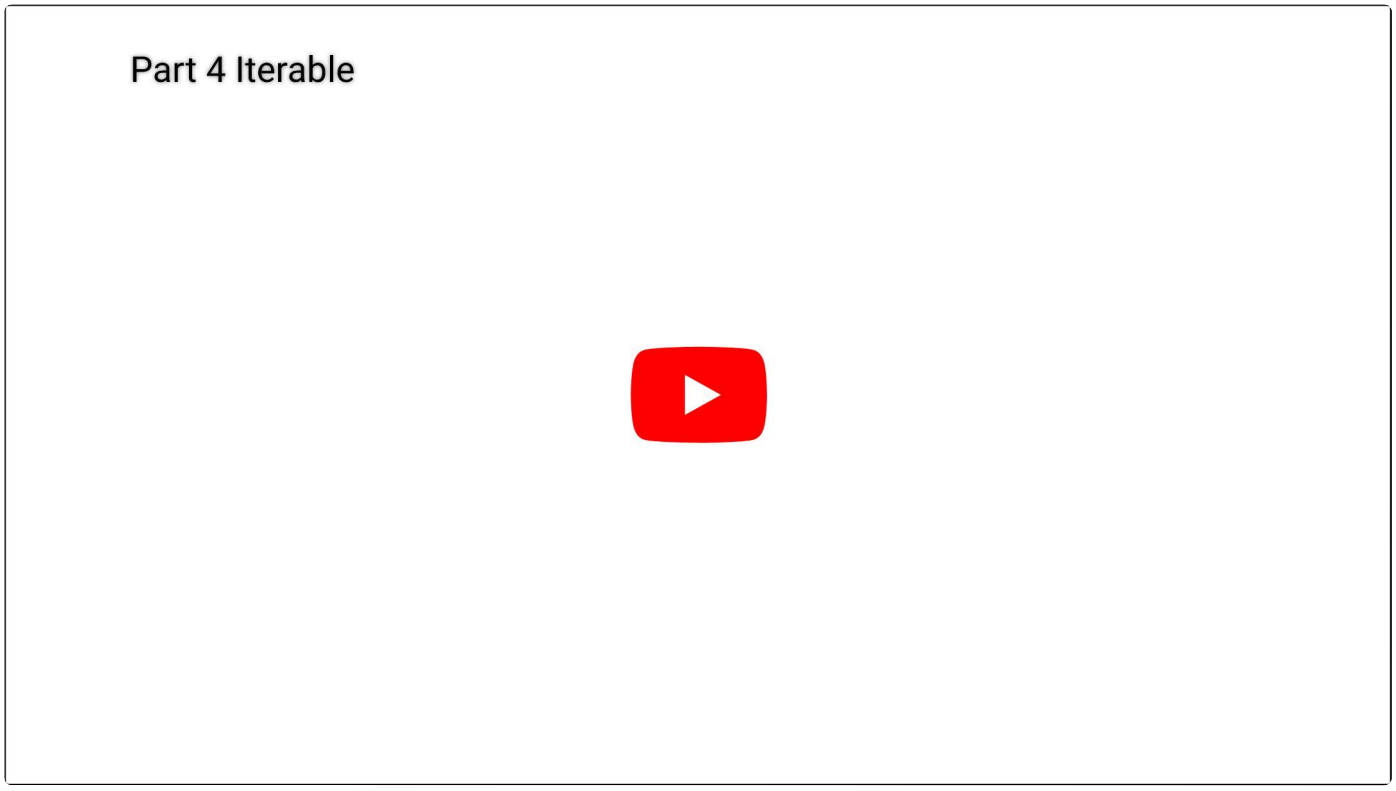
12.3 Iteration



We can use a clean enhanced for loop with Java's `HashSet`

```
Set<String> s = new HashSet<>();
s.add("Tokyo");
s.add("Lagos");
for (String city : s) {
    System.out.println(city);
}
```

However, if we try to do the same with our `ArraySet`, we get an error. How can we enable this functionality?

Enhanced For Loop

Let's first understand what is happening when we use an enhanced for loop. We can "translate" an enhanced for loop into an ugly, manual approach.

```
Set<String> s = new HashSet<>();
...
for (String city : s) {
    ...
}
```

The above code translates to:

```
Set<String> s = new HashSet<>();
...
Iterator<String> seer = s.iterator();
while (seer.hasNext()) {
    String city = seer.next();
    ...
}
```

Let’s strip away the magic so we can build our own classes that support this.

The key here is an object called an *iterator*.

For our example, in List.java we might define an `iterator()` method that returns an iterator object.

```
public Iterator<E> iterator();
```

Now, we can use that object to loop through all the entries in our list:

```
List<Integer> friends = new ArrayList<Integer>();
...
Iterator<Integer> seer = friends.iterator();

while (seer.hasNext()) {
System.out.println(seer.next());
}
```

This code behaves identically to the foreach loop version above.

There are three key methods in our iterator approach:

First, we get a new iterator object with `Iterator<Integer> seer = friends.iterator();`

Next, we loop through the list with our while loop. We check that there are still items left with `seer.hasNext()`, which will return true if there are unseen items remaining, and false if all items have been processed.

Last, `seer.next()` does two things at once. It returns the next element of the list, and here we print it out. It also advances the iterator by one item. In this way, the iterator will only inspect each item once.

Implementing Iterators

In this section, we are going to talk about how to build a class to support iteration.

Let's start by thinking about what the compiler needs to know in order to successfully compile the following iterator example:

```
List<Integer> friends = new ArrayList<Integer>();
Iterator<Integer> seer = friends.iterator();

while(seer.hasNext()) {
    System.out.println(seer.next());
}
```

We can look at the static types of each object that calls a relevant method. `friends` is a List, on which `iterator()` is called, so we must ask:

- Does the List interface have an iterator() method?

`seer` is an Iterator, on which `hasNext()` and `next()` are called, so we must ask:

- Does the Iterator interface have next/hasNext() methods?

So how do we implement these requirements?

The List interface extends the Iterable interface, inheriting the abstract iterator() method. (Actually, List extends Collection which extends Iterable, but it's easier to codethink of this way to start.)

```
public interface Iterable<T> {
    Iterator<T> iterator();
}
```

```
public interface List<T> extends Iterable<T>{
    ...
}
```

Next, the compiler checks that Iterators have `hasNext()` and `next()`. The Iterator interface specifies these abstract methods explicitly:

```
public interface Iterator<T> {
    boolean hasNext();
    T next();
}
```

What if someone calls `next` when `hasNext` returns false?

```
This behavior is undefined. However, a common convention is to throw a `NoSuchEl
```

Will `hasNext` always be called before `next`?

```
Not necessarily. This is sometimes the case when someone using the iterator know
```

Specific classes will implement their own iteration behaviors for the interface methods. Let's look at an example. (Note: if you want to build this up from the start, follow along with the live coding in the video.)

We are going to add iteration through keys to our `ArraySet` class. First, we write a new class called ArraySetIterator, nested inside of ArraySet:

```
private class ArraySetIterator implements Iterator<T> {
    private int wizPos;

    public ArraySetIterator() {
        wizPos = 0;
    }

    public boolean hasNext() {
        return wizPos < size;
    }

    public T next() {
        T returnItem = items[wizPos];
        wizPos += 1;
        return returnItem;
    }
}
```

This ArraySetIterator implements `Iterator<T>`, which means it implements a `hasNext()` method, and a `next()` method, using a `wizPos` position as an index to keep track of its position in the array. For a different data structure, we might implement these two methods differently.

Thought Exercise: How would you design `hasNext()` and `next()` for a linked list?

Now that we have the appropriate methods, we could use a ArraySetIterator to iterate through an `ArraySet`:

```
ArraySet<Integer> aset = new ArraySet<>();
aset.add(5);
aset.add(23);
aset.add(42);

Iterator<Integer> iter = aset.iterator();

while(iter.hasNext()) {
    System.out.println(iter.next());
}
```

We still want to be able to support the enhanced for loop, though, to make our calls cleaner. So, we need to make `ArraySet` implement the Iterable interface. The essential method of the Iterable interface is `iterator()`, which returns an Iterator object for that class. All we have to do is return an instance of our `ArraySetIterator` that we just wrote!

```
public Iterator<T> iterator() {
    return new ArraySetIterator();
}
```

Now we can use enhanced for loops with our `ArrraySet`!

```
ArraySet<Integer> aset = new ArraySet<>();
...
for (int i : aset) {
    System.out.println(i);
}
```

Here we've seen **Iterable**, the interface that makes a class able to be iterated on, and requires the method `iterator()`, which returns an Iterator object. And we've seen **Iterator**, the interface that defines the object with methods to actually do that iteration. You can think of an Iterator as a machine that you put onto an iterable that facilitates the iteration. Any iterable is the object on which the iterator is performing.

With these two components, you can make fancy for loops for your classes!

`ArraySet` code with iteration support is below:

```java
import java.util.Iterator;

public class ArraySet<T> implements Iterable<T> {
    private T[] items;
    private int size; // the next item to be added will be at position size

    public ArraySet() {
        items = (T[]) new Object[100];
        size = 0;
    }

    /* Returns true if this map contains a mapping for the specified key.
     */
    public boolean contains(T x) {
        for (int i = 0; i < size; i += 1) {
            if (items[i].equals(x)) {
                return true;
            }
        }
        return false;
    }

    /* Associates the specified value with the specified key in this map.
       Throws an IllegalArgumentException if the key is null. */
    public void add(T x) {
        if (x == null) {
            throw new IllegalArgumentException("can't add null");
        }
        if (contains(x)) {
            return;
        }
        items[size] = x;
        size += 1;
    }

    /* Returns the number of key-value mappings in this map. */
    public int size() {
        return size;
    }

    /** returns an iterator (a.k.a. seer) into ME */
    public Iterator<T> iterator() {
        return new ArraySetIterator();
    }

    private class ArraySetIterator implements Iterator<T> {
        private int wizPos;

        public ArraySetIterator() {
```

```
        public ArraySetIterator() {
            wizPos = 0;
        }

        public boolean hasNext() {
            return wizPos < size;
        }

        public T next() {
            T returnItem = items[wizPos];
            wizPos += 1;
            return returnItem;
        }
    }

    public static void main(String[] args) {
        ArraySet<Integer> aset = new ArraySet<>();
        aset.add(5);
        aset.add(23);
        aset.add(42);

        //iteration
        for (int i : aset) {
            System.out.println(i);
        }
    }

}
```

Previous
12.2 Exceptions

Next
12.4 Object Methods

Last updated 4 months ago