Iterator Pattern

Michal Moravik, SD20w2

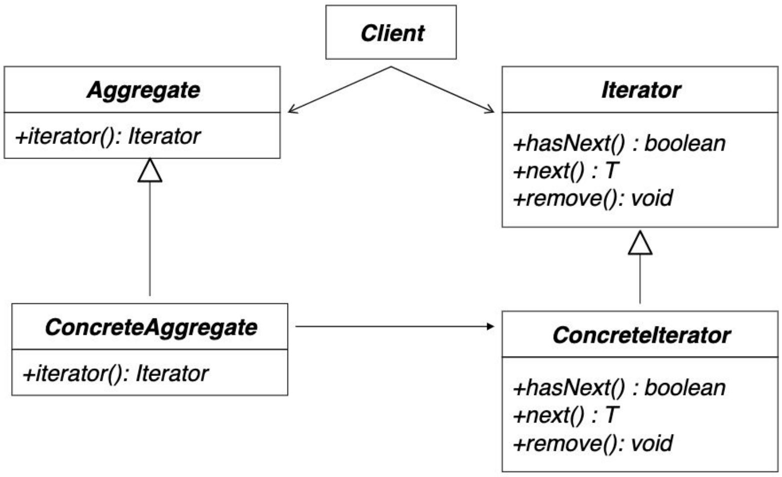
Name: Iterator Pattern

Category: Behavioral

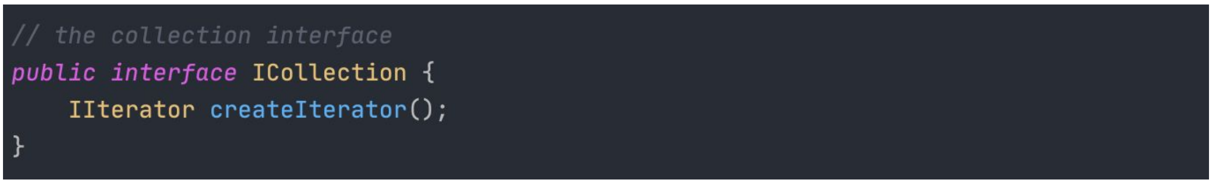
Intent: To build an iterator which goes over a collection of items independent of their type (to build generic and custom iterators).

Motivation: Imagine a situation when we need to iterate over a collection (list, set, trees, …) without knowing what type the collection is. Also, imagine that we want to build our own custom iteration algorithms and then apply them on a collection. That's when iterator comes handy. You can create your custom algorithms for iteration and then apply them on collections of various types.

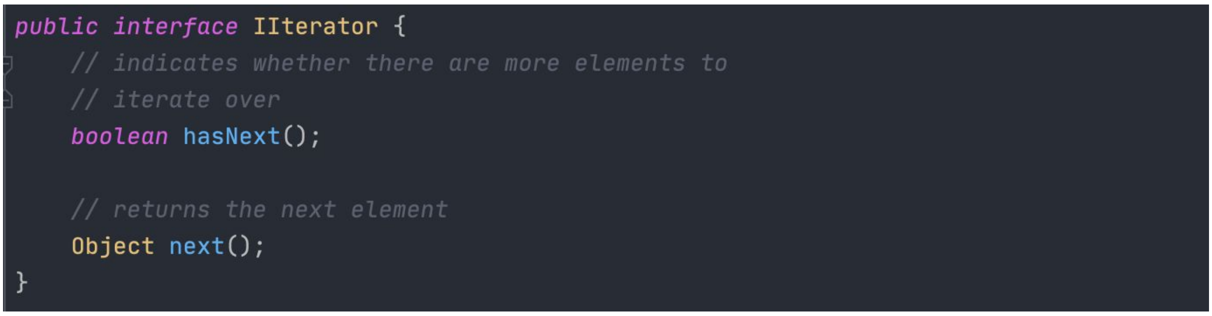
UML:



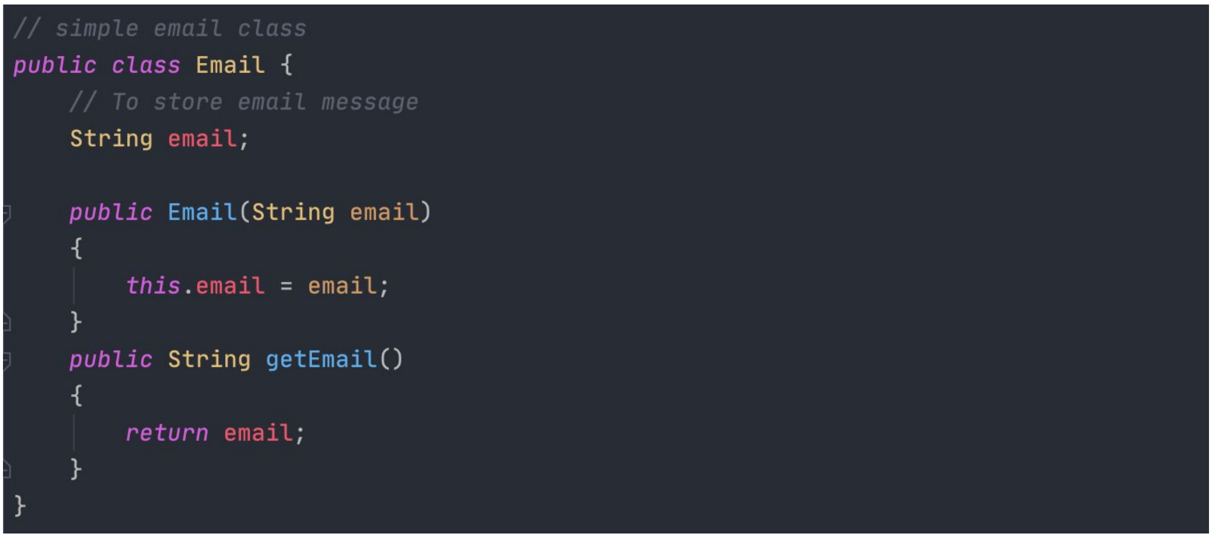
Implementation:



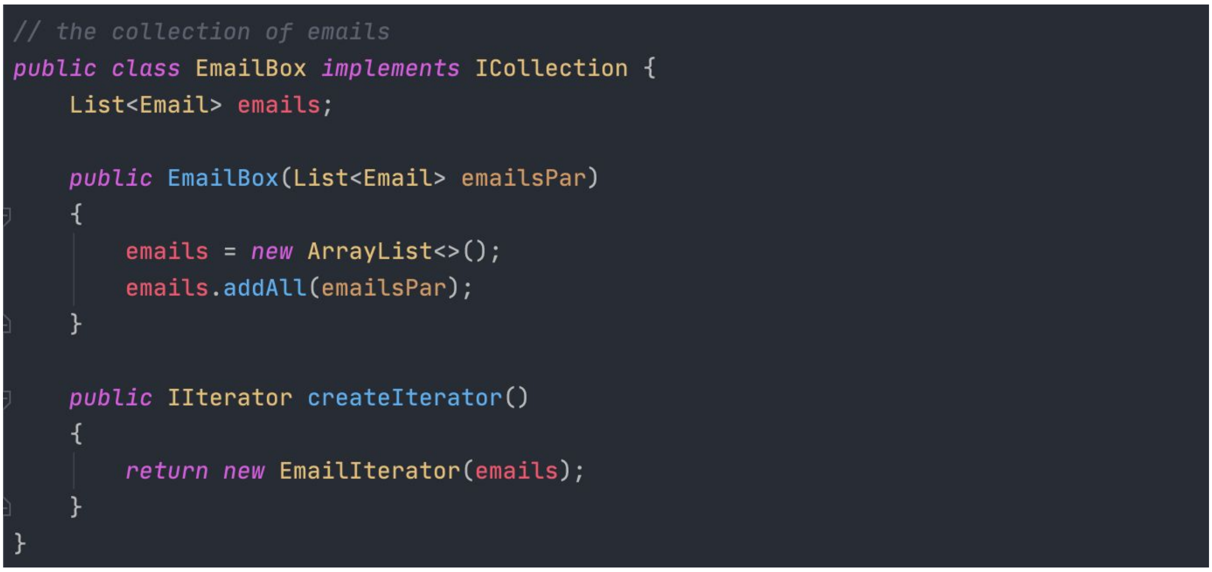
```
// the collection interface
public interface ICollection {
    IIterator createIterator();
}
```



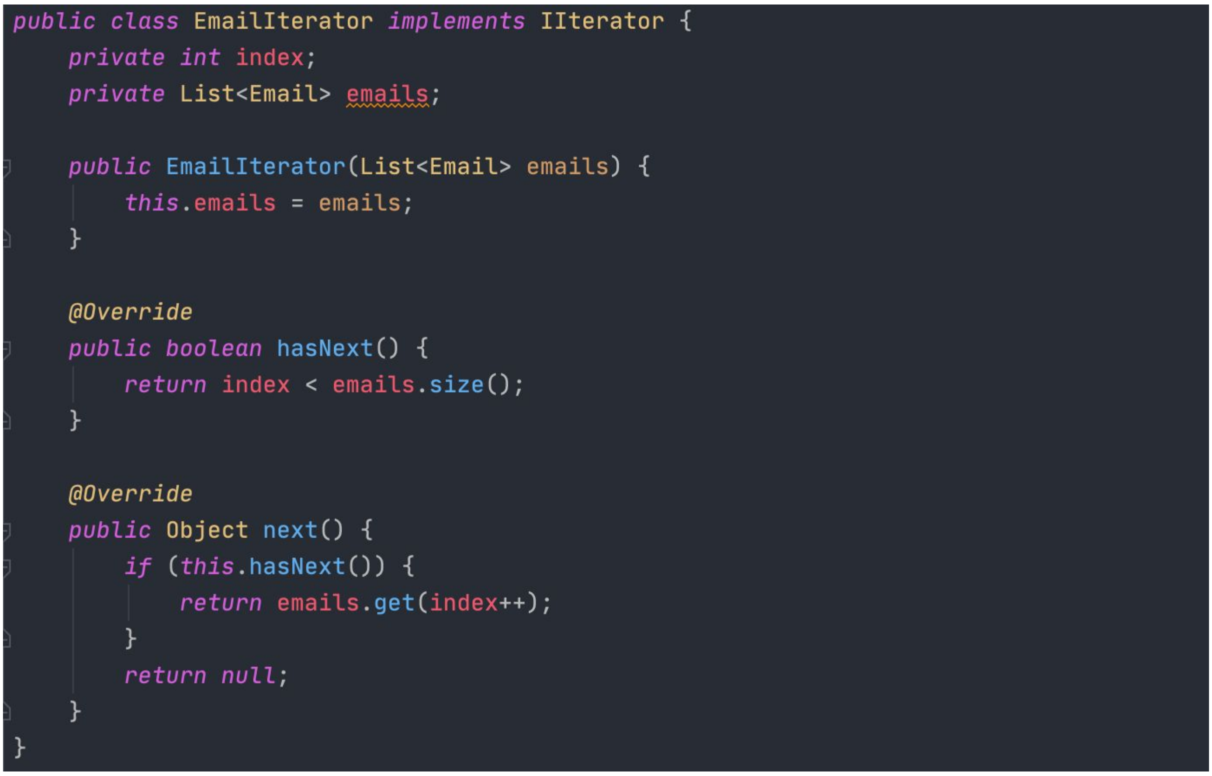
```java
public interface IIterator {
    // indicates whether there are more elements to
    // iterate over
    boolean hasNext();

    // returns the next element
    Object next();
}
```



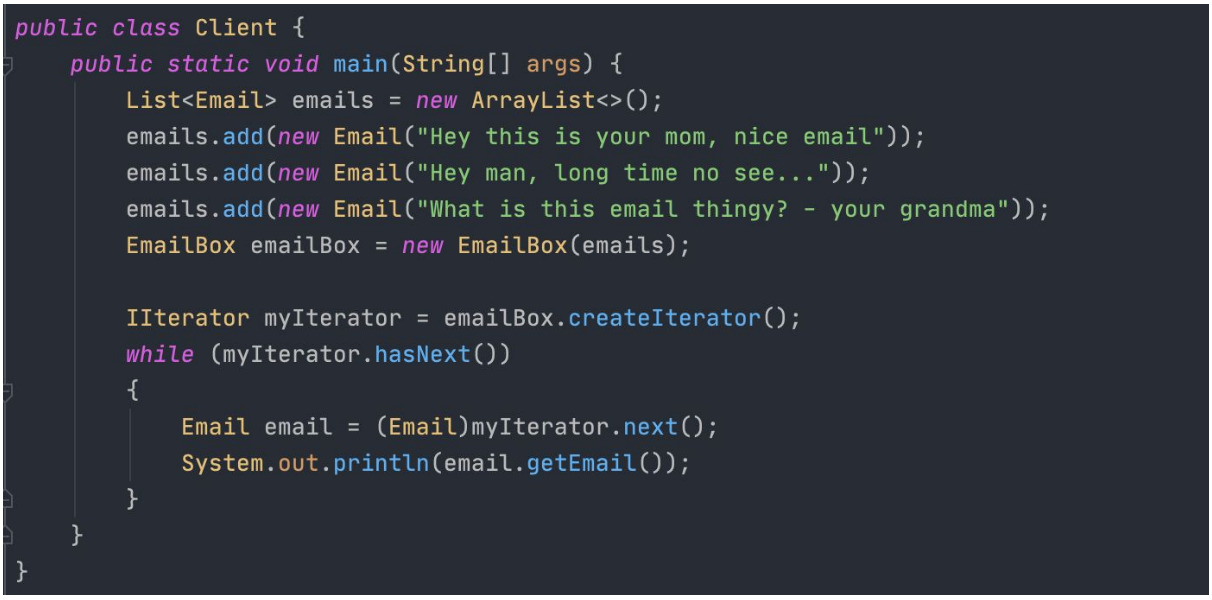
```java
// simple email class
public class Email {
    // To store email message
    String email;

    public Email(String email)
    {
        this.email = email;
    }
    public String getEmail()
    {
        return email;
    }
}
```



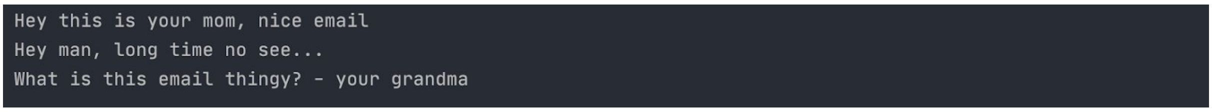
```java
// the collection of emails
public class EmailBox implements ICollection {
    List<Email> emails;

    public EmailBox(List<Email> emailsPar)
    {
        emails = new ArrayList<>();
        emails.addAll(emailsPar);
    }

    public IIterator createIterator()
    {
        return new EmailIterator(emails);
    }
}
```



```java
public class EmailIterator implements IIterator {
    private int index;
    private List<Email> emails;

    public EmailIterator(List<Email> emails) {
        this.emails = emails;
    }

    @Override
    public boolean hasNext() {
        return index < emails.size();
    }

    @Override
    public Object next() {
        if (this.hasNext()) {
            return emails.get(index++);
        }
        return null;
    }
}
```



```java
public class Client {
    public static void main(String[] args) {
        List<Email> emails = new ArrayList<>();
        emails.add(new Email("Hey this is your mom, nice email"));
        emails.add(new Email("Hey man, long time no see..."));
        emails.add(new Email("What is this email thingy? - your grandma"));
        EmailBox emailBox = new EmailBox(emails);

        IIterator myIterator = emailBox.createIterator();
        while (myIterator.hasNext())
        {
            Email email = (Email)myIterator.next();
            System.out.println(email.getEmail());
        }
    }
}
```



```
Hey this is your mom, nice email
Hey man, long time no see...
What is this email thingy? - your grandma
```

Consequences:

Pros:

- Handy to have iterators in separate classes - single responsibility principle
- Can implement new types of collections and iterators and pass them to existing code without breaking anything
- Can iterate over the same collection in parallel because each iterator object contains its own iteration state

- You can delay an iteration and continue it when needed - you have your own iterators, that’s why

Cons:

- Can be an overkill
- Can be less efficient if you are not pro in making perfect iteration algorithms

Known uses:

- Is implemented in Java core libraries - java.util.Iterator, java.util.Enumeration

Related patterns:

- Iterator can traverse Composite Pattern’s trees