Interator and Iterable

Two commonly used interfaces in Java.

And: for-each loop.

Iterator

Problem: how can we access all members of a collection without knowing the structure of the collection?

Solution: each collection (List, Set, Stack, ...) provides an Iterator we can use.

<<interface>>

Iterator

hasNext( ): boolean

next( ): T (Object)

remove(): void

How to Use Iterator

```
List<String> list =
                 new ArrayList<String>( );
list.add(  ...  ); // add stuff

Iterator<String> iter = list.iterator();


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

create a new Iterator for this collection.

Iterator Reduces Dependency

Suppose we have a Purse that contains Coins and a method `getContents` to show what is in the purse:

```
// Suppose a purse has a collection of coins
List<Coin> coins = purse.getContents();
for(int k=0; k<coins.size(); k++) {
   Coin c = coins.get(k);
   //TODO process this coin
}
```

Now the Purse must always create a List for us, even if the coins are stored is some other kind of collection, or a database.

Iterator Reduces Dependency (2)

If `getContents` instead just returns an Iterator, then:

```
// Suppose a purse has a collection of coins
Iterator<Coin> coins = purse.getContents();
while( coins.hasNext() ) {
   Coin c = coins.next();
   //TODO process this coin
}
```

The purse is free to internally use any collection it wants, and does not need to create a List for us.

Iterable

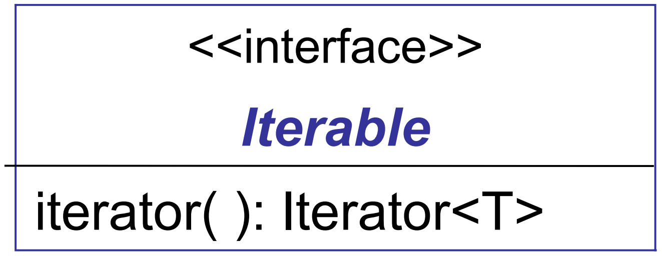
Problem: how can we get an Iterator?

Forces:

(1) the collection should create the iterator itself since only the collection knows its own elements.
(2) every collection should provide same interface for getting an Iterator (for polymorphism).

Solution: define an interface for creating iterators.

Make each collection implement this interface.



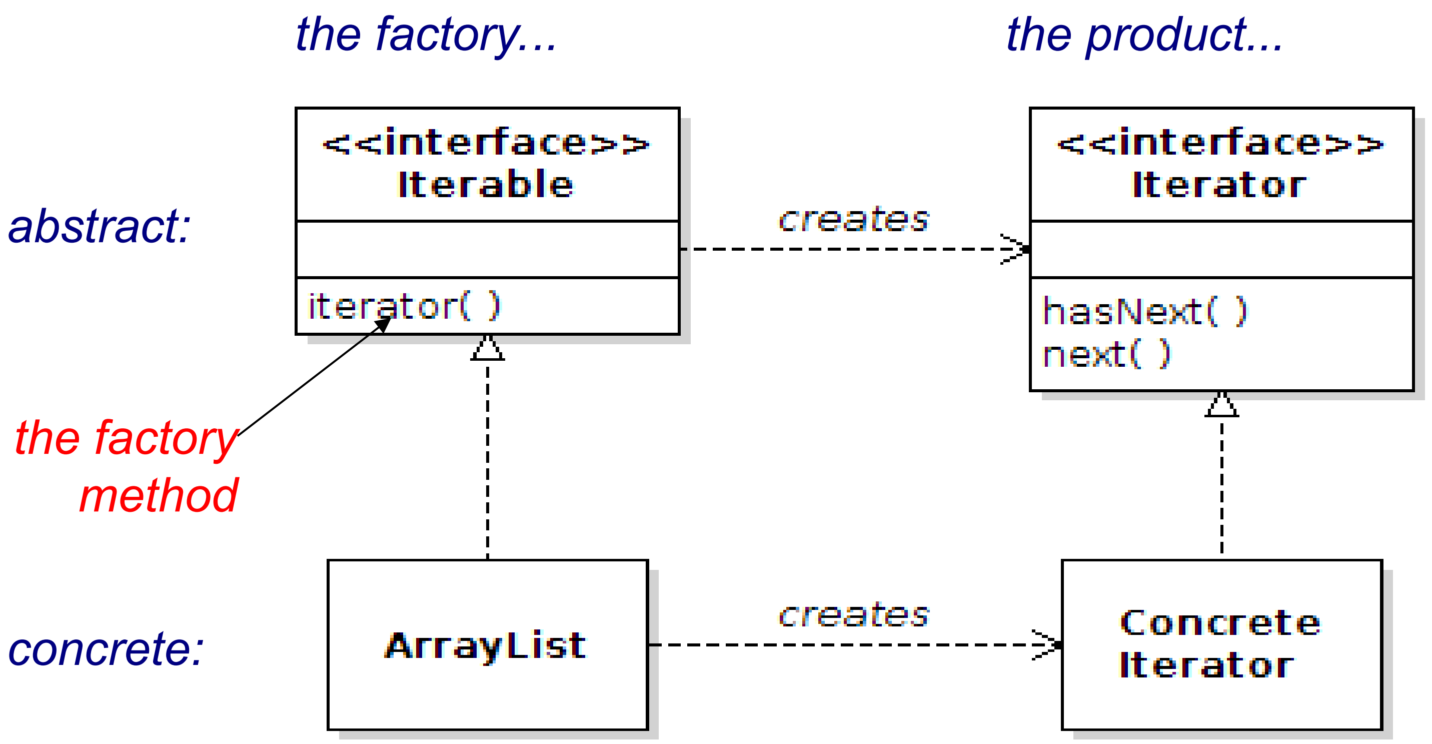
How to Use Iterable

```
List<Student> list =
                    new ArrayList<String>( );
list.add( ... );
list.add( ... );
Iterator<String> iter = list.iterator();
```

`iterator()` creates a new Iterator each time.

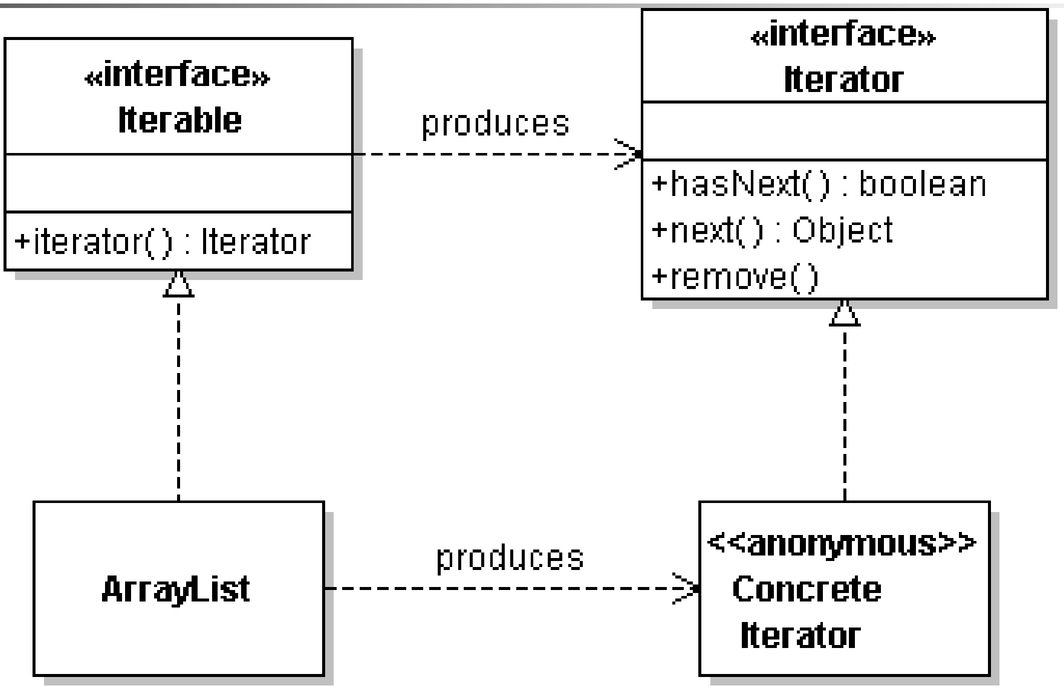
Iterable is a *Factory Method*

You can eliminate direct dependency between classes by creating an interface for required behavior.



Factory Method

The Pattern



Factory Interface	Iterable
Factory Method	iterator( )
Product	Iterator
Concrete Factory	any collection

for-each loop

```
List<String> list =
              new ArrayList<String>( );
list.add( . . . ); // add things to list

// print the list
for( String s: list ) {
    System.out.println(s);
}
```

"for each String s in list" { . . . }

for-each compared to while

For-each loop. **`stuff`** is any Collection or an array.

```
for( Object x: stuff) {
    System.out.println( x );
}
```

While loop does same thing:

```
Iterator iterator = stuff.iterator( );
while( iterator.hasNext() ) {
    Object x = iterator.next( );
    System.out.println( x );
}
```

for-each in detail

"For each Datatype x in ... do { . . . }"

```
for( Datatype x: _collection_ ) {
    System.out.println(x);
}
```

Datatype of the elements in *_collection_*

*_collection_* can be:
1) array
2) any Iterable object

for-each with array

Indexed for loop. **`array[ ]`** is an array of double

```
double [] array = . . .;
for(int k=0; k<array.length; k++) {
    System.out.println(array[k]);
}
```

for-each loop to do the same thing:

```
for( double x: array ) {
    System.out.println( x );
}
```

Error:
modifying a collection while using iterator

Iterator throws an exception if the collection is modified while using an iterator.

```
List<String> words = /* a list of strings */;
Iterator iter = words.iterator( );


System.out.println(iter.next()); // OK
words.add("elephant");
System.out.println(iter.next()); // error
      // exception thrown
```

"for-each" also throws exception if you modify collection while inside loop. (what about for-each with array?)