07 Iterators and Generators

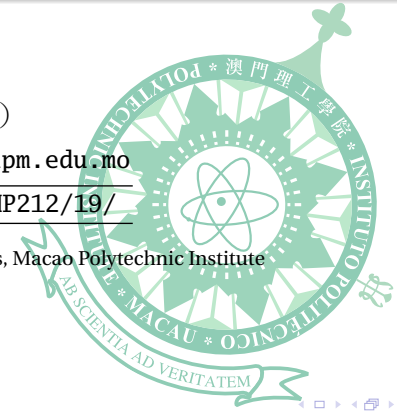
Instructor: Ke Wei（柯韋）

➽ A319 ✆ Ext. 6452 ✉ wke@ipm.edu.mo

http://brouwer.ipm.edu.mo/COMP212/19/

Bachelor of Science in Computing, School of Applied Sciences, Macao Polytechnic Institute

October 10, 2019



Outline

Iterables and Iterators

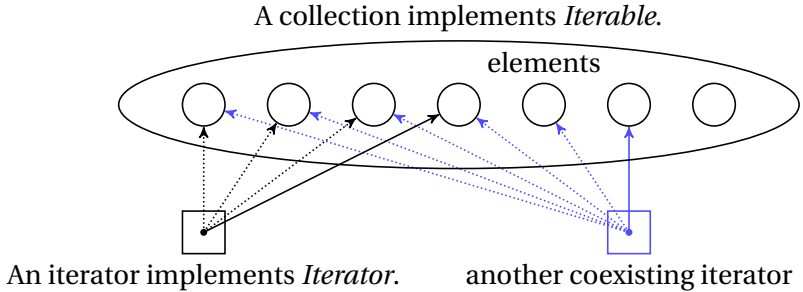
- A common operation of a collection is to enumerate its elements, such as to print all the elements in a linked list.
- In Java (and many other programming languages), such an operation is called *iteration*, which is closely related to the loop statement.
- If a collection is to support Java's standard iteration mechanism, it must implement the *Iterable* interface, in which it returns an instance of the *Iterator* interface (declared in *java.util*).

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

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

Iterables and Iterators (2)

- Method `boolean` *hasNext*() returns true if the iteration has more elements.
- Method *E next*() returns the next element in the iteration.



The "foreach" Control Structure

- Java provides the *foreach* control structure to simplify common iteration calls, by overloading the `for` keyword.
- If c is an instance of a collection class that implements $Iterable\langle E\rangle$, and e is a variable of type E, we can write the following to loop through all the elements in collection c.

```
1 for ( E e : c )  {
2     System.out.println(e.toString());
3 }
```

- And the above construct is equivalent to the following

```
1 for ( Iterator<E> i = c.iterator(); i.hasNext(); ) {
2     E e = i.next();
3     System.out.println(e.toString());
4 }
```

Implementing Iterables

- An iterable can return an instance of an anonymous class which implements the *Iterator*.
- The *ArrayBackward* iterable iterates the elements of an array in reverse order.

```
1  public class ArrayBackward⟨E⟩ implements Iterable⟨E⟩ {
2      private E[] a;
3      public ArrayBackward(E[] a) { this.a = a; }
4      public Iterator⟨E⟩ iterator() {
5          return new Iterator⟨E⟩() {
6              private int i = a.length;
7              public boolean hasNext() { return i > 0; }
8              public E next() { return a[--i]; }
9          };
10     }
11 }
```

Using Iterables and Iterators

- We can now use the *Iterable* and *Iterator* interfaces to write elegant loop constructs. The details of the collections are completely hidden.

```
1 void print(Iterable⟨Integer⟩ ns) {
2     for ( int n : ns )
3         System.out.println(n);
4 }
```

- An instance of *ArrayBackward⟨Integer⟩* can be passed as an instance of *Iterable⟨Integer⟩*.
- We can also use an iterator directly.

```
1 void printArray(Integer[] a) {
2     Iterator⟨Integer⟩ i = new ArrayBackward⟨Integer⟩(a).iterator();
3     while ( i.hasNext() )
4         System.out.println(i.next());
5 }
```

Iterators and Generators

- Iterators are usually used as auxiliary constructs to enumerate elements in collections.
- A collection is often not to be changed during an enumeration, thus many iterators are *read-only* accessors to the collections.
- If we abstract the detail of the underlying collection away, an iterator is merely an object that can return elements one at a time by the *hasNext* and *next* methods.
- We don't see in any way that an underlying collection must be present to implement an iterator.
- An iterator that focuses on generating elements one at a time is sometimes called a generator.

The Empty Iterable

- A trivial yet important iterable is the empty iterable, it stands for the empty collection.
- To implement the empty iterable *Nil*, we simply return `false` in its *hasNext* iterator method.
- *Nil* is the identity element of iterable concatenations.

```
1 public class Nil⟨T⟩ implements Iterable⟨T⟩ {
2     private Iterator⟨T⟩ theItr = new Iterator⟨T⟩() {
3         public boolean hasNext() { return false; }
4         public T next() { throw new NoSuchElementException(); }
5     };
6
7     public Iterator⟨T⟩ iterator() { return theItr; }
8 }
```

A Number Generator: *Range*

- The iterable *Range* is a generator that iterates through a range of integers. The range is specified by a lower bound a (inclusive) and an upper bound b (exclusive).

```
1  public class Range implements Iterable<Integer> {
2      private int a, b, s;
3      public Range(int a, int b) { this(a, b, 1); }
4      public Range(int a, int b, int s) { this.a = a; this.b = b; this.s = s; }
5      public Iterator<Integer> iterator() {
6          return new Iterator<Integer>() {
7              private int i = a;
8              public boolean hasNext() { return i*s < b*s; }
9              public Integer next() { int x = i; i += s; return x; }
10         };
11     }
12 }
```

Using *Range*

An iterable object can have multiple iterator instances to iterate itself simultaneously.

```
1 Range r = new Range(0, 4);
2 for ( int i : r ) {
3     for ( int j : r )
4         System.out.print(String.format("(%d,%d) ", i, j));
5     System.out.println();
6 }
```

```
(0,0) (0,1) (0,2) (0,3)
(1,0) (1,1) (1,2) (1,3)
(2,0) (2,1) (2,2) (2,3)
(3,0) (3,1) (3,2) (3,3)
```

Infinite Sequences

- An infinite sequence contains an infinite number of elements, such as the collection of all integers greater than 10.
- We never see an infinite sequence as a whole. The infinity is represented by on-demand element generation — when we need the next element, it gives us one.
- Infinite sequences can be implemented by iterators that always generate the next element, that is, the *hasNext* constantly returns `true`.

```
1 public abstract class InfItr⟨T⟩ implements Iterator⟨T⟩ {
2     @Override public boolean hasNext() { return true; }
3 }
```

Two Infinite Sequences

```
public class Repeat
implements Iterable⟨Integer⟩ {
    private int x;
    public Repeat(int x) { this.x = x; }
    public Iterator⟨Integer⟩ iterator() {
        return new InfItr⟨Integer⟩() {
            @Override
            public Integer next() {
                return x;
            }
        };
    }
}
```

```
public class From
implements Iterable⟨Integer⟩ {
    private int s;
    public From(int s) { this.s = s; }
    public Iterator⟨Integer⟩ iterator() {
        return new InfItr⟨Integer⟩() {
            private int i = s;
            @Override
            public Integer next() {
                return i++;
            }
        };
    }
}
```

Higher-Order Iterables

- We may define functional iterable objects that operate on other iterables, such as to return a part of an iterable.
- The $take(s, n)$ of an iterable s returns only the first n elements in s as a new iterable.
- The $drop(s, n)$ of an iterable s returns the elements in s except the first n as a new iterable.
- Such iterables that take other iterables as arguments are called *higher-order* iterables.

$$take(drop(from(100), 10), 6) = [110, 111, 112, 113, 114, 115].$$

Take

```
1  public class Take⟨T⟩ implements Iterable⟨T⟩ {
2      private Iterable⟨T⟩ s;
3      private int n;
4      public Take(Iterable⟨T⟩ s, int n) { this.s = s; this.n = n; }
5      public Iterator⟨T⟩ iterator() {
6          return new Iterator⟨T⟩() {
7              private int m = 0;
8              private Iterator⟨T⟩ i = s.iterator();
9              public boolean hasNext() {
10                 return m < n && i.hasNext();
11             }
12             public T next() { ++m; return i.next(); }
13         };
14     }
15 }
```

Drop

```
1  public class Drop⟨T⟩ implements Iterable⟨T⟩ {
2      private Iterable⟨T⟩ s;
3      private int n;
4      public Drop(Iterable⟨T⟩ s, int n) { this.s = s; this.n = n; }
5      public Iterator⟨T⟩ iterator() {
6          return new Iterator⟨T⟩() {
7              private Iterator⟨T⟩ i = s.iterator();
8              { // a constructor of the anonymous class
9                  for ( int m = 0; m < n && i.hasNext(); ++m ) i.next();
10             }
11             public boolean hasNext() { return i.hasNext(); }
12             public T next() { return i.next(); }
13         };
14     }
15 }
```

Generic Methods

- Java also allows a method to take type parameters, for example, an insertion sort method for all comparable element types. Such a method is called a generic method.

```
public static ⟨T extends Comparable⟨T⟩⟩
void insertionSort(T[] a) ...
```

- A generic method has an additional type parameter list just ahead of the return type, such as the ⟨*T* `extends` *Comparable*⟨*T*⟩⟩ in the above code.
- The type parameters can be used in the following method signature and method body, just like normal class types.
- One good thing for generic methods is that the actual type arguments need not to be specified when calling the generic method, the type argument can be inferred from the types of the method arguments or the type of the return result.

```
String[] a = {"Peach", "Apple", "Banana"};
insertionSort(a);
```

Wrapping Functional Generic Classes in Static Methods

- To take the advantage of the type inference of generic methods, we often wrap generic classes for convenience.

```
public static
Iterable<Integer> from(int s) { return new From(s); }

public static <T>
Iterable<T> take(Iterable<T> s, int n) { return new Take<>(s, n); }

public static <T>
Iterable<T> drop(Iterable<T> s, int n) { return new Drop<>(s, n); }
```

- Now we can use these functional classes in an elegent way, that is, as functions.

```
for ( int x : take(drop(from(100), 10), 6) )
    System.out.println(x);
```