به نام پروردگار دانایی

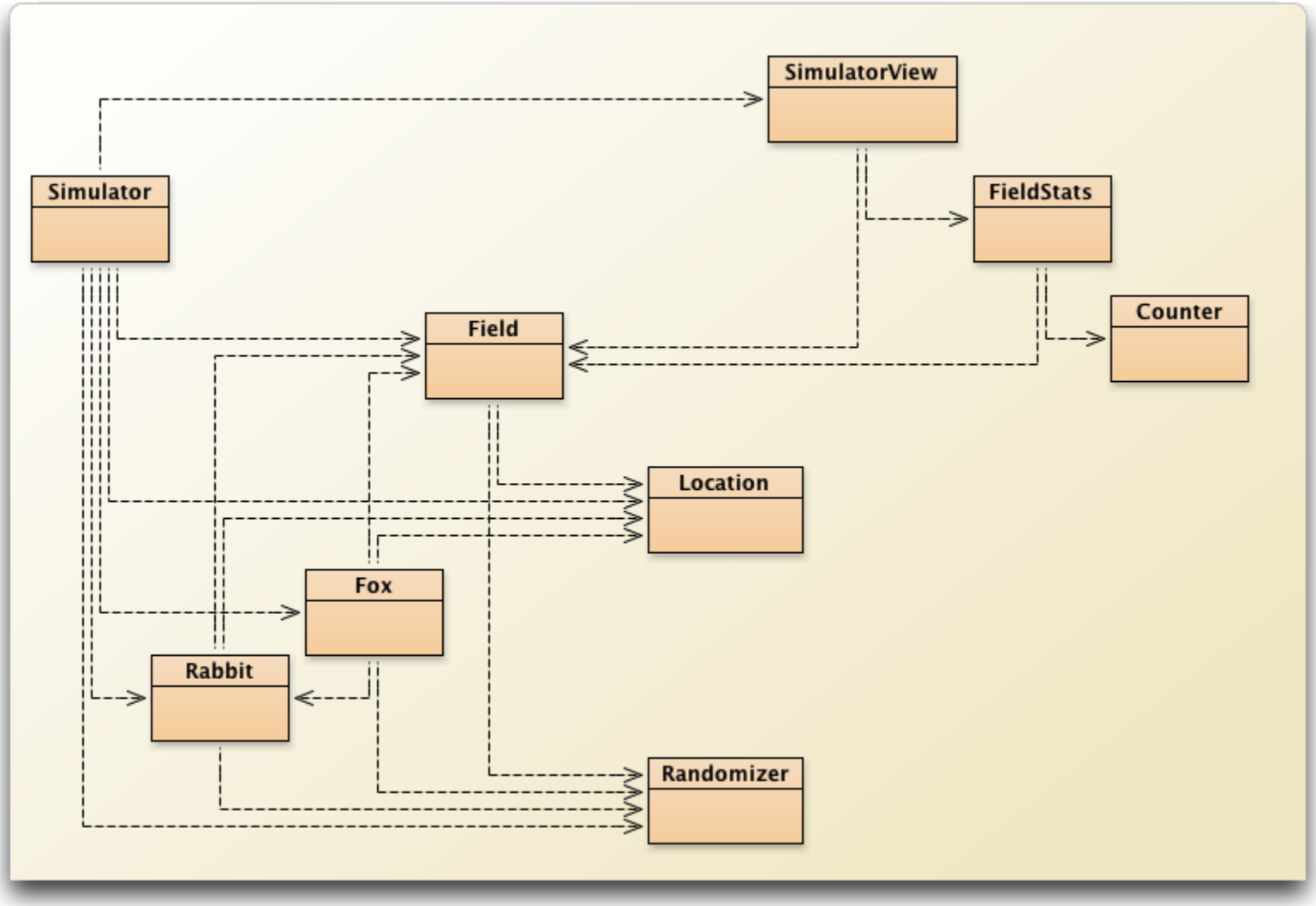
طراحی سیستم‌های شی‌گرا

برنامه‌نویسی شی‌گرا

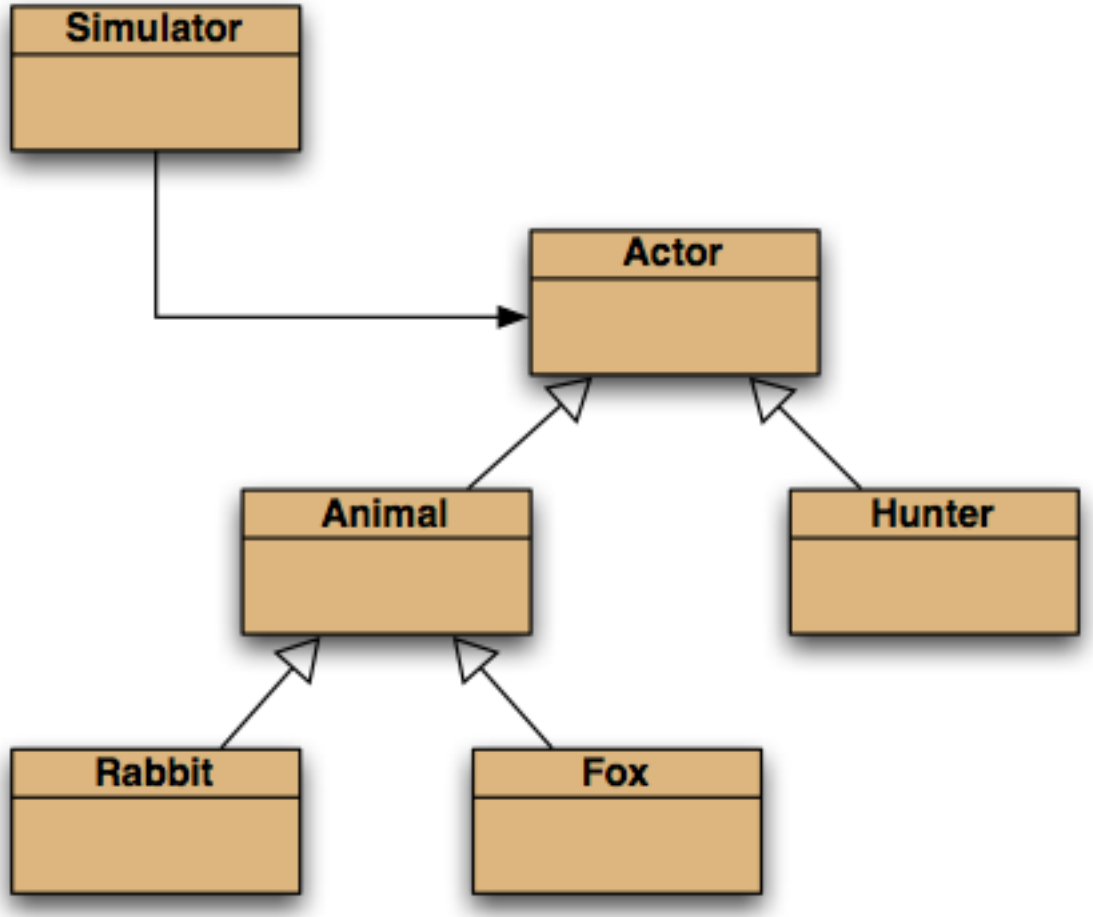
درس هشتم: interfaceها

سید کاوه احمدی

پروژه روباه و خرگوش (foxes-and-rabbits)



abstraction

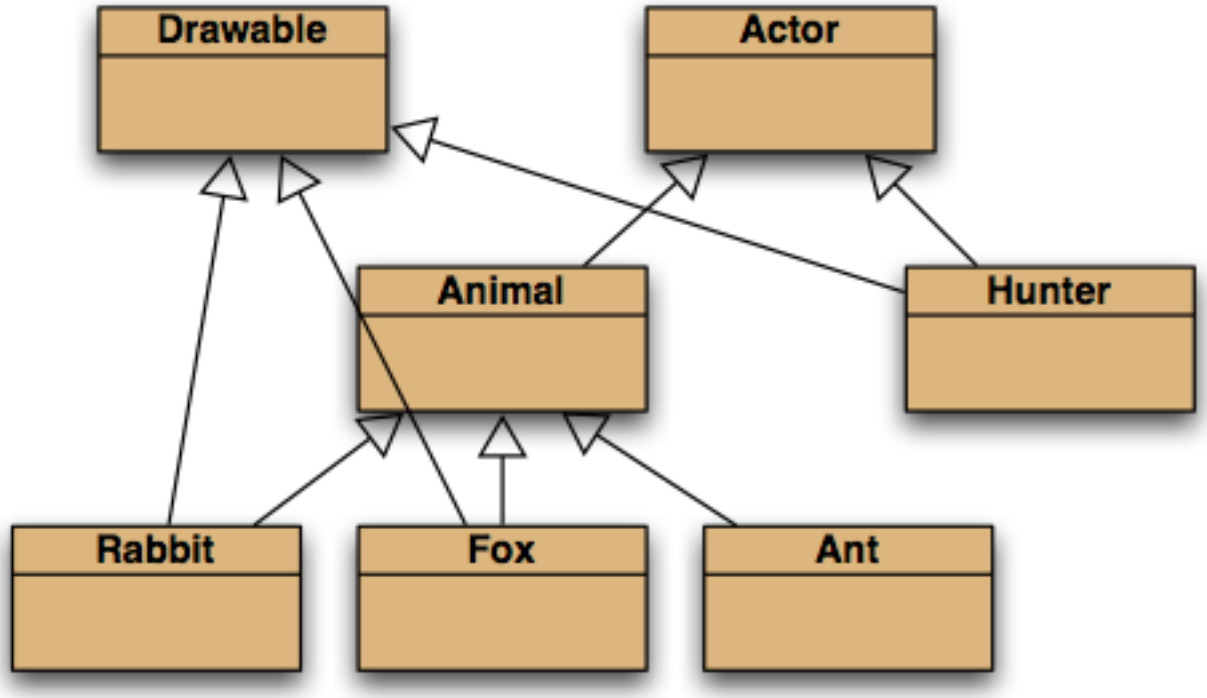


نمایش برخی از انواع اشیا

- می‌خواهیم برخی از اشیا را نمایش دهیم.
  - جداسازی نمایش (visualization) ازعمل (acting)
  - استفاده از وراثت چندگانه
    - اشیا هم Actor هستند و هم می‌توانند Drawable باشند.

وراثت چندگانه

- می‌توانیم اشیا کلاس‌های خاصی را روی صفحه نمایش دهیم (نه لزوما اشیا همه کلاس‌ها را).



نمایش برخی از انواع اشیا

```
for(Iterator iter = actors.iterator(); iter.hasNext(); )
{
      Actor actor = (Actor) iter.next();
      actor.act(...);
}
```

```
for(Iterator iter = drawables.iterator(); iter.hasNext();)
{
      Drawable item = (Drawable) iter.next();
      item.draw(...);
}
```

وراثت چندگانه

- وراثت چندگانه یعنی یک کلاس مستقیما از چند کلاس ارث‌بری داشته باشد.
- مشکل زمانی بوجود می‌آید که دو ابرکلاس متد یکسانی را پیاده‌سازی کرده باشند.
  - کدام پیاده‌سازی مورد استفاده قرار می‌گیرد؟
  - هر زبان قوانین خاص خود را دارد.
- جاوا وراثت چندگانه از کلاس‌های عادی و abstract را منع می‌کند.
- اما این امر از interfaceها مجاز است.
  - یک نمونه از کلاس کاملا abstract (فقط امضای متدها بدون هیچگونه پیاده‌سازی‌ای).
    - بنابراین هیچ پیاده‌سازی متداخلی در وراثت چندگانه رخ نخواهد داد.
  - Interfaceها می‌توانند interfaceهای چندگانه دیگر را گسترش دهند.

یک Interface (Actor)

```
public interface Actor
{
    /**
     * Perform the actor's regular behavior.
     * @param newActors A list for storing newly created
     *                  actors.
     */
    void act(List<Actor> newActors);

    /**
     * Is the actor still active?
     * @return true if still active, false if not.
     */
    boolean isActive();
}
```

یک نوع کلاس است

متدها public و abstract است و فاقد پیاده‌سازی

کلاس‌هایی که interface را پیاده‌سازی می‌کنند

- یک کلاس abstract معین را گسترش می‌دهیم در حالی که interfaceها را پیاده‌سازی می‌کنیم.

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


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

Hunter چند نوع دارد؟

interface به عنوان نوع

- پیاده‌سازی کلاس‌ها کدی به ارث نمی‌برند. اما...
- ... کلاس‌های پیاده‌سازی شده زیر نوع‌هایی از نوع interface هستند.
- بنابراین چند ریختی با interface مانند کلاس‌ها وجود دارد.

خصوصیات interface

- تمام متدها abstract هستند.
- نمی‌تواند دارای سازنده باشد.
- تمام متدها public هستند.
- تمام فیلدها public، static و final است.
- به همین خاطر نوشتن لغات کلیدی abstract، public، static و final اختیاری است.

Interface

- یکی از استفاده‌های مهم interface در جاوا، پیاده‌سازی وراثت چندگانه است.
- این کارکرد مهم است اما استفاده‌های دیگری نیز برای Interface وجود دارد.

کارکردهای اینترفیس: interface برای تعریف مشخصات

- جداسازی مطلق عملکرد از پیاده‌سازی
  - اگرچه پارامترها و نوع بازگشتی یکسان اجباری است
- برقراری ارتباط توسط کلاینت‌ها مستقل از پیاده‌سازی است.
  - اما می‌توانند بین پیاده‌سازی‌های مختلف انتخاب کنند.

کارکردهای اینترفیس: پیاده‌سازی عملکردها و رفتارها

- وراثت: پیاده‌سازی رفتارهای مشترک (با پیاده‌سازی مشترک – کد).
- کلاس‌های abstract: اجبار به پیاده‌سازی رفتارهای مشخص.
  - Interface همانند کلاس abstract نیز می‌تواند برای این کار استفاده شود.
- Interface: پیاده‌سازی عملکردهایی که رفتارهای مختلفی را می‌طلبد.
  - مقایسه
    - عمل مقایسه اشیا در کلاس‌های مختلف، عملکرد مشترکی (مقایسه) است که کلاس‌های متفاوت به آن نیاز دارند اما رفتارهای (پیاده‌سازی – مفهوم) متفاوتی را می‌طلبد.
      - این مفهوم با وراثت متفاوت است!
  - مرتب سازی

Comparable

- **Interface Comparable<T>**
  - T: the type of objects that this object may be compared to
  - `int compareTo(T o)`
    - Compares this object with the specified object for order.

- همچنین ببینید:
  - `Comparator`
  - `Sortable`

مثال

```
public interface Edible {
  /** Describe how to eat */
  public String howToEat();
}
```

```
class Animal {
  ...
}

class Chicken extends Animal implements Edible {
  public String howToEat() {
    return "Fry it";
  }
}

class Tiger extends Animal {
  ...
}
```

زیرکلاس‌ها می‌توانند Edible باشند.

مثال

```
public interface Edible {
  /** Describe how to eat */
  public String howToEat();
}
```

```
abstract class Drinks implements Edible {
  public final String howToEat() {
    return "Drink it";
  }
}

class Pepsi extends Drinks {
        …
}
```

تمام زیرکلاس‌ها Edible هستند و یک پیاده‌سازی مشترک دارند که در ابرکلاس پیاده‌سازی شده (و قابل تغییر در زیر کلاس‌ها هم نیست).

مثال

```
public interface Edible {
  /** Describe how to eat */
  public String howToEat();
}
```

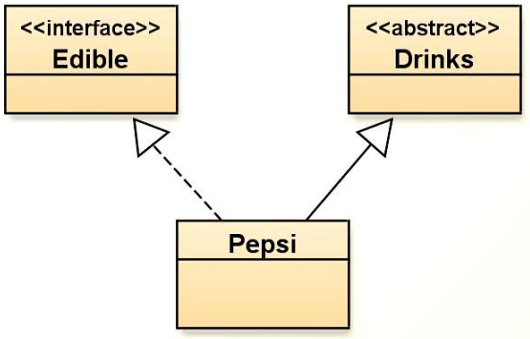
```
abstract class Fruit implements Edible {
  ...
}

class Apple extends Fruit {
  public String howToEat() {
    return "Make apple cider";
  }
}

class Orange extends Fruit {
  public String howToEat() {
    return "Make orange juice";
  }
}
```

تمام زیرکلاس‌ها Edible هستند. اما هرکدام باید پیاده‌سازی خاص خود را داشته باشند.

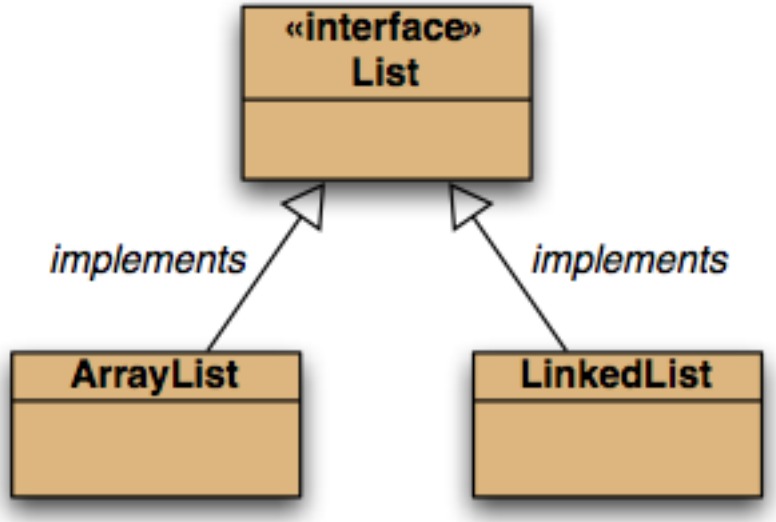
مثال

```
public interface Edible {
  /** Describe how to eat */
  public String howToEat();
}
```

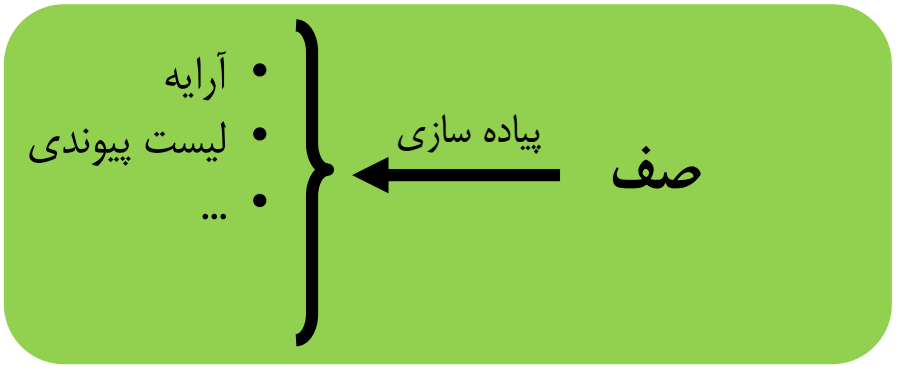
```
abstract class Drinks {
  public String howToEat() {
    return "Drink it";
  }
}

class Pepsi extends Drinks implements Edible {

}
```



کارکردهای اینترفیس: پیاده‌سازی‌های جایگزین



تجرید رفتار (Behavioral Abstraction)



پیاده کردن چندین interface

```
public interface Animal
{
    public void eat( );
}

public interface Cat
{
    void purr();  // public by default;
}

public class Lion implements Animal, Cat
{
    public void eat() {System.out.println("Big Gulps");}
    public void purr()  {
           System.out.println("ROOOAAAR!");
    }
}
```

واسط‌های ناسازگار

- در جاوا هر کلاس فقط می تواند یک کلاس پایه داشته باشد.
  - تا دو متد با سرآیند یکسان، تعاریف متعدد نداشته باشند.
- اما هر کلاس می تواند چندین واسط را پیاده کند.
  - چون که متدهای واسط بدنه ندارند، این موضوع ایرادی ندارد.
  - اما مشکلات دیگری وجود دارد.
- اگر واسط‌ها دو متد با اسم یکسان و مقدار برگشتی متفاوت داشته باشند، مشکل پیش می‌آید.
- اگر کلاسی دو واسط ناسازگار را پیاده کند، خطا تولید شده و تعریف کلاس غیرقانونی است.

interface یا abstract

- می‌خواهیم کد به ارث برده شود؟
  - بله: استفاده از وراثت
  - خیر: interface بهتر است چون از گسترش کلاس جلوگیری نمی‌کند.
    - به جز زمانی که محدودیت‌های interface مشکل ایجاد می‌کند.

```
A extends B  //A cannot later extend C

A implements B //A can later extend C
```

متدها و کلاس‌های final

- یک متد final نمی‌تواند توسط زیرکلاس‌ها override شود.
  - برخلاف متدهای abstract که باید حتما override می‌شد (یا زیرکلاس هم abstract می‌شد).
- یک کلاس final نمی‌تواند extend شود.
  - برعکس یک کلاس abstract.
- چرا؟
  - برای جلوگیری از override شدن یک متد توسط زیرکلاس‌ها.