اندازی در سازنده

- متد populate

- به هر حیوان یک سن اولیه تصادفی داده می شود.

- متد simulateOneStep

- پیمایش جمعیت های جداگانه ای از خرگوش ها و روباه ها

- دو شی Field مورد استفاده قرار می گیرد. Field و updatedField.

مرحله به روز رسانی (کلاس Simulator)

```
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();
    }
}
```

فضایی برای بهبود

- Fox و Rabbits مشابهت‌های بسیاری دارند اما یک ابرکلاس مشترک ندارند.
- مرحله به روز رسانی شامل کدهای مشابه هستند.
- Simulator شدیداً به کلاس‌های Fox و Rabbit چفت شده.
 - بیش از اندازه در مورد رفتار این کلاس‌ها می‌داند!
 - می‌داند آنها وجود دارند.
 - می‌داند که Fox دارای متد hunt و Rabbit دارای متد run است.
 - اگر بخواهیم Pigeon (کبوتر) اضافه کنیم، Simulator باید در مورد آن اطلاعاتی داشته باشد.
 - اضافه کردن ابرکلاسی مانند Animal به نظر عاقلانه است!

ابر کلاس Animal

- قرار دادن فیلدهای مشترک در Animal.
 - `location, alive, age`
- اسم متدها را برای پشتیبانی از پنهان سازی اطلاعات تغییر می دهیم.
 - `run` و `hunt` را به `act` تغییر می دهیم.
- حالا Simulator می تواند به طور قابل ملاحظه ای از این کلاس ها جدا شود.

- تا به اینجا نکته جدیدی ندیدیم!

- اما دیدیم:

- وراثت (اضافه کردن کلاس `Animal`) تکرار کد را برای کلاس‌های `Fox` و

- `Rabbit` در کلاس `Simulator` کاهش داد.

- استفاده از ابرکلاس `Animal` به عنوان نوع استاتیک، استفاده از چند ریختی را با استفاده از

- زیرنوع‌ها فراهم کرد.

تکرار تجدیدنظر شده (جدا شده)

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

■ بهبود

– Simulator از Fox و Rabbit جدا شده است.

– تکرار کد وجود ندارد.

■ توجه به چند ریختی

– نوع ایستا Animal است اما نوع پویا Fox یا Rabbit است.

– رفتار شی (شی فراخوانی شده) به نوع پویا بستگی دارد.

متد act در Animal

- بررسی نوع ایستا به متد act در Animal نیاز دارد.
- هیچ پیاده‌سازی مشترکی برای act (بین دو کلاس خرگوش و روباه) وجود ندارد.
- می‌توانیم یک متد خالی پیاده‌سازی کنیم.

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

- اما تضمینی وجود ندارد که زیر کلاس‌ها آنرا override کند.
— راه‌حل: ایجاد یک متد abstract.

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

کلاس‌ها و متدهای `abstract`

- متدهای `abstract` دارای کلمه کلیدی `abstract` در امضای خود هستند.
- متدهای `abstract` فاقد بدنه هستند (حتی `{ }` هم ندارند).
- متدهای `abstract` منجر به `abstract` شدن کلاس می‌شود.
 - اضافه کردن `abstract` به امضای کلاس هم می‌تواند آنرا `abstract` کند.
- نمی‌توان از روی یک کلاس `abstract` نمونه‌ای ایجاد کرد.
 - صرفاً برای تجلی زیرکلاس‌ها ایجاد می‌شود.
- زیرکلاس متصل، پیاده‌سازی را تکمیل می‌کند.
 - زیرکلاس‌ها هم `abstract` خواهد بود اگر متدهای `abstract` را پیاده‌سازی نکند.

```
public abstract class Animal
{
    fields omitted

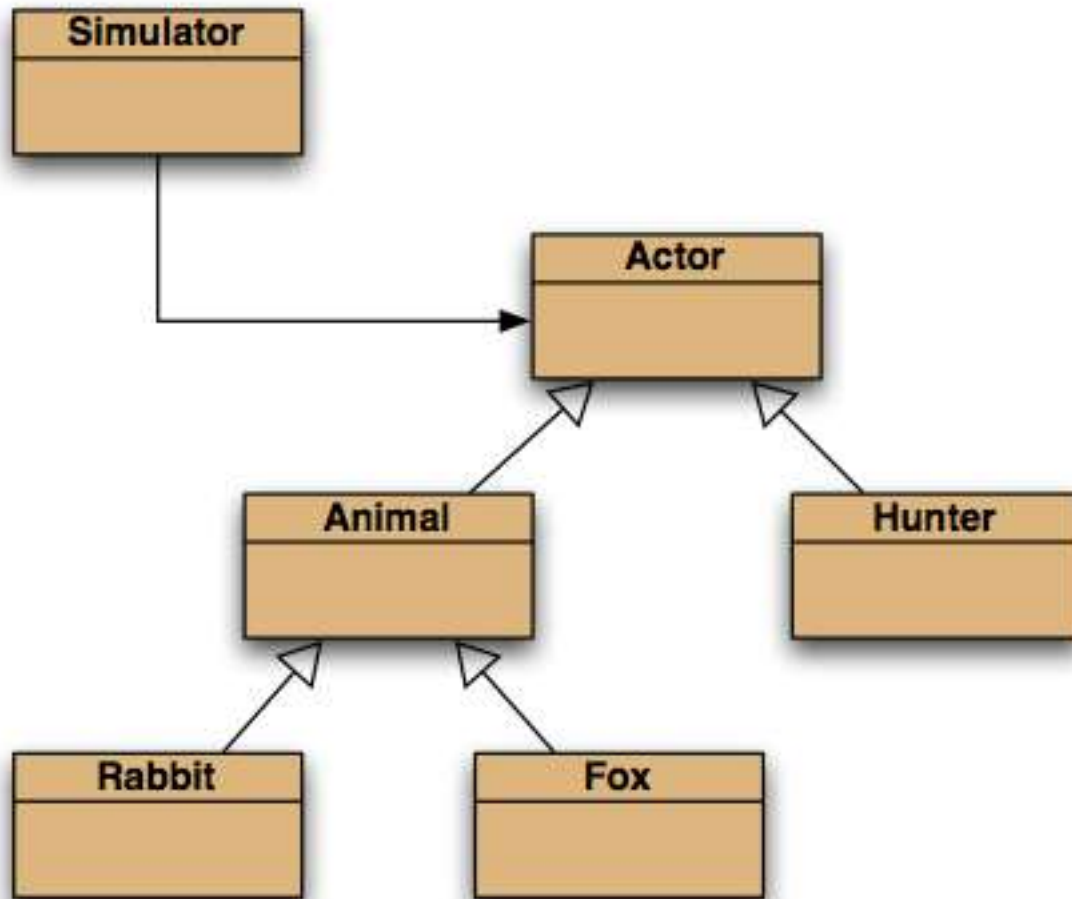
    /**
     * Make this animal act - that is: make it do
     * whatever it wants/needs to do.
     */
    abstract public void act(List<Animal> newAnimals);

    other methods omitted
}
```


چرا از کلاس‌های abstract استفاده کنیم؟

- دلایل مشابه مانند ابرکلاس‌ها
 - امکان وراثت در کد و داده‌ها (ارتقاء استفاده مجدد از کد، جلوگیری از تکرار کد).
 - امکان چندریختی (زیرکلاس نوع ابرکلاس را به خود می‌گیرد).
- دلایل بیشتر
 - مجبور کردن / تضمین کردن پیاده‌سازی متدهای معین در زیرکلاس‌ها.

abstraction بیشتر



abstraction بیشتر

```
public abstract class Actor
{
    abstract public void act(Field currentField,
        Field updatedField,
        List newAnimals);
}
```

```
public abstract class Animal extends Actor
{
    abstract public void act(Field currentField,
        Field updatedField,
        List newAnimals);
}
```

پیاده‌سازی متد `act` در کلاس‌های `Fox`، `Rabbit` و `Hunter`.