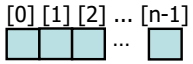
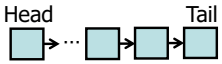
Just in Case:
Major Collection Types in Java

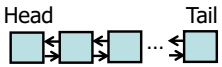
- List
  - Orders elements with their integer index numbers.
  - Offers index-based random access.
  - Can contain duplicate elements.



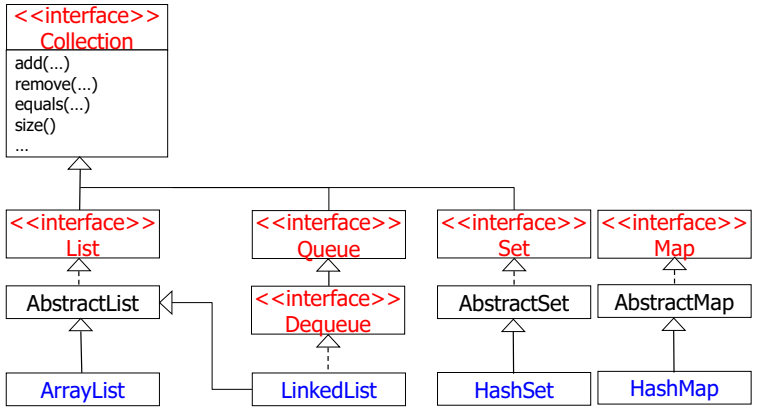
- Queue
  - Orders elements with links.
  - Offers FIFO (First-In-First-Out) access.
  - Can contain duplicate elements.



- Dequeue
  - Stands for "Double Ended QUEUE" (pronounced "deck").
  - Orders elements with links.
  - Offers both FIFO and LIFO (Last-In-First-Out) access.
  - Can contain duplicate elements.

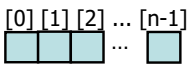


- Set
  - Contains non-duplicate elements without an order.
- Map
  - Contains key-value pairs (w/ non-duplicate keys) without an order.

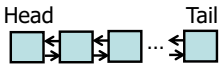


ArrayList v.s. LinkedList

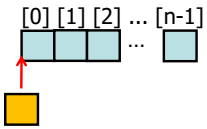
- ArrayList
  - Array-based implementation of the List interface



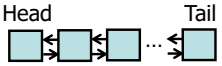
- LinkedList
  - Doubly-linked implementation of the List and Deque interfaces



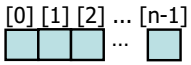
- ArrayList
  - Array-based impl of the List interface
  - Fast index-based access
  - Slow insertion and removal of non-tail elements
    - Fast insertion and removal of the tail element



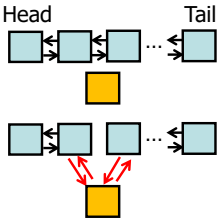
- LinkedList
  - Doubly-linked impl of the List and Deque interfaces



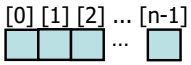
- ArrayList
  - Array-based impl of the List interface
  - Fast index-based access
  - Slow insertion and removal of non-tail elements
    - Fast insertion and removal of the tail element



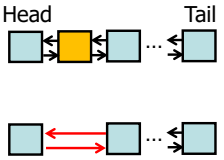
- LinkedList
  - Doubly-linked impl of the List and Deque interfaces
  - Fast insertion and removal of elements



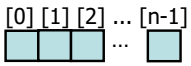
- ArrayList
  - Array-based impl of the List interface
  - Fast index-based access
  - Slow insertion and removal of non-tail elements
    - Fast insertion and removal of the tail element



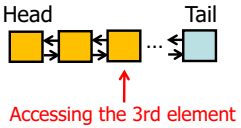
- LinkedList
  - Doubly-linked impl of the List and Deque interfaces
  - Fast insertion and removal of elements



- ArrayList
  - Array-based impl of the List interface
  - Fast index-based access
  - Slow insertion and removal of non-tail elements
    - Fast insertion and removal of the tail element



- LinkedList
  - Doubly-linked impl of the List and Deque interfaces
  - Fast insertion and removal of elements
  - Slow index-based access for “middle” elements.



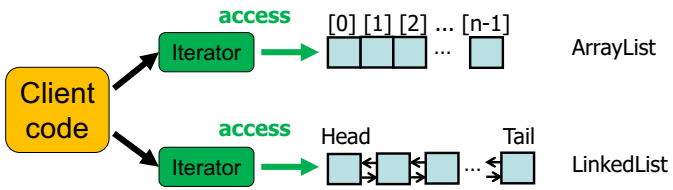
- Use ArrayList
  - If you often need to access “middle” elements.
- Use LinkedList
  - If you often need to insert/remove elements.
- Both yield the same performance for element traversal (i.e. sequential element access).

Iterator Design Pattern

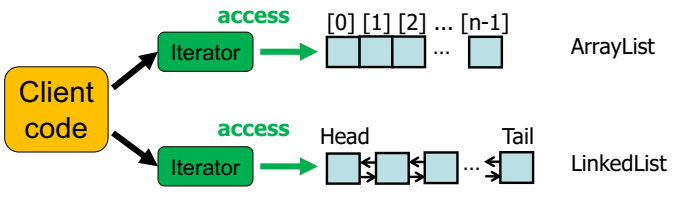
Iterator Design Pattern

- Intent
  - Provides a uniform way to sequentially access collection elements without exposing its underlying representation (i.e. data structure).

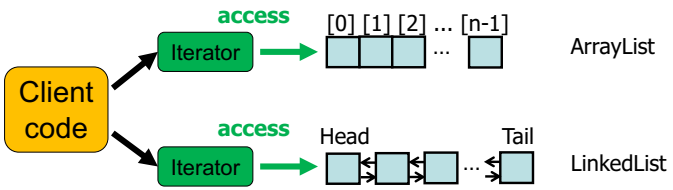
- Provides a uniform way to sequentially access collection elements without exposing its underlying representation (data structure).
  - Offers the same way (i.e., same set of methods) to access different types of collection elements
    - e.g., lists, sets, maps, queues, stacks, trees, graphs…



- Provides a uniform way to sequentially access collection elements without exposing its underlying representation (data structure).
  - Enables to access collection elements one by one

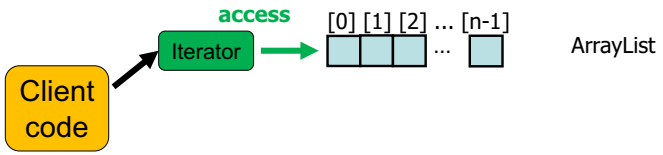


- Provides a uniform way to sequentially access collection elements without exposing its underlying representation (data structure).
  - Abstracts away different access mechanisms for different collection types.
    - Separates (or decouples) a collection's data structure and its access mechanism (i.e., how to get elements)
      - Seeks a loosely-coupled design
    - Hides access mechanisms from collection users (client code)



An Example in Java

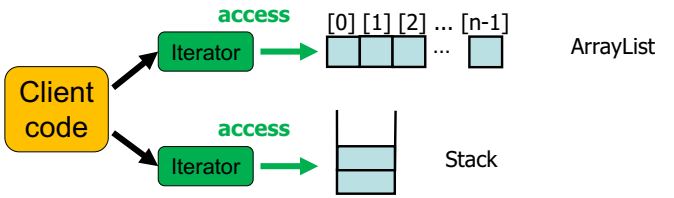
```
ArrayList<Integer> collection = new ArrayList<Integer>();
...
java.util.Iterator<Integer> iterator = collection.iterator();
while ( iterator.hasNext() ) {
     Object o = iterator.next();
     System.out.print( o ); }
```



An Example in Java

```
ArrayList<Integer> collection = new ArrayList<Integer>();
...
java.util.Iterator<Integer> iterator = collection.iterator();
while ( iterator.hasNext() ) {
     Object o = iterator.next();
     System.out.print( o ); }
```

```
Stack<String> collection = new Stack<String>();
...
java.util.Iterator<String> iterator = collection.iterator();
while ( iterator.hasNext() ) {
     Object o = iterator.next();
     System.out.print( o );}
```



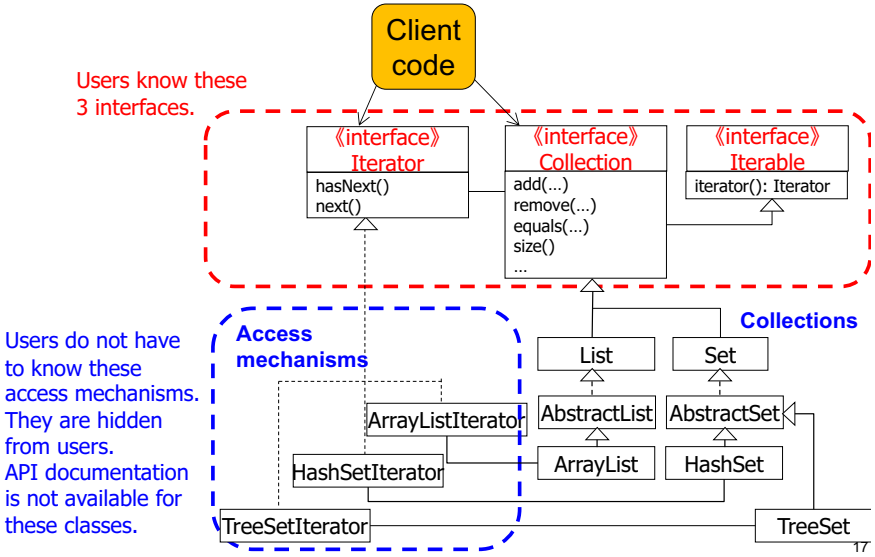
An Example in Java

```
ArrayList<Integer> collection = new ArrayList<Integer>();
...
java.util.Iterator<Integer> iterator = collection.iterator();
while ( iterator.hasNext() ) {
     Object o = iterator.next();
     System.out.print( o ); }
```

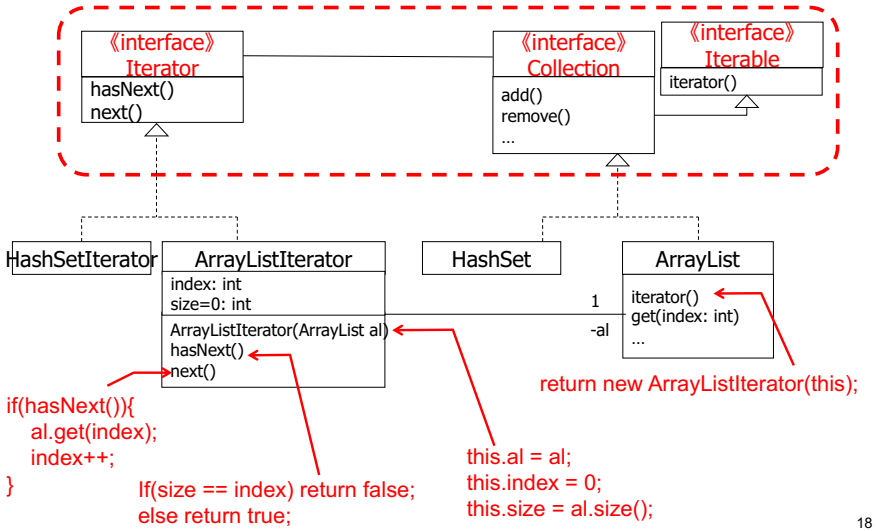
```
Stack<String> collection = new Stack<String>();
...
java.util.Iterator<String> iterator = collection.iterator();
while ( iterator.hasNext() ) {
     Object o = iterator.next();
     System.out.print( o );}
```

- Collection users can enjoy a uniform/same interface (i.e., a set of 3 methods) for different collection types.
  - Users do not have to learn/use different access mechanisms for different collection types.
- Access mechanisms (i.e., how to get collection elements) are hidden by iterators.

Class Structure



What's Hidden from Users?

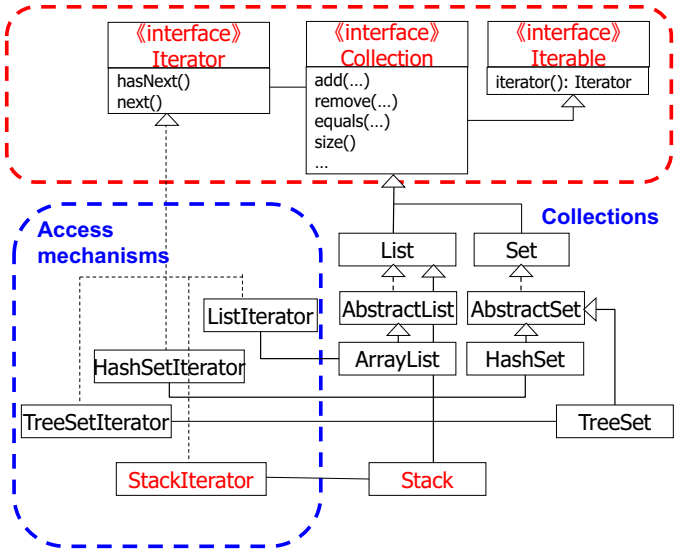


Key Points

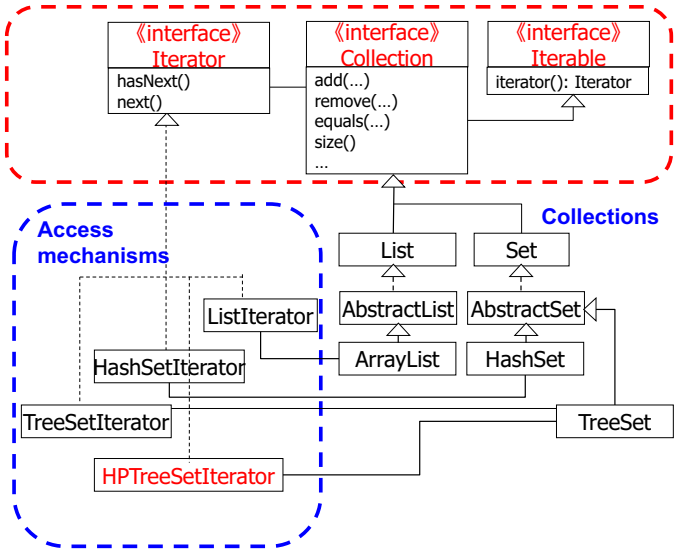
- In user's point of view
  - `java.util.Iterator iterator = collection.iterator();`
  - An iterator always implement the Iterator interface.
  - No need to know what specific implementation class is returned/used.
    - In fact, `ArrayListIterator` does not appear in the Java API documentation.
  - Simple "contract" to know/remember: get an iterator with iterator() and call next() and hasNext() on that.
  - No need to change client code even if
    - Collection classes (e.g., their methods) change.
    - New collection classes are added.
    - Access mechanisms are changed.
- Important principle: ***Program to an interface, not an implementation***

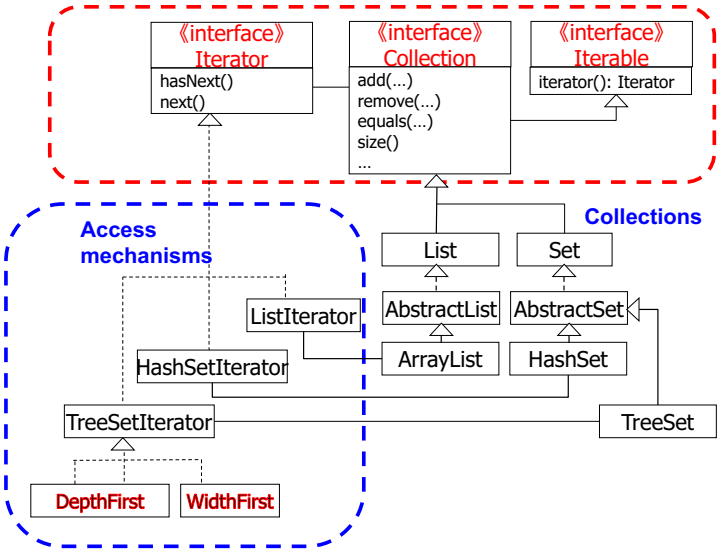
- In collection developer's (API designer's) point of view
  - No need to change
    - `Iterator` and `Iterable` interfaces
    - existing access mechanism classes
  - even if…
    - a new collection class is added.
    - existing collections (their method bodies) need to be modified.
- Important principle: ***Have Your Users Program to an interface, not an implementation***

Adding a New Collection



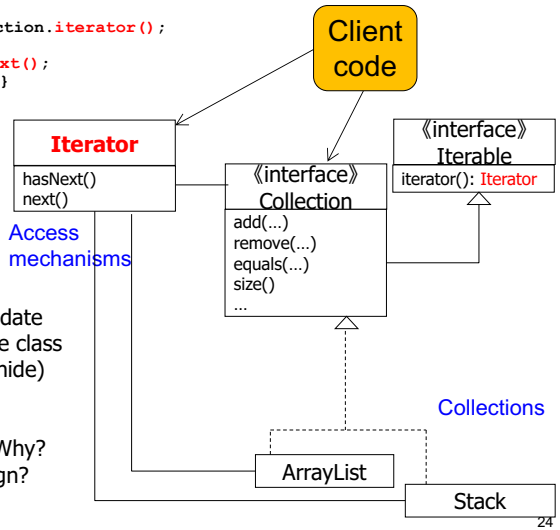
Adding New Access Mechanisms





What's Wrong in this Design?

```
ArrayList<...>(); collection = new ArrayList<...>();
...
Iterator<...> iterator = collection.iterator();
while ( iterator.hasNext() ) {
        Object o = iterator.next();
        System.out.print( o );}
```

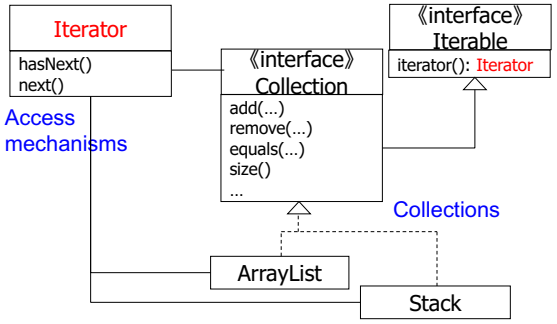


Iterator is defined as a class.

Java API designers could consolidate all access mechanisms in a single class and do not have to define (and hide) multiple iterator classes.

However, they did not do that? Why? Anything bad/wrong in this design?

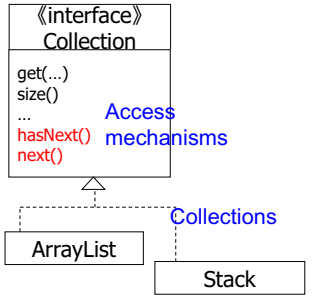
- Iterator becomes error-prone (not that maintainable).
  - Iterator's methods need to have a long sequence of conditional statements.
    - What if a new collection class is added or an existing collection class is modified?
    - What if a collection class's access methods are modified?
- This design is okay for collection users, but not good for collection API designers.
- Several books on design patterns use this design as an example of *Iterator*…



These two designs are same in that both do not decouple collections and access mechanisms.

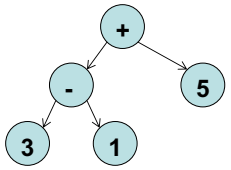
In fact, the right one is better in that it does not have conditional statements in hasNext() and next().

In both designs, you cannot define different iterators in a "pluggable" way.



What Kind of Custom Iterators can be Useful?

- High-performance access to elements
- Secure access to elements
- Get elements from the last one to the first one.
- Get elements at random.
- Sort elements before returning the next element.
  - c.f. `Collections.sort()` and `Comparator`
- "leaf-to-root" width-first policy



By the way… *for-each* Expression

- JDK 1.5 introduced ***for-each*** expressions.

```
ArrayList<String> strList = new ArrayList<String>();
strList.add("a"); strList.add("b");
for(String str: strList){
    System.out.println(str) }
```

  - No need to explicitly use an iterator.
- Note that "for-each" is a *syntactic sugar* for iterator-based code.
  - The above code is automatically transformed to the following code during a compilation:

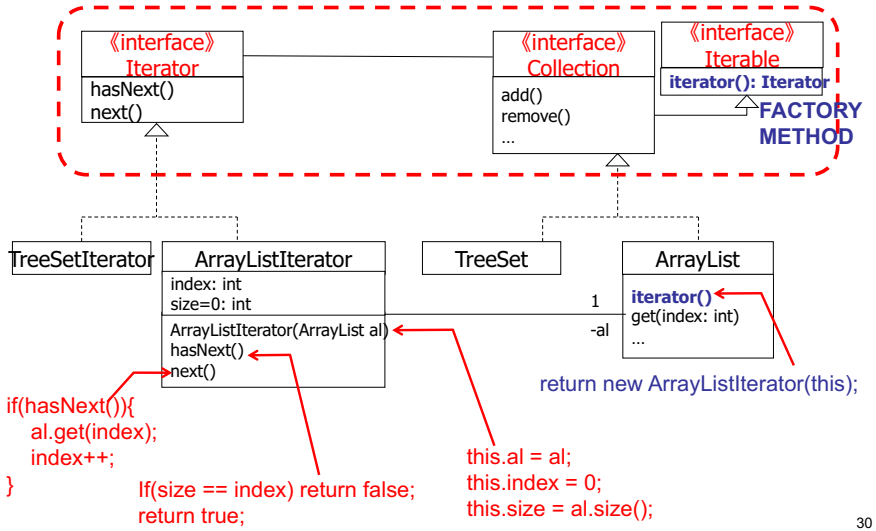
```
for(Iterator itr=strList.iterator(); itr.hasNext();){
    String str = strList.next();
    System.out.println(str)) }
```

Recap

- ```
  Stack<String> collection = new Stack<String>();
  ...
  java.util.Iterator<String> iterator = collection.iterator();
    // Get an iterator.
    // Iterator is an interface. Can't get its instance by "new" it.
  while ( iterator.hasNext() ) {
       Object o = iterator.next();
       System.out.print( o );}
  ```

- ```
  ArrayList<Integer> collection = new ArrayList<Integer>();
  ...
  java.util.Iterator<Integer> iterator = collection.iterator();
  while ( iterator.hasNext() ) {
       Object o = iterator.next();
       System.out.print( o ); }
  ```


iterator() is a Factory Method



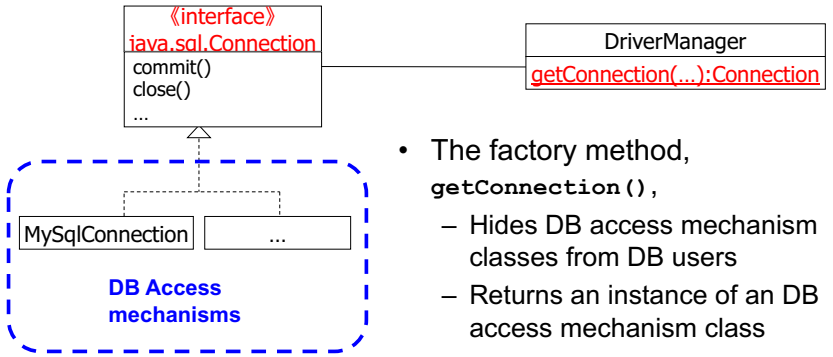
What's the Point?

- The factory method, `iterator()`,
  - Hides access mechanism classes from collection users
  - Returns an instance of an access mechanism class

A Similar Example:

`DriverManager.getConnection()` in JDBC API

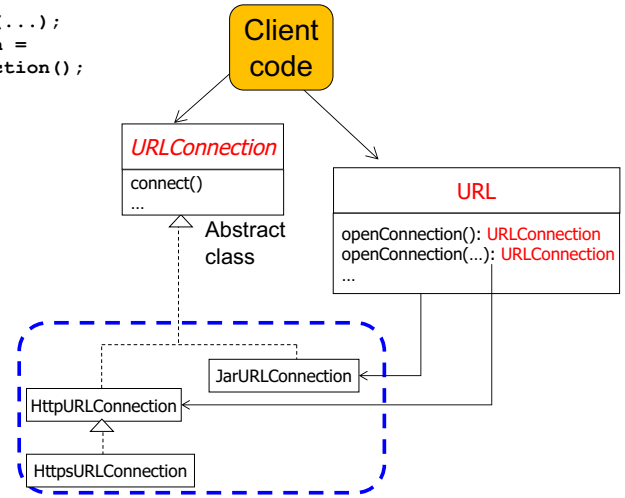
- ```
  Connection conn =
    DriverManager.getConnection("jdbc:mysql://localhost:3306/mydb",
                                "user-name", "password");
  conn.commit(...);
  ```



- The factory method, `getConnection()`,
  - Hides DB access mechanism classes from DB users
  - Returns an instance of an DB access mechanism class
  ```

Another Similar Example:
`URL` and `URLConnection` in Java API

- ```
  URL url = new URL(...);
  URLConnection conn =
      url.openConnection();
  conn.connect(...);
  ```


  ```