Interator and Iterable

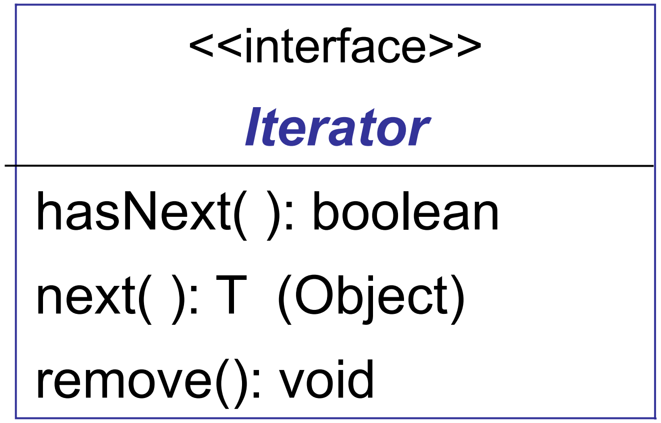
Two commonly used interfaces in Java.

Iterator

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

That is... we want a *polymorphic* way to access a 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( "apple" );
list.add( ... );
Iterator<String> iter = list.iterator();
while ( iter.hasNext() ) {
  String s = iter.next( );
  // do something with s
}
```

`iterator()` creates a new Iterator for the collection.

Iterator Reduces Dependency

Suppose we have a Purse that contains some 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
}
```

But 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 Iterator<Coin> 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 use (internally) any collection it wants.

Another benefit: we can't modify the Iterator.

Iterable

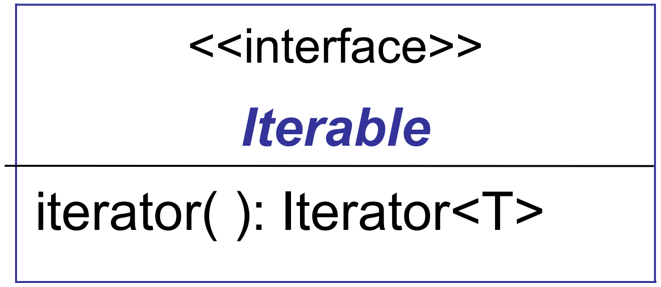
Problem: how can we get an Iterator?

Forces:

(1) the collection should create the iterator itself since only the collection knows its elements.
(2) collections are intended to be polymorphic, so every collection should provide same interface for getting an Iterator.

Solution: define an interface for creating iterators.

Make each collection implement this interface.



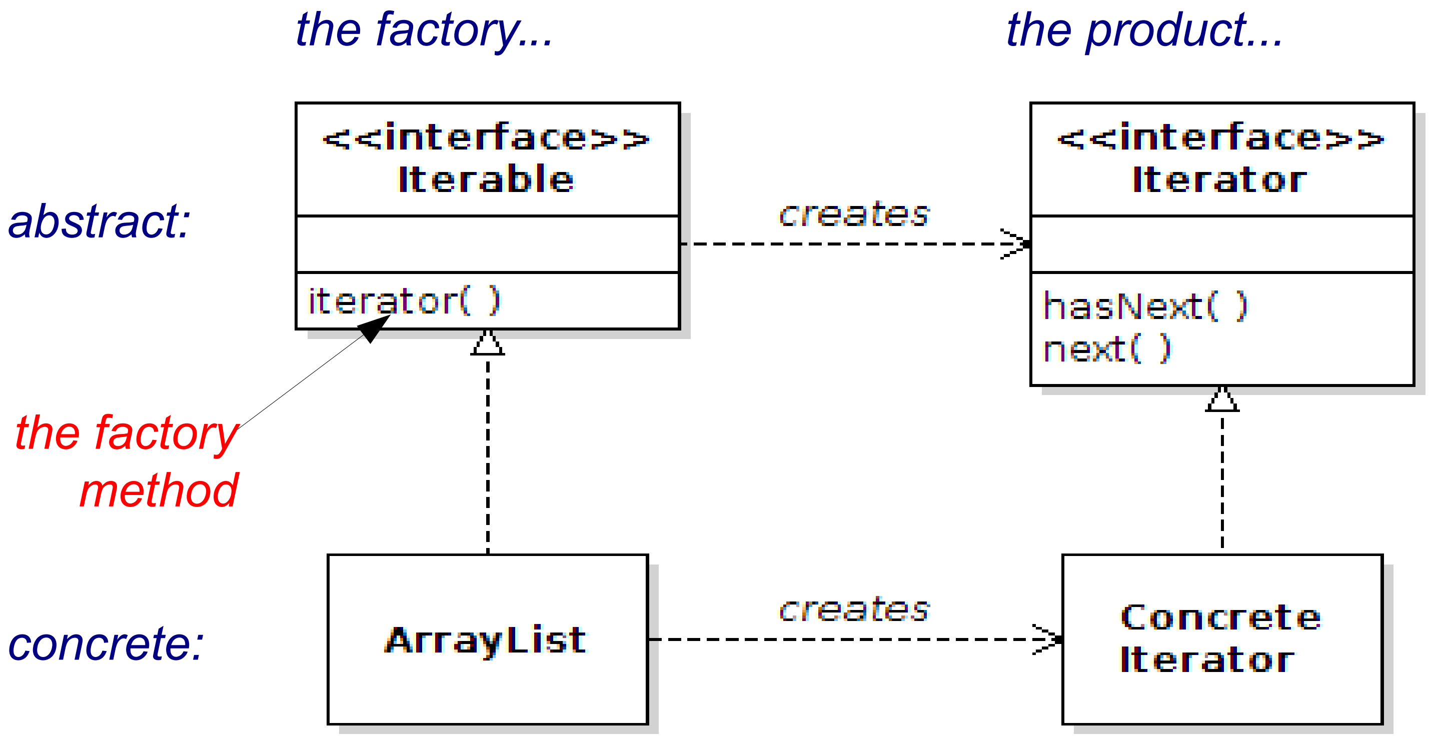
How to Use Iterable

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

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

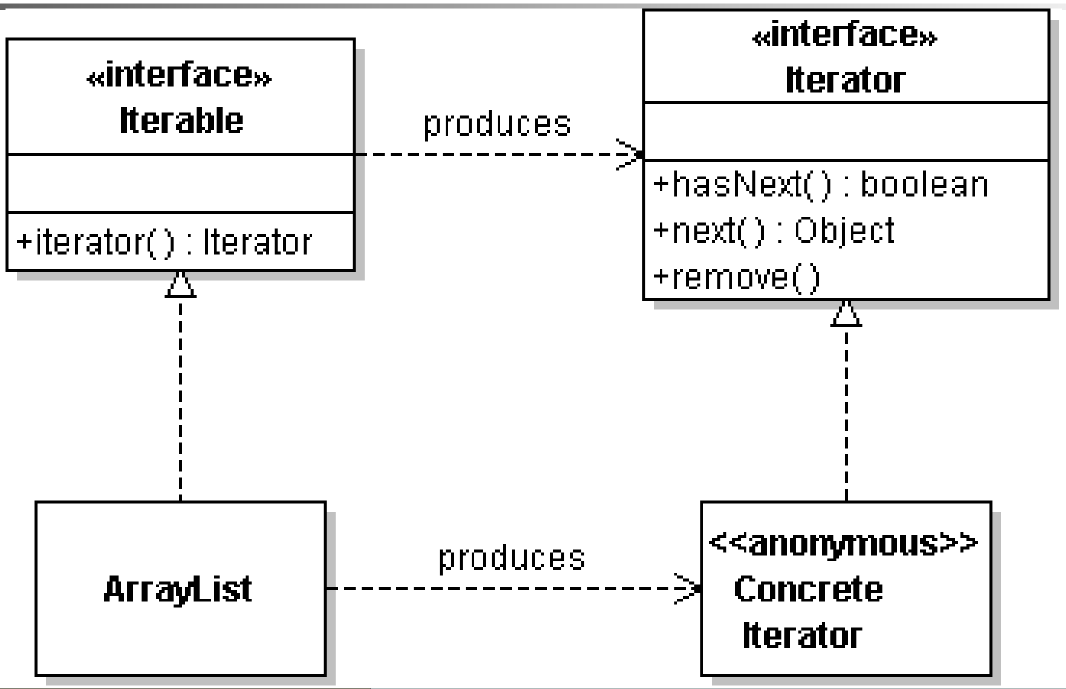
Iterable is a *Factory Method*

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( "apple" );
list.add( ... ); // add more elements

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

"For each String s in list do { . . . }"

for-each compared to while

For any Iterable _stuff_, for-each loop:

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

is the same as this while loop:

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

for-each in detail

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

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

Datatype of the elements in *_stuff_*

*_stuff_* can be: array or *Iterable*

Error:
modifying a collection while using iterator

Suppose: List<String> words = /* list of strings */;

```
Iterator iterator = words.iterator( );
while( iterator.hasNext() ) {
    String x = iterator.next( );
    if (x.isEmpty()) {
      words.remove(x);
      words.add("Empty"); //throws exception
    }
}
```

"for-each" also throws exception if you modify loop target while inside loop.