COMP2511

☕ 8.2 - Iterator Pattern

In this lecture

Why?

- Understand the concepts of iterators and iterables
- Understand the motivation for the Iterator Pattern
- Discuss implementation of the Iterator Pattern in different languages

How does a for loop actually work?

```
List<String> shoppingList = new ArrayList<String>(
  Arrays.asList(new String[] {
    "apple", "banana", "pineapple", "orange"
}));

for (String item : shoppingList) {
    System.out.println(item);
}
```

Under the hood

```
1 Iterator<String> iter = shoppingList.iterator();
2 while (iter.hasNext()) {
3     String item = iter.next();
4     System.out.println(item);
5 }
```

Iterators

- An **iterator** is an object that enables a programmer to traverse a container
- Allows us to access the contents of a data structure while abstracting away its underlying representation
- In Java, for loops are an abstraction of iterators
- Iterators can tell us:
  - Do we have any elements left?
  - What is the next element?

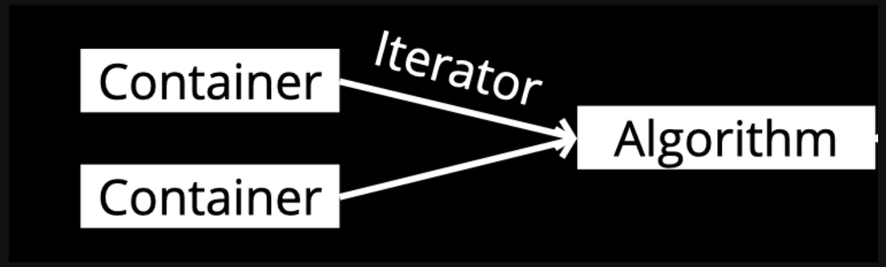
Custom Iterators

Traversing a Data Structure

- Aggregate entities (Containers)
  - Stacks, Queues, Lists, Trees, Graphs, Cycles
- How do we traverse an aggregate entity without exposing its underlying representation?
- Maintain abstraction and encapsulation
- Initial solution - a method in the interface
  - What if we want multiple ways to traverse the container?

Abstracting the Traversal

- Seperate Containers, Iterators and Algorithms
- Allows for many possible ways of traversal
- Avoid bloating interfaces with different traversal methods
- Client (Algorithm) requests an iterator from the container
- Container needs to provide a method for creating an iterator, to show that it is *iterable*



Iterators vs Iterables

- An **iterable** is an object that can be iterated over
- All iterators are iterable, but not all iterables are iterators
- For loops only need to be given something *iterable*

```java
public interface Iterator<E> {
    /**
     * Returns {@code true} if the iteration has more elements.
     * (In other words, returns {@code true} if {@link #next} would
     * return an element rather than throwing an exception.)
     *
     * @return {@code true} if the iteration has more elements
     */
    boolean hasNext();

    /**
     * Returns the next element in the iteration.
     *
     * @return the next element in the iteration
     * @throws NoSuchElementException if the iteration has no more elements
     */
    E next();
```

```java
public interface Iterable<T> {
    /**
     * Returns an iterator over elements of type {@code T}.
     *
     * @return an Iterator.
     */
    Iterator<T> iterator();
```

Example: Custom Iterator

```
Hashtable<String, MenuItem> menuItems =
        new Hashtable<String, MenuItem>();
```

```
public Iterator<MenuItem> createIterator() {
    return menuItems.values().iterator();
}
```

Using or forwarding an **iterator** method from a collection (i.e. Hashtable, ArrayList, etc.)

Implement **Iterator** interface, and provide the required methods (and more if required).

```
public class DinerMenuIterator implements Iterator<MenuItem> {
    MenuItem[] list;
    int position = 0;

    public DinerMenuIterator(MenuItem[] list) {
        this.list = list;
    }

    public MenuItem next() {
        MenuItem menuItem = list[position];
        position = position + 1;
        return menuItem;
    }

    public boolean hasNext() {
        if (position >= list.length || list[position] == null) {
            return false;
        } else {
            return true;
        }
    }

    public void remove() {
        if (position <= 0) {
            throw new IllegalStateException
                ("You can't remove an item until you've done at least one next()");
        }
        if (list[position-1] != null) {
            for (int i = position-1; i < (list.length-1); i++) {
                list[i] = list[i+1];
            }
            list[list.length-1] = null;
        }
    }
}
```

Read the example code discussed/developed in the lectures, and also provided for this week

Iterator Invalidation

- What happens when we modify something we're iterating over?

```
1 numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9]
2
3 for number in numbers:
4     if number == 3 or number == 4:
5         numbers.remove(number)
6
7 print(numbers)
```

Design by Contract

- In many languages, part of the **postconditions** of iterators is that modifying the container in certain ways causes the iterator to become **invalidated** (the behaviour of the iterator is undefined)
  - Python
  - C++

Iterator Invalidation: Java

- What happens when we modify something we're iterating over?

```java
List<Integer> numbers = new ArrayList<Integer>(
  Arrays.asList(new Integer[] {1, 2, 3, 4, 5, 6, 7, 8, 9}
));

for (Integer number : numbers) {
    if (number.equals(3) || number.equals(4)) {
        numbers.remove(number);
    }
}

System.out.println(numbers);
```

Iterator Invalidation: Java

- What happens when we modify something we're iterating over?

```
Exception in thread "main" java.util.ConcurrentModificationException
        at java.base/java.util.ArrayList$Itr.checkForComodification(ArrayLi
        at java.base/java.util.ArrayList$Itr.next(ArrayList.java:997)
        at dungeonmania.DungeonManiaController.main(IterExample.java:120)
```

Generators

- A functional way of writing iterators
- Defined via generator functions instead of classes
- Example generator

```
1 def shopping_list():
2     yield 'apple'
3     yield 'orange'
4     yield 'banana'
5     yield 'pineapple'
6
7 for item in shopping_list():
8     print(item)
```

Iterator Categories (C++)

- Output (Write-only)
- Input (Read-only)
- Forward (most iterators, standard Java iterators)
- Bidirectional (forward and backwards)
- Random Access (iterators which function as arrays)