

به نام پروردگار دانایی

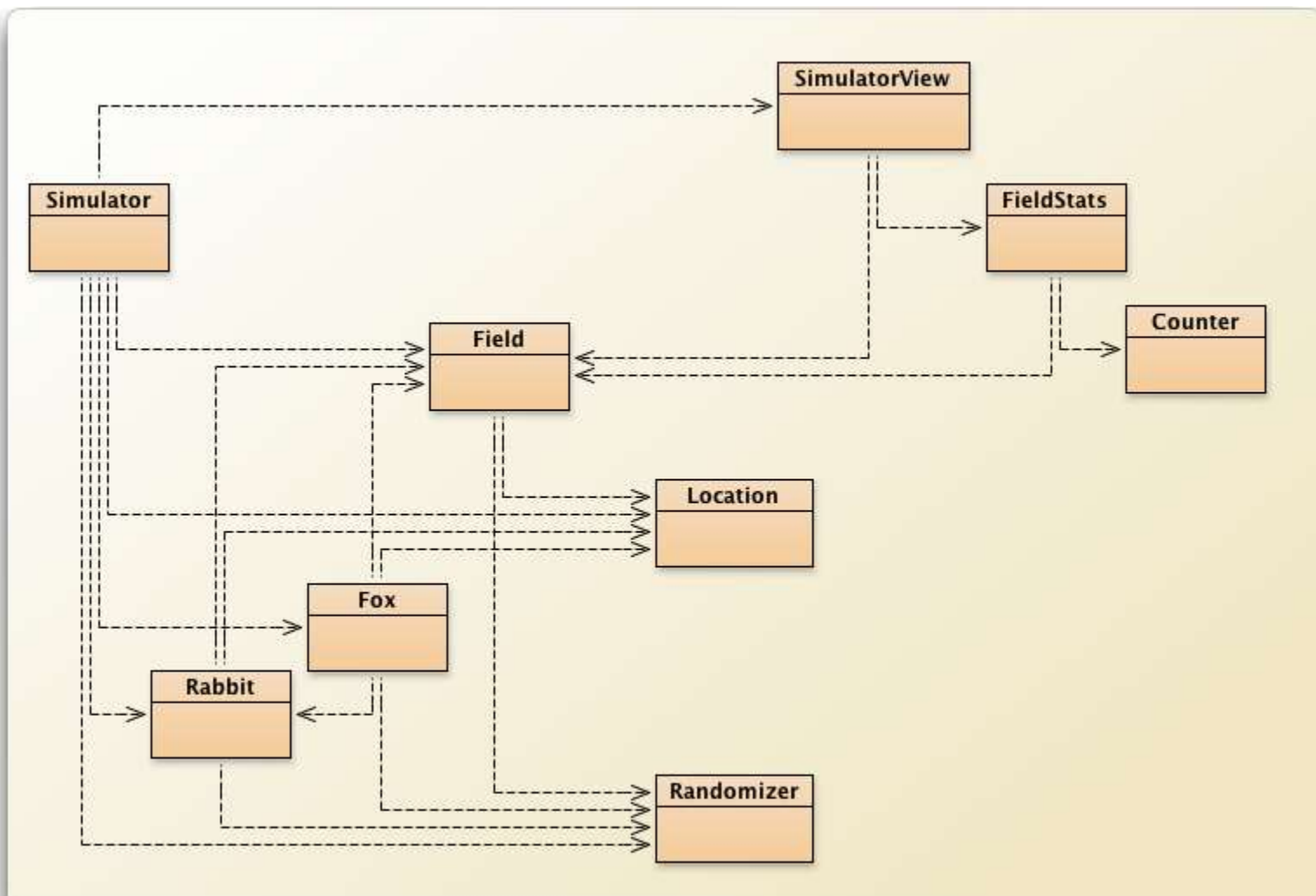
طراحی سیستم‌های شی گرا

برنامه‌نویسی شی گرا

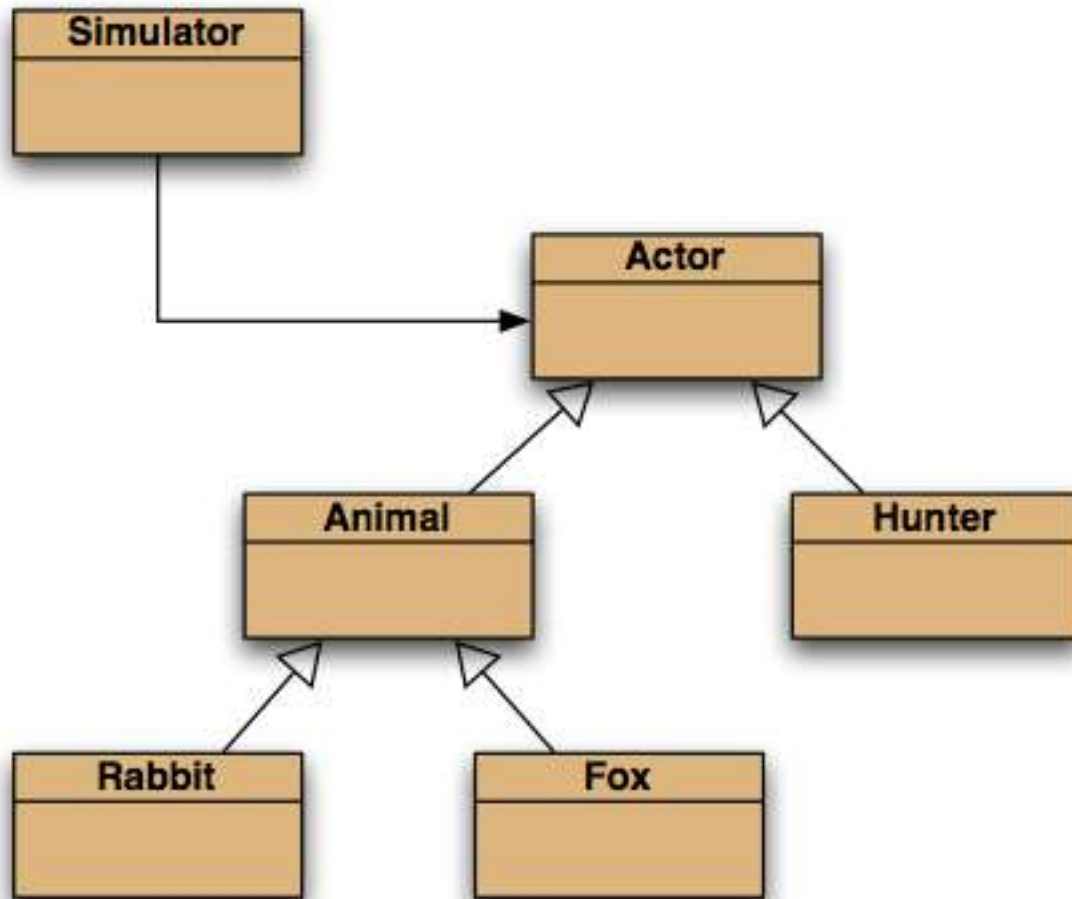
درس هفتم: interfaceها

سید کاوه احمدی

پروژه روباه و خرگوش (foxes-and-rabbits)



abstraction



نمایش برخی از انواع اشیا

- می‌خواهیم برخی از اشیا را نمایش دهیم.

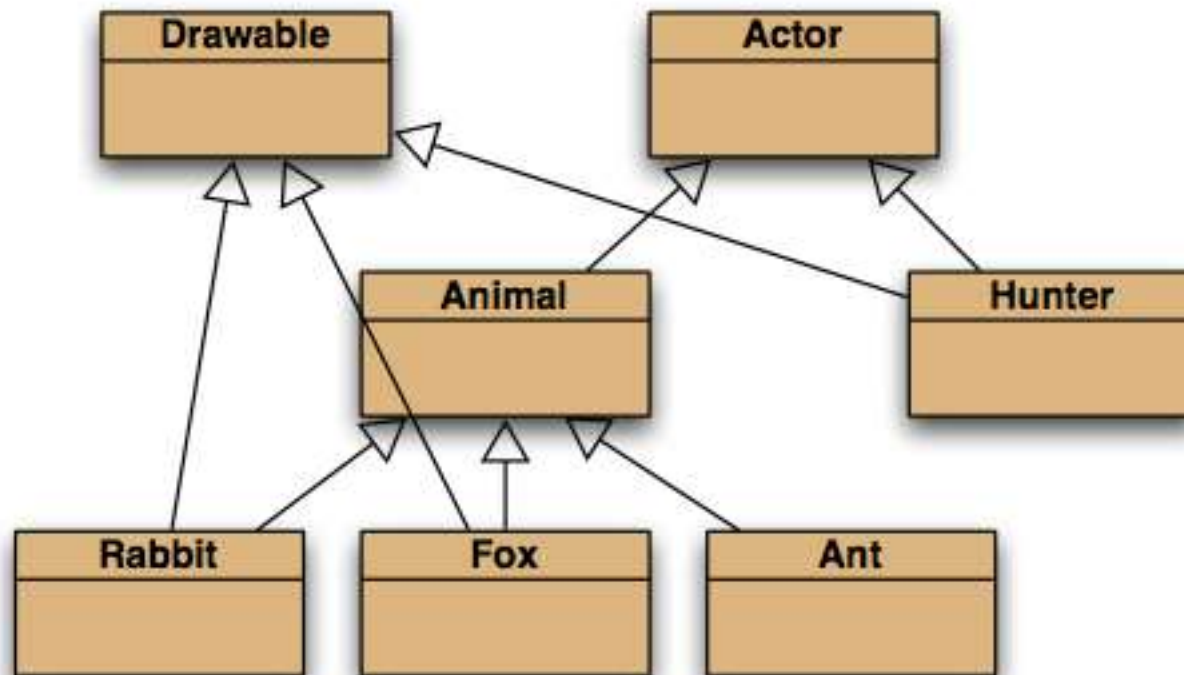
- جداسازی نمایش (visualization) از عمل (acting)

- 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 لزوماً 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 مستقیماً 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 `interface`‌ها مجاز است.
 - یک نمونه از کلاس کاملاً `abstract` (فقط امضای متدها بدون هیچگونه پیاده‌سازی‌ای).
 - بنابراین هیچ پیاده‌سازی متداخلی در وراثت چندگانه رخ نخواهد داد.
 - `Interface`‌ها می‌توانند `interface`‌های چندگانه دیگر را گسترش دهند.

یک (Actor) Interface

```
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 interface 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.

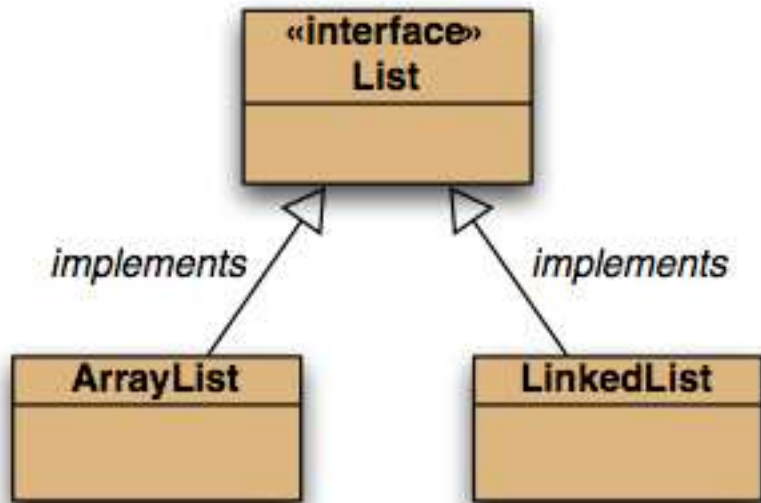
خصوصیات interface

- 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.

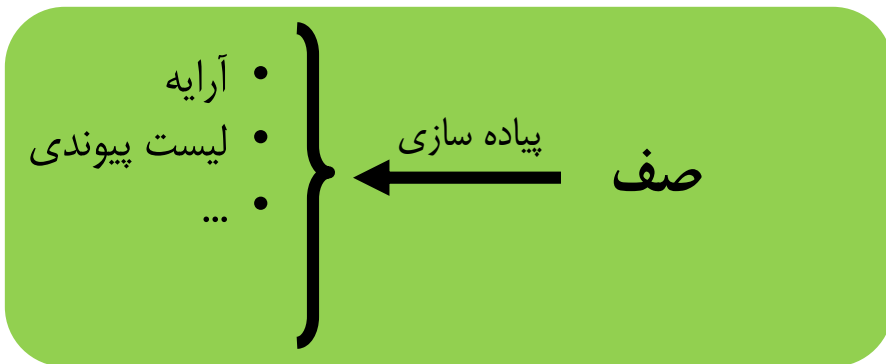
کارکردهای اینترفیس ۱: پیاده‌سازی‌های جایگزین



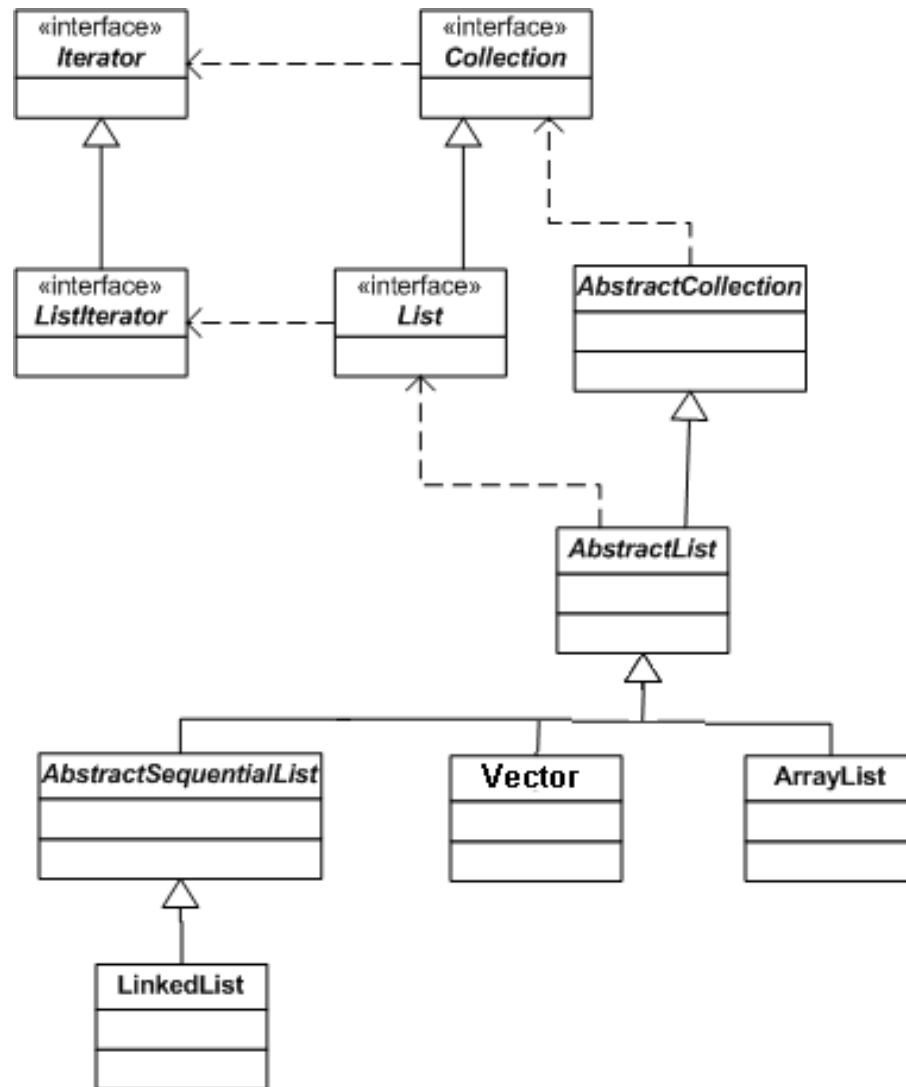
- اگر کلاسی بنویسیم که اینترفیس List را پیاده‌سازی کند، جاوا آنرا به عنوان یک لیست از ما می‌پذیرد و می‌توانیم عملیات مربوط به لیست‌ها را روی آن به کار بگیریم.

- همانند استفاده از کلاس Iterator برای آن و یا حتی استفاده از حلقه‌ی for each و...

- عمل‌های همان تجرید رفتار (Behavioral Abstraction) فراهم شده است.



The List Interface



کارکردهای اینترفیس ۲: 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.

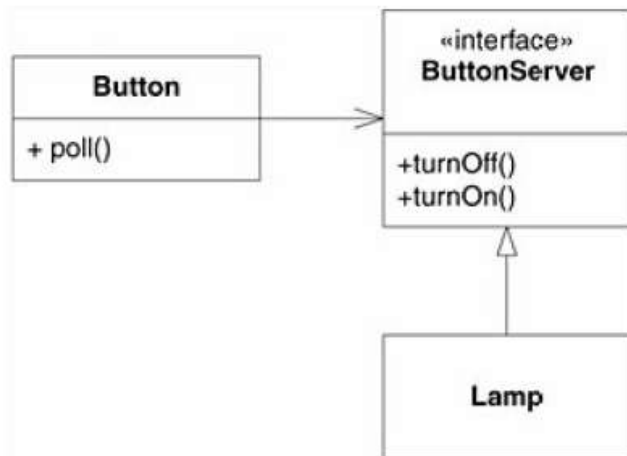
کارکردهای اینترفیس ۲: interface برای تعریف مشخصات

- در این طراحی کلاس Button به واسطه ButtonServer وابسته است.

- این واسطه ButtonServer، قراردادی برای روشن و خاموش کردن چیزی تعریف می کند و کلاس Button به این انتزاع وابسته می شود.

- کلاس Lamp واسطه ButtonServer را پیاده سازی می کند.
- کلاس Lamp است به واسطه ButtonServer وابسته است.

- طراحی فوق باعث می شود که کلاس Button با هر شی ای که ButtonServer را پیاده سازی می کند، کار کند مانند یک موتور، پنکه و غیره. و بالعکس Lamp می تواند از هر کلاسی که واسطه ButtonServer را پیاده سازی کرده باشد استفاده نماید.



کارکردهای اینترفیس ۳: 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 عملکردهای مشابه و قابل ارجاع در کلاس‌های مختلف در حالتی که پیاده‌سازی مشترکی برای همه‌ی کلاس‌ها وجود ندارد.
- مثلاً مقایسه‌ی نمونه‌های ایجاد شده از یک کلاس با یکدیگر
- عمل مقایسه‌ی اشیا در کلاس‌های مختلف، عملکرد مشترکی است که کلاس‌های متفاوت به آن نیاز دارند
- اما رفتارهای (پیاده‌سازی - مفهوم) متفاوتی را می‌طلبند.
- این مفهوم با وراثت متفاوت است!
- مرتب‌سازی

کارکردهای اینترفیس ۳: پیاده‌سازی عملکردها و رفتارها

- فرض کنید یک لیست از کارمندان به شکل زیر داریم:

- `ArrayList <Employee> employees`

- می‌خواهیم لیست بالا را مرتب کنیم. `ArrayList` متدی به اسم `sort`

برای این منظور دارد.

- سوال این است که مبنای مقایسه کارمندان برای این `sort` چگونه تعیین می‌شود؟

کارکردهای اینترفیس ۳: پیاده‌سازی عملکردها و رفتارها

- API متد `sort` سرنخ‌هایی به ما می‌دهد:

```
public static void sort(List list)
```

Sorts the specified list into ascending order, according to the natural ordering of its elements. All elements in the list must implement the Comparable interface. Furthermore, all elements in the list must be mutually comparable (that is, `e1.compareTo(e2)` must not throw a `ClassCastException` for any elements `e1` and `e2` in the list).

- همانطور که مشخص است برای اینکه بتوان از متد `sort` استفاده کرد، تمام المان‌های درون لیست باید اینترفیس `Comparable` را پیاده‌سازی کنند (و در نتیجه متد `compareTo` را داشته باشند).

- متد `sort` با استفاده از متد `compareTo` (که مطمئن هستیم همه اشیا آنرا دارند) و مقایسه‌ی اشیا با یکدیگر، عمل مرتب‌سازی را انجام می‌دهد.

Comparable Interface

- **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.

- می‌دانیم اشیای هر کلاسی که این اینترفیس را پیاده‌سازی کرده باشد (Comparable باشد) می‌تواند با اشیای از کلاس‌های مشخص شده مقایسه شود.

- همچنین ببینید:

- Comparator

- Sortable

```
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 {  
    ...  
}
```

زیر کلاس‌ها می‌توانند
Edible باشند.

```
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 زیر کلاس‌ها
Edible هستند و یک
پیاده‌سازی مشترک دارند که
در ابر کلاس پیاده‌سازی شده
(و قابل تغییر در زیر
کلاس‌ها هم نیست).

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

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

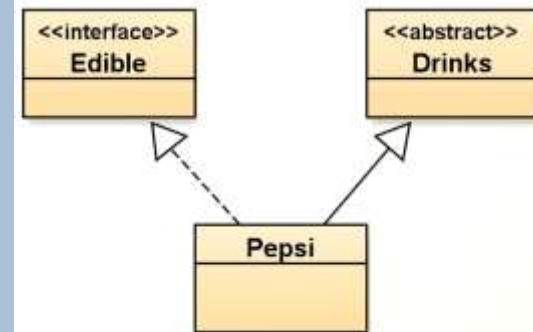
```
class Apple extends Fruit {  
    public String howToEat() {  
        return "Make apple cider";  
    }  
}
```

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

تمام زیر کلاس‌ها
Edible هستند. اما
هر کدام باید پیاده‌سازی
خاص خود را داشته باشند.

```
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 {  
}
```



پایاده 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 می‌تواند یک کلاس پایه داشته باشد.

- تا دو متد با سرآیند یکسان، تعاریف متعدد نداشته باشند.

- اما هر کلاس می‌تواند 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 حتماً `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.