Iterator Pattern

By Group of Four

A problem to be solved

- When you have different collections which share some identity and you want them to iterate without violate encapsulation

Situation : The livestock farm

- Two livestock farm is about to merge.
- Microsoft wants to keep his livestock in ArrayList so that Lou can expand his easily.
- On the other hand Apple wants to keep his in Array so he can control the maximum size of his livestock.

The livestock

- Both Microsoft and Apple agreed on livestock's implementation

```
public class Livestock(){
        String name;
        String category;
        boolean sex; //true – male, false – female
        int age;
        public Livestock(String name,String category,boolean sex, int age)
        { … }
        public String getName(){ … }
        public String category(){ … }
        public boolean isMale(){ … }
        public int getAge(){ … }
```

Lets look at each farm's code

- Microsoft's

```
public class MicrosoftList{
        ArrayList livestocks;
        public MicrosoftList(){
          livestockList = new ArrayList();
          addLivestock(new     Livestock(
                XP,horse,true,10 ) );
         addLivestock (new     Livestock(
                Vista,cow,true,7 ) );
        }
        public void addLivestock(Livestock l){
          livestocks.add(l);
        }
        public ArrayList getLivestockList(){ . }
        // some other methods
}
```

- Apple's

```
public class AppleList{
        static final int MAX_LIVESTOCK = 5;
        int count = 0;
        Livestock[] livestocks;
        pulic AppleList(){
          livestocks = new
            Livestock[MAX_LIVESTOCK];
          addLivestock(new
              Livestock(tiger,cow,false,9));
          addLivestock(new
              Livestock(leopard,cow,false,5));
        }
        public void addLivestock(Livestock l){
           if (count >= MAX_LIVESTOCK)
              //prints error
          else{ livestocks[count] = l; count++}
        }
        public Livestock[] getLivestocks(){…}
```

The inconvenient truth

- When the doctor comes to check this new farm he has to check every livestock so he has to get a list from a farm
- Printing a list will be a problem due to the difference in collection of each livestocks
- Who would wants to go back and change the implementation

Iterator saves the day

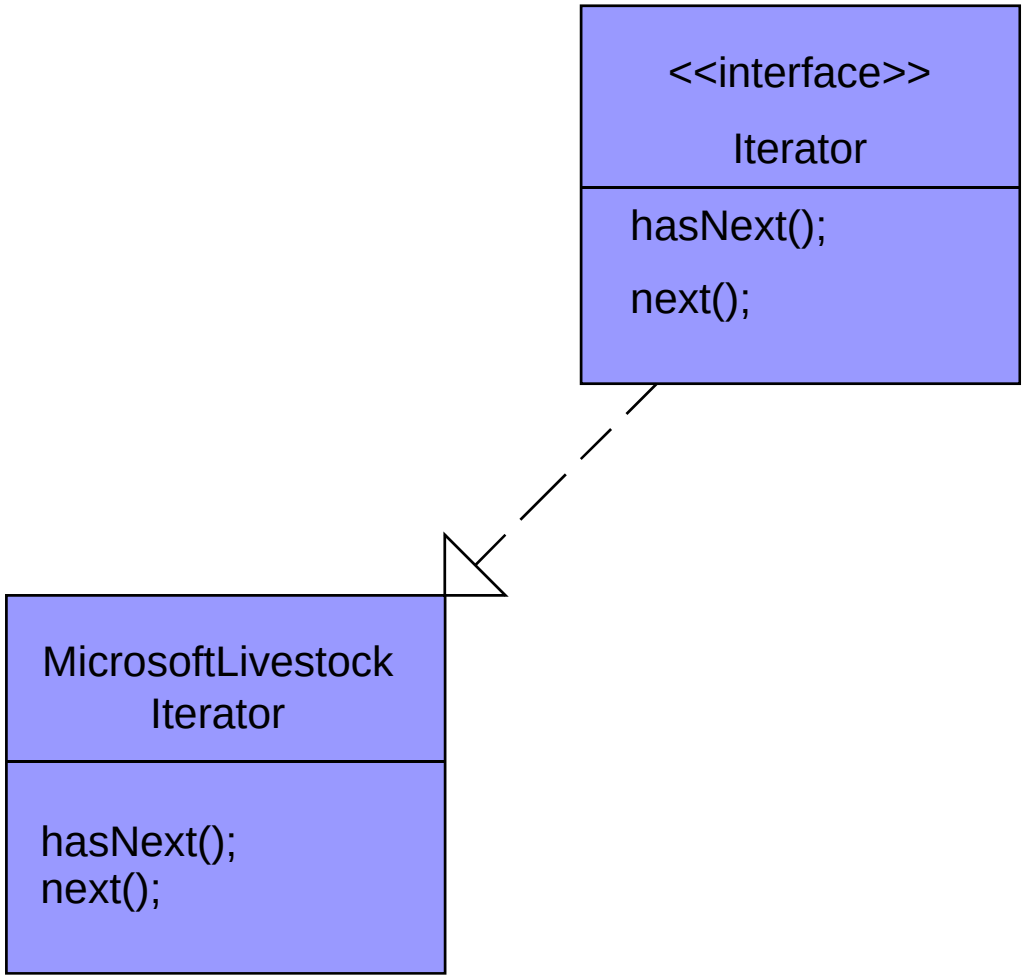
- Using Iterator which is super class of collections
- Iterator encapsulates the way we iterate our objects

We don't have to create our own iterator, we just call the Iterator() methods provided in library

```
Iterator iterator = microsoftList.createIterator() //ask for an iterator of its livestock
while ( iterator.hasNext( ) ){ Livestock livestocks = (Livestock)iterator.next(); }

Iterator iterator = appleList.createIterator() //ask for an iterator of its livestock
while ( iterator.hasNext( ) ){ Livestock livestocks = (Livestock)iterator.next(); }
```

The Design model



Creating aggregate

- Microsoft's

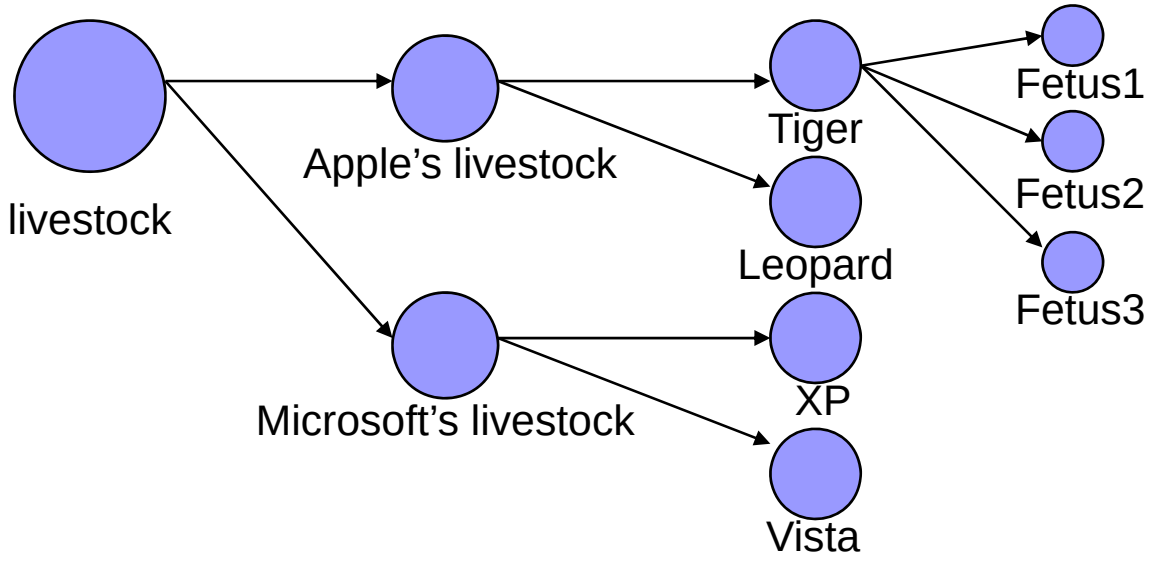
```
public class AppleLivestockIterator implements Iterator{
        Livestock[] llist;
        int index = 0;
        public AppleLivestockIterator (Livestock[] list){
             this.list = list;
        }
        public Object next(){
            return (Livestock) list[index++];
        }
        public boolean hasNext(){
            if ( index < list.length || list[index] != null ) return true;
            else return false;
        }
}
```

The doctor's getting the list

- So we can easily print a list by using Iterator without knowing which collection is used in each farm

The problem beyond expectation

- What if some livestock is pregnant.
- Of course! Doctor has to check on fetus



Composite : A hand from a friend

- Composite pattern allows us to compose objects into tree structure.
- With a little bit of code implementation it is easy to print a list of all livestock including fetus

composite livestocks

```
Public class Livestock extends LivestockComponent{
    String name;
    String category;
    boolean sex;
    int age;
    public Livestock
        (String name,String category,boolean sex, int age){
            //this part is too easy to write
    }
    // getter methods just like our old implementation
    public void print(){
       System.out.printf("%s :%s\n"
            name, category);
    }
```

```
Public class LivestockFetus extends
                                LivestockComponent{
    ArrayList fetusList = new ArrayList();
    // the rest of attribute is the same
    public void add(LivestockComponent mC){
        fetusList.add(mC);
    }
    public void remove(LivestockComponent mC){
        fetusList.remove(mC);
    }
    public LivestockComponent getChild(int index){
        return (LivestockComponent)
            fetusList.get(index);
     public void print(){ // see next page}
    // getter methods just like our old implementation
```

Implementing a print method

- public void print(){
  System.out.printf(“%s :%s\n”
  name, category);

```
Iterator iterator = livestockfetus.iterator();
while(iterator.hasNext()) {
    LivestockComponent livestock = (LivestockComponent) iterator.next();
    livestock.print();
```

}