# Iterator Design Pattern

# Iterator Design Pattern

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

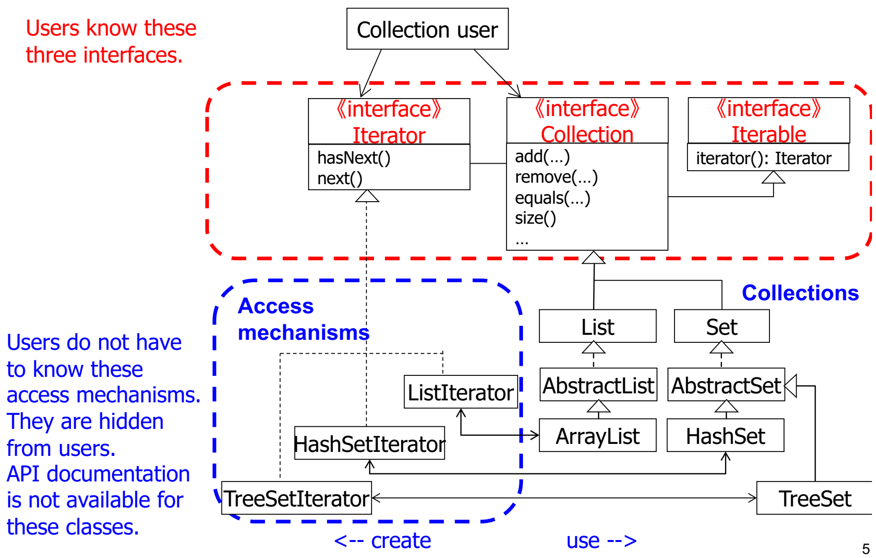
- Provides a uniform way to sequentially access/traverse the elements in a collection without exposing its underlying representation (data structure).
  - Same way (i.e., same interface/method) to access/traverse different types of collections
    - e.g., lists, sets, bags, trees, graphs, tables, queues, stacks…
  - Access/traverse collection elements one by one
  - Abstract away different access mechanisms for different collection types.
    - Separate (or decouple) a collection's data structure and its access mechanisms (i.e., how to get collection elements)
      - » Loosely-coupled design
    - Hide access mechanisms from collection users

# An Example in Java

-
  ```
  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 ); }
  ```
- Collection users can enjoy a uniform/same interface (i.e., a set of 3 methods) for different collection types.
  - There are so many collection types in Java.
  - 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



Users know these three interfaces.

Collection user

《interface》 Iterator
hasNext()
next()

《interface》 Collection
add(...)
remove(...)
equals(...)
size()
...

《interface》 Iterable
iterator(): Iterator

**Access mechanisms**

**Collections**

Users do not have to know these access mechanisms. They are hidden from users.
API documentation is not available for these classes.

ListIterator

HashSetIterator

TreeSetIterator

List

Set

AbstractList

AbstractSet

ArrayList

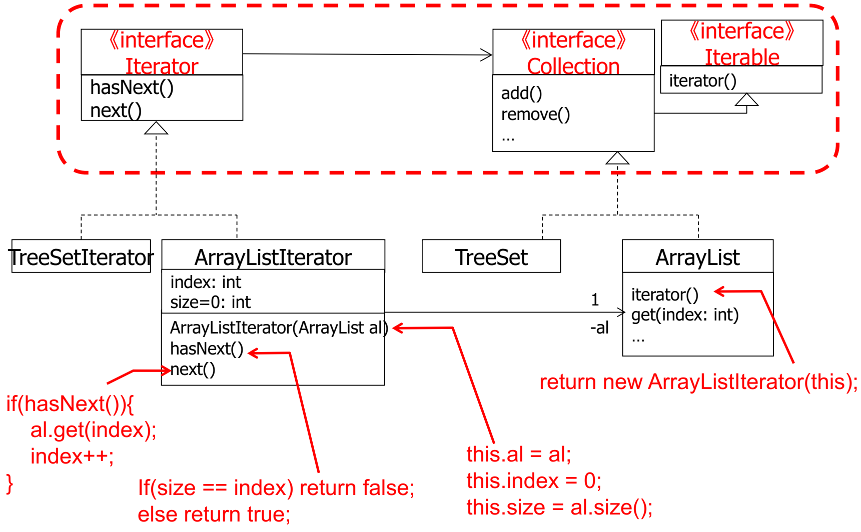
HashSet

TreeSet

<-- create

use -->

# What's Hidden from Users?



《interface》 Iterator
hasNext()
next()

《interface》 Collection
add()
remove()
...

《interface》 Iterable
iterator()

TreeSetIterator

ArrayListIterator
index: int
size=0: int
ArrayListIterator(ArrayList al)
hasNext()
next()

TreeSet

ArrayList
iterator()
get(index: int)
...

1

-al

return new ArrayListIterator(this);

```
if(hasNext()){
    al.get(index);
    index++;
}
```

If(size == index) return false;
else return true;

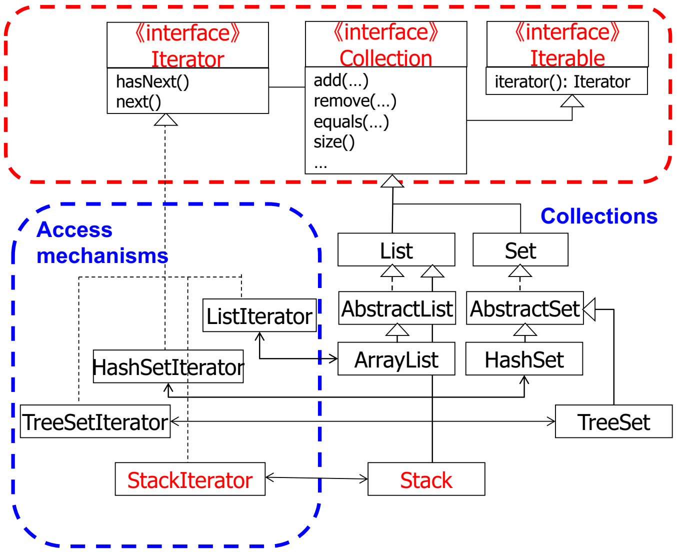
this.al = al;
this.index = 0;
this.size = al.size();

# Key Points

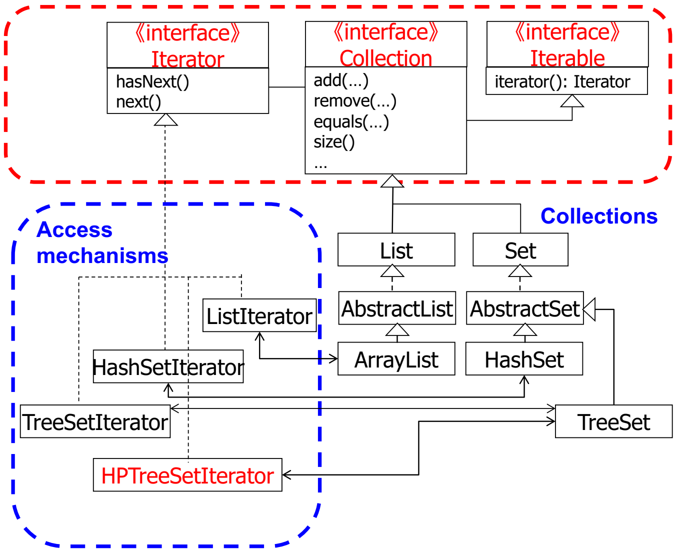
- In user's point of view
  - java.util.**Iterator** iterator = collection.**iterator()**;
  - An iterator always implement the Iterator interface.
  - No need to know what specific implementation class is returned/used.
    - In fact, ArrayListIterator does not appear in the Java API documentation.
  - Simple "contract" to know/remember: get an iterator with iterator() and call next() and hasNext() on that.
  - No need to change user 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***

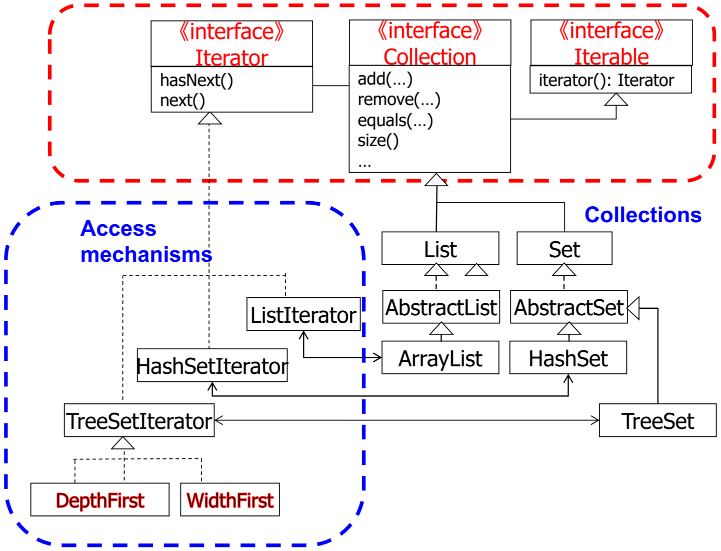
- In collection developer's (API designer's) point of view
  - No need to change
    - Iterator and Iterable interfaces
    - existing access mechanism classes
  - even if…
    - a new collection class is added.
    - existing collections (their method bodies) need to be modified.
- Important principle: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

# Adding a New Collection (Stack)



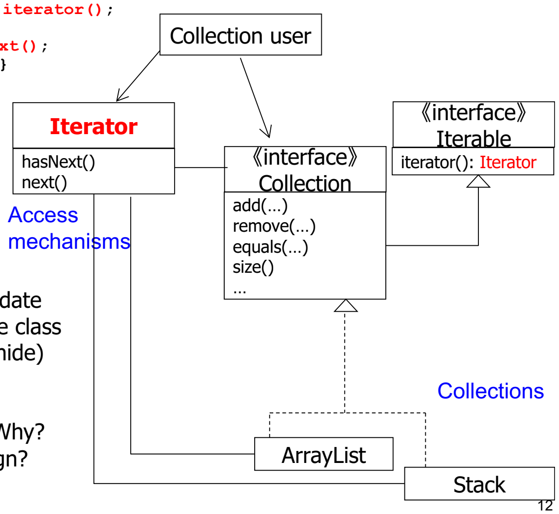
# Adding New Access Mechanisms





# 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 );}
```

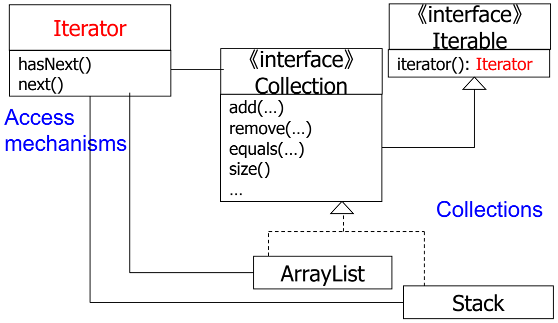


Iterator is defined as a class.

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*…



Iterator

hasNext()
next()

Access mechanisms

《interface》
Collection

add(…)
remove(…)
equals(…)
size()
…

《interface》
Iterable

iterator(): Iterator

Collections

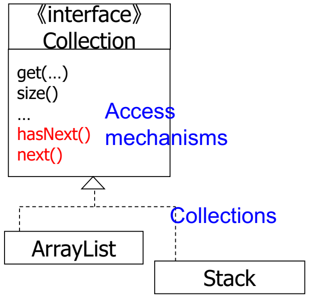
ArrayList

Stack

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.



《interface》
Collection

get(…)
size()
…
hasNext()
next()

Access mechanisms

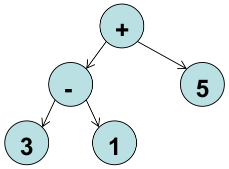
Collections

ArrayList

Stack

## What Kind of Custom Iterators can be Useful?

- Get elements from the last one toward the first one.
- Get elements at random.
- Implement next() and previous()
- 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:

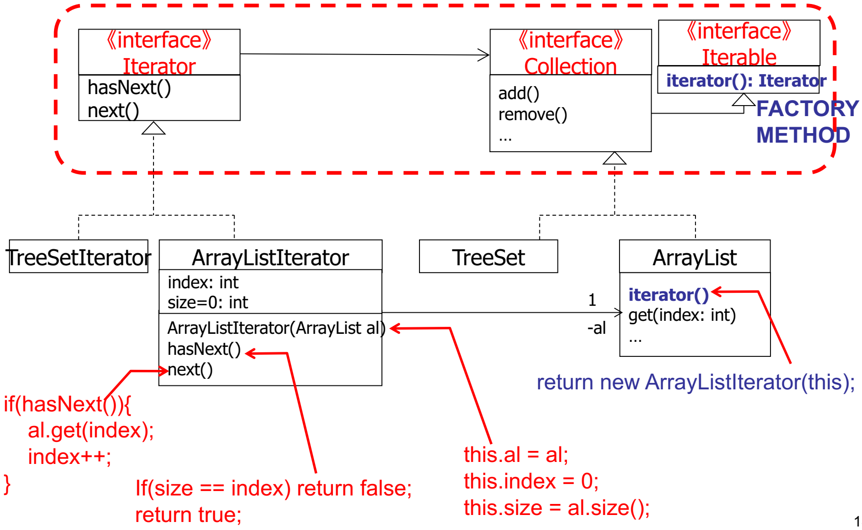
```
– for(Iterator itr=strList.iterator(); itr.hasNext();){
      String str = strList.next();
      System.out.println(str)) }
```

# 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



```
if(hasNext()){
   al.get(index);
   index++;
}
```

```
If(size == index) return false;
return true;
```

```
this.al = al;
this.index = 0;
this.size = al.size();
```

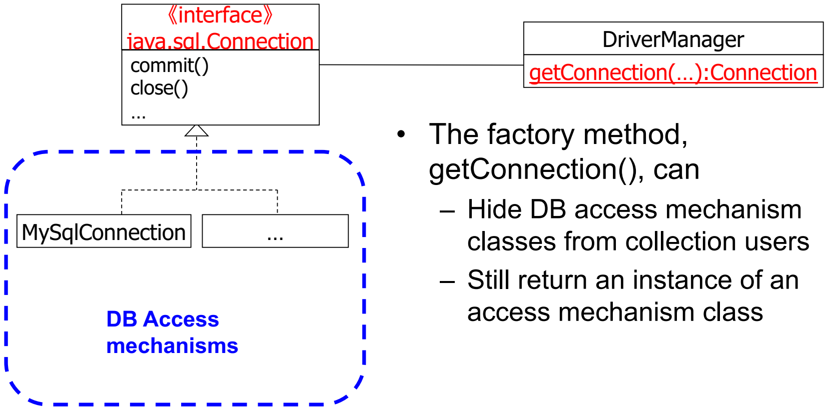
```
return new ArrayListIterator(this);
```

# What's the Point?

- The factory method, iterator(), can
  - Hide access mechanism classes from collection users
  - Still return an instance of an access mechanism class

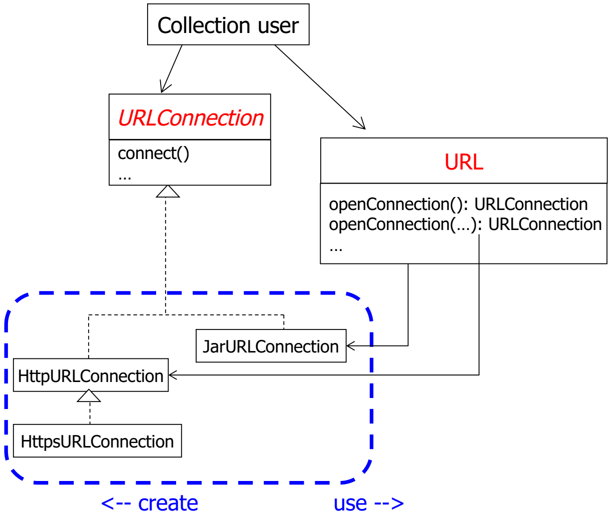
# A Similar Example: DriverManager.getConnection() in JDBC API

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



- The factory method, getConnection(), can
  - Hide DB access mechanism classes from collection users
  - Still return an instance of an access mechanism class

# Another Example:
# URL and URLConnection in Java API


Collection user
URLConnection
connect()
…
URL
openConnection(): URLConnection
openConnection(…): URLConnection
…
JarURLConnection
HttpURLConnection
HttpsURLConnection
<-- create
use -->