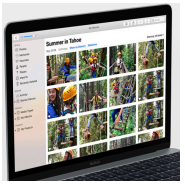
Recap: *Proxy* Design Pattern

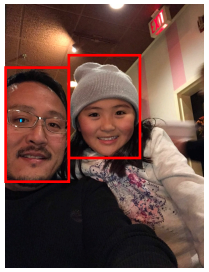
- Intent
  - Provide a surrogate or placeholder for another object to control access to it.

Face Detection in Pictures

- Suppose you are implementing an app to organize, edit and analyze pictures.
  - e.g., Photos from Apple



  - The app loads each raw picture and then superimpose a rectangle on a human face by (dynamically) calling an external face detection/recognition API.
    - e.g., Microsoft Azure Face API, Google Cloud Vision API

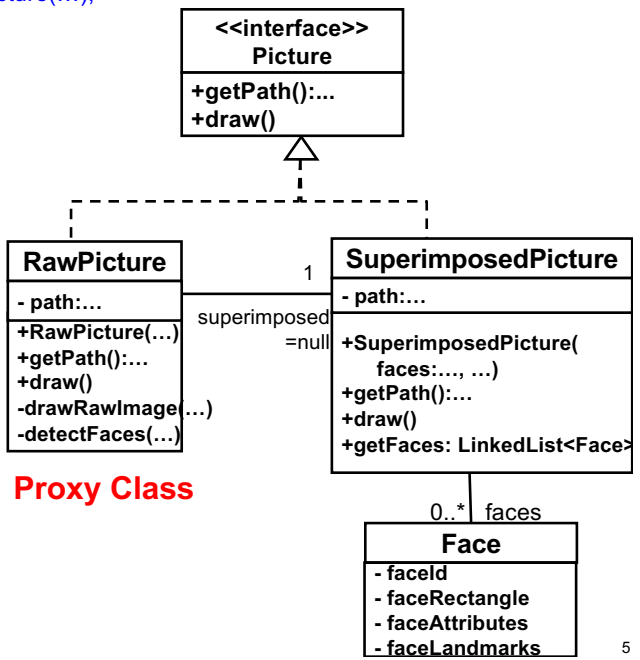


- Some delay is expected to receive a face detection result from an external API.
  - The user is not patient enough to keep watching a blank app window until receiving a detection result.

- Lazy loading of detection results
  - Show the user a raw picture first.
  - Call a face detection API.
  - Receive a detection result.
  - Replace the raw picture with a superimposed one, which contains a detection result.

Client code (app):

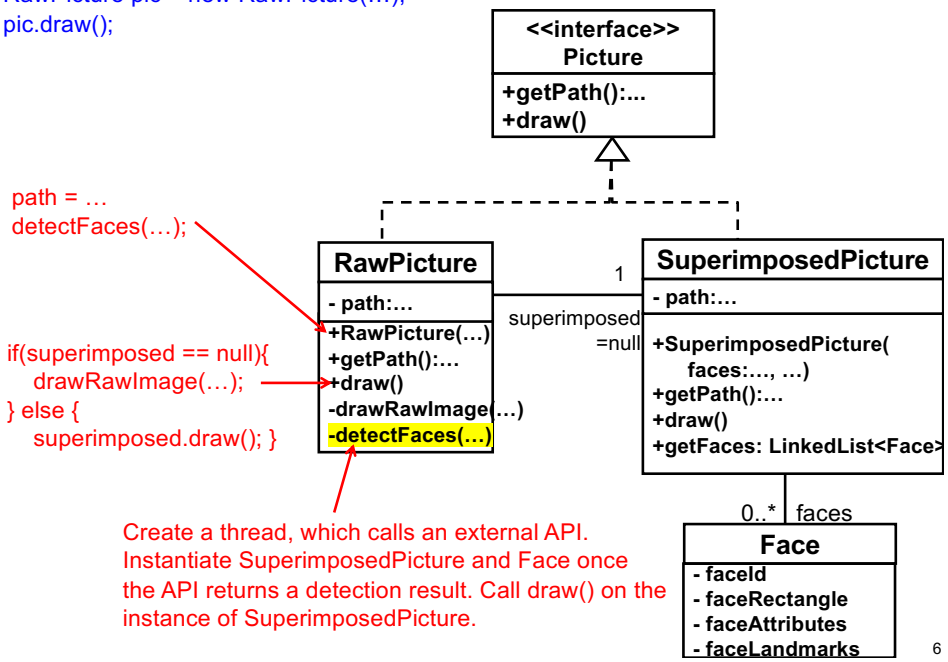
```
RawPicture pic = new RawPicture(…);
pic.draw();
```



Proxy Class

Client code (app):

```
RawPicture pic = new RawPicture(…);
pic.draw();
```



An Example Detection Result

```
Detection result:
JSON:
 [
  {
    "faceId": "f1720101-......",
    "faceRectangle": {
      "top": 128,
      "left": 459,
      "width": 224,
      "height": 224
    },
    "faceAttributes": {
      "smile": 1.0,
      "gender": "female",
      "age": 24.0,
      "emotion": {
        "anger": 0.0,
        "happiness": 1.0,
        "sadness": 0.0,
        "surprise": 0.0
      }
    },
    "faceLandmarks": {
      "pupilLeft": {
        "x": 504.8,
        "y": 206.8
      },
      "pupilRight": {
        "x": 602.5,
        "y": 178.4
      }
    }
  }
]
```

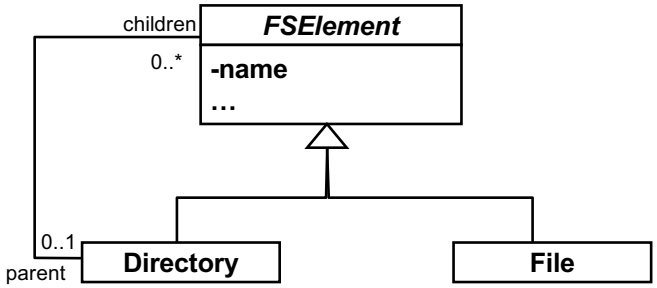
Separation of Concerns

- Lazy loading of face detection results
  - How to display raw pictures
  - How to call an external API and receive a detection result
- Rendering of superimposed pictures
  - How to show face contours
  - What other detection results to display
    - e.g., age, gender, pupil locations

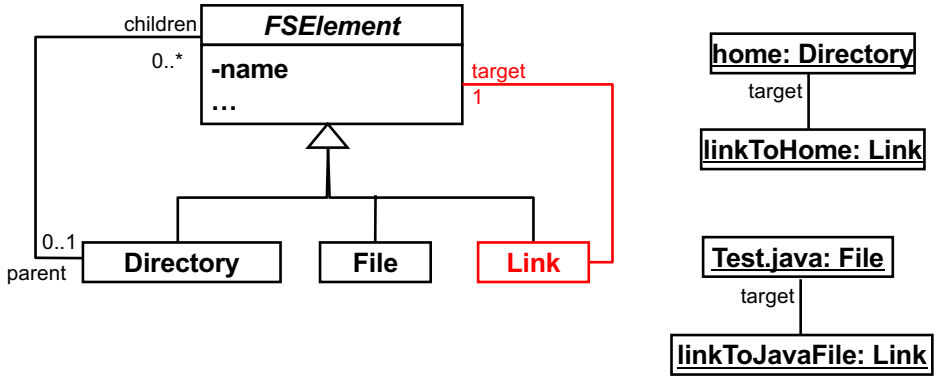
Further Potential Improvements

- Lazy loading of detection results is tightly coupled with client
  - You can consider further design extensions that we saw in the previous example.
    - Static factory method(s) in Picture.
- An API call for face detection is tightly coupled with (or embedded, or hard-coded in) RawPicture.
  - The choice of an external API might change in the near future.

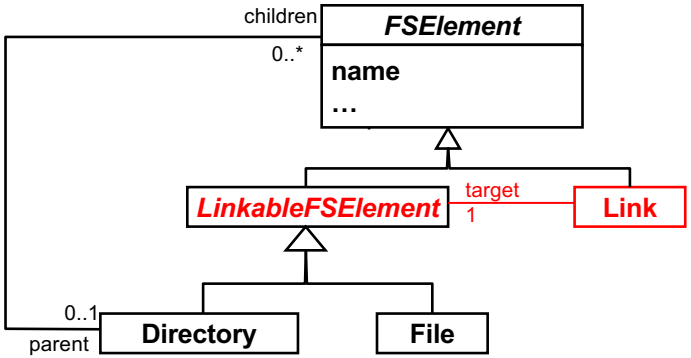
Another Example: Proxies of Files and Directories in File Systems



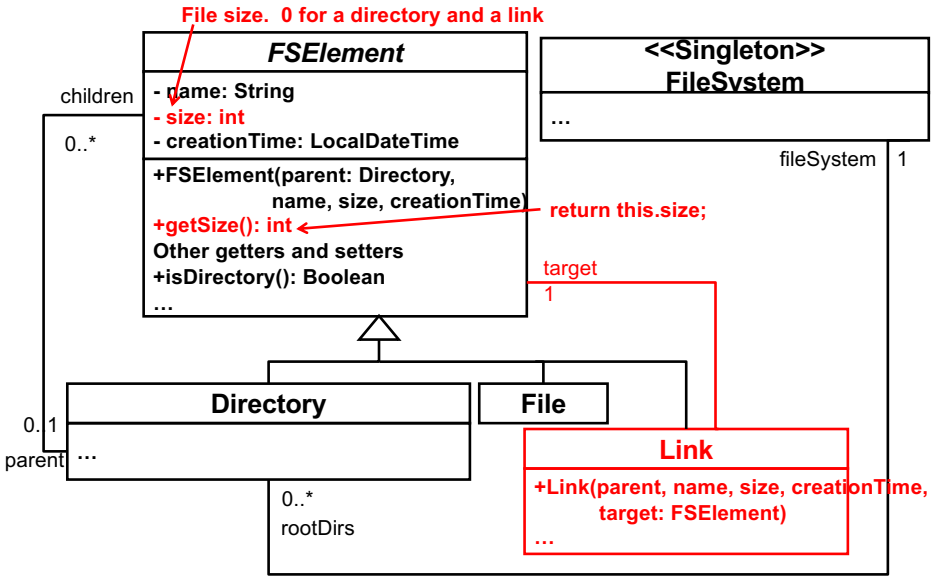
- Let's add symbolic links in addition to files and directories
  - a.k.a. alias (Mac), shortcut (Windows)
  - `> ln -s <destination path> <link name/path>`
- A link acts as a proxy of a directory or file.
- Use the *Proxy* design pattern.

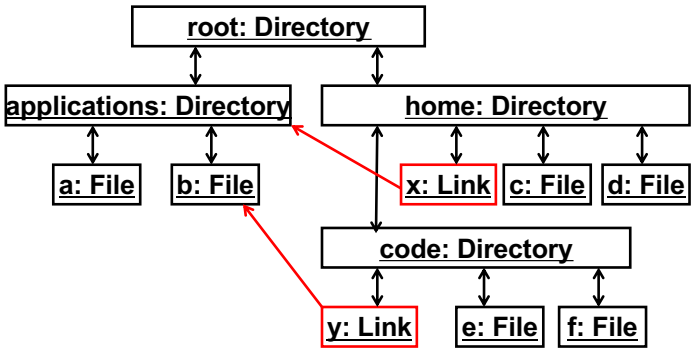


- A link acts as a proxy of a directory or file.
  - A link can act as a proxy of another link too.



HW 7: Implement This





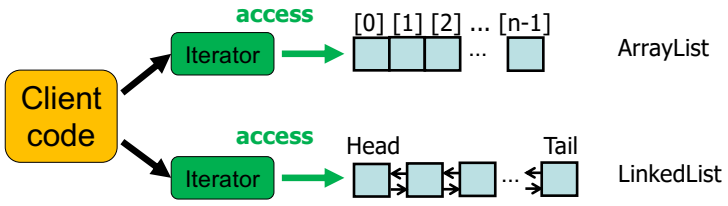
- Use this tree structure as a test fixture for your test cases.
  - Assign values to data fields (size, etc) as you like.
- Due: Nov 12 (Tue) midnight

Iterator Design Pattern

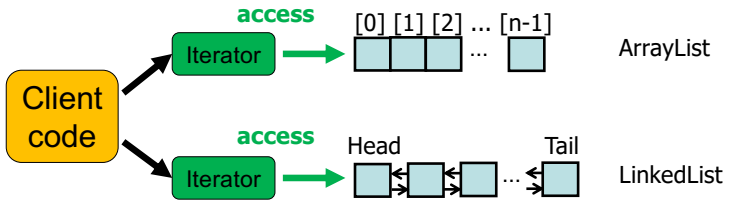
Iterator Design Pattern

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

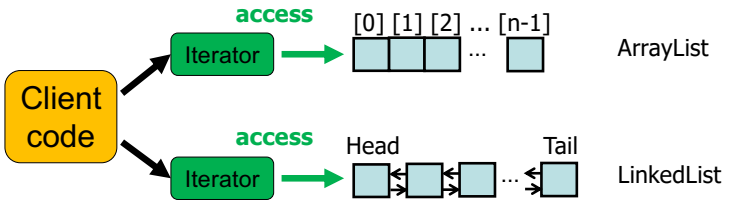
- Provides a uniform way to sequentially access collection elements without exposing its underlying representation (data structure).
  - Offers the same way (i.e., same set of methods) to access different types of collection elements
    - e.g., lists, queues, sets, maps, stacks, trees, graphs…



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

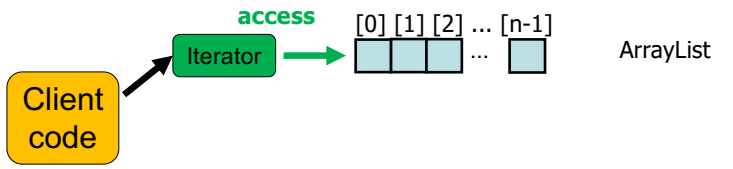


- Provides a uniform way to sequentially access collection elements without exposing its underlying representation (data structure).
  - Abstracts away different access mechanisms for different collection types.
    - Separates (or decouples) a collection's data structure and its access mechanism (i.e.,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



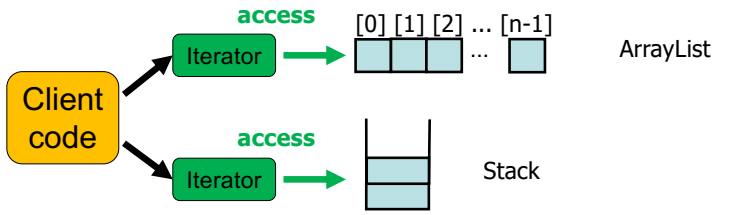
An Example in Java

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




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

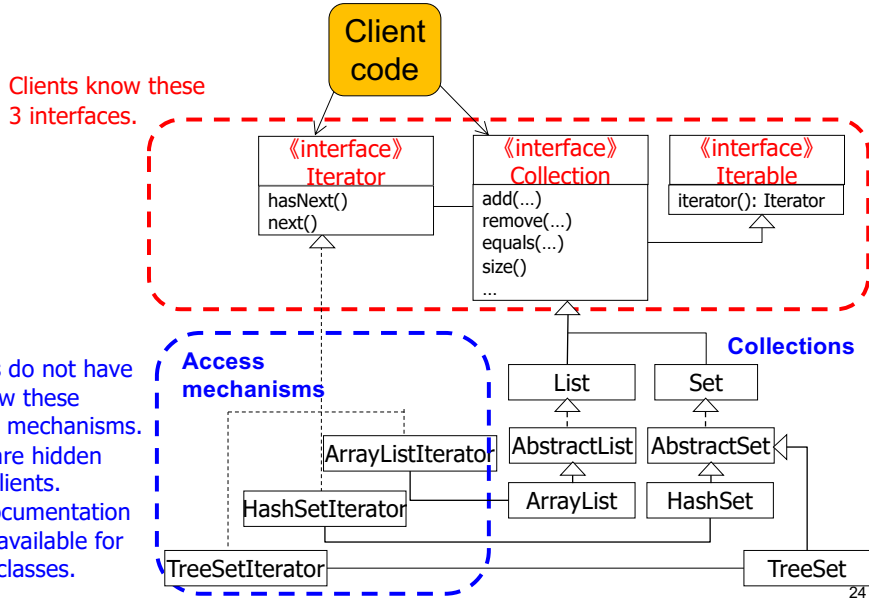
```
Stack<String> collection = new Stack<String>();
...
java.util.Iterator<String> iterator = collection.iterator();
while ( iterator.hasNext() ) {
    Object o = iterator.next();
    System.out.print( o );}
```



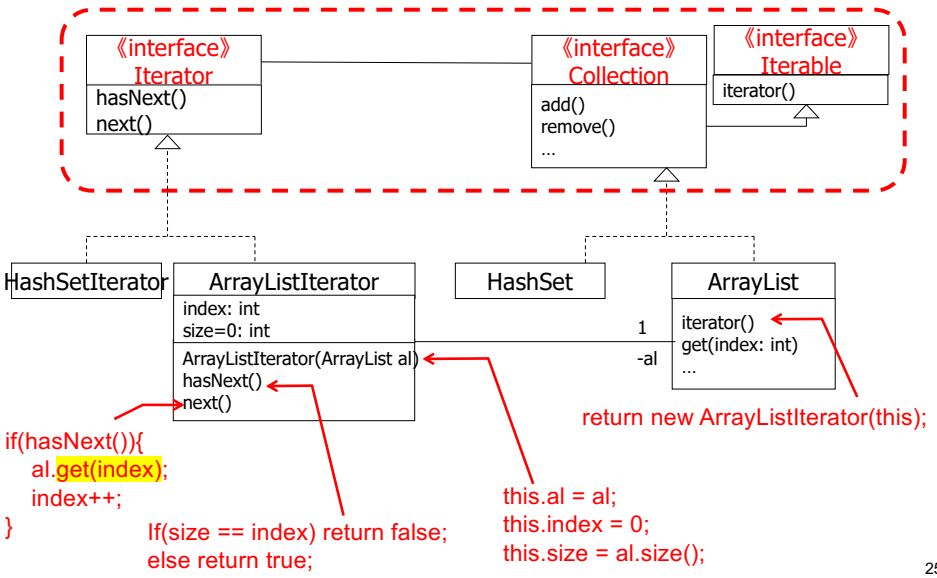
```
• ArrayList<Integer> collection = new ArrayList<Integer>();
  ...
  java.util.Iterator<Integer> iterator = collection.iterator();
  while ( iterator.hasNext() ) {
       Object o = iterator.next();
       System.out.print( o ); }

• 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 Clients?

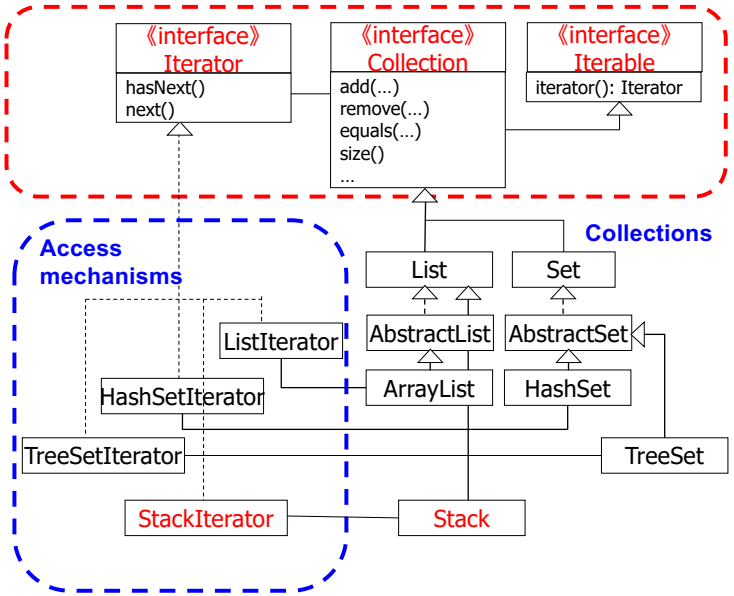


Key Points

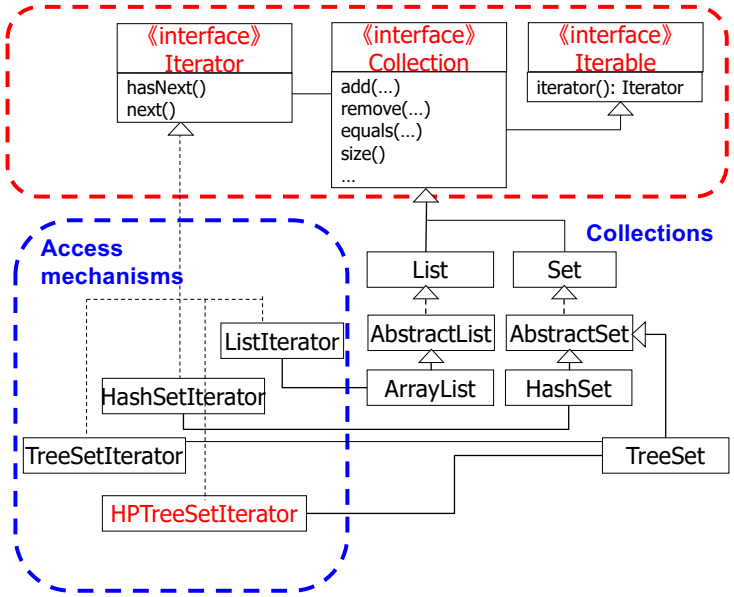
- In client's point of view
  - `java.util.Iterator iterator = collection.iterator();`
  - An iterator always implement the `Iterator` interface.
  - No need to know what specific *implementation class* is returned/used.
    - In fact, `ArrayListIterator` does not appear in the Java API documentation.
  - 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.

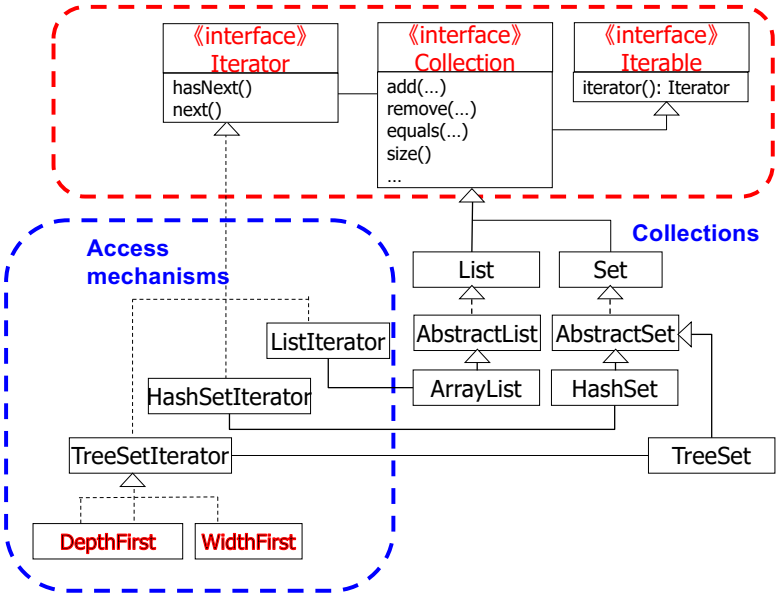
- In collection developer's (API designer's) point of view
  - No need to change
    - **`Iterator`** and **`Iterable`** interfaces
    - existing access mechanism classes
  - even if…
    - new collection classes are added.
    - existing collections (their method bodies) need to be modified.

Adding a New Collection

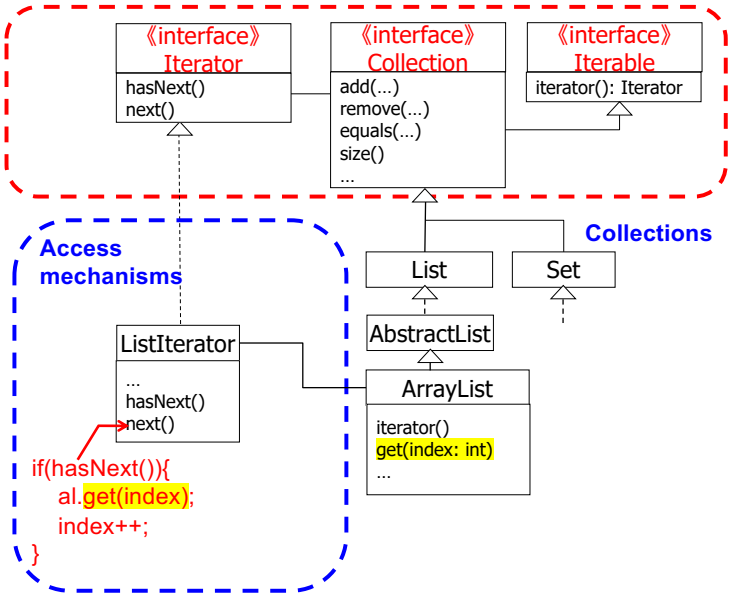


Adding New Access Mechanisms



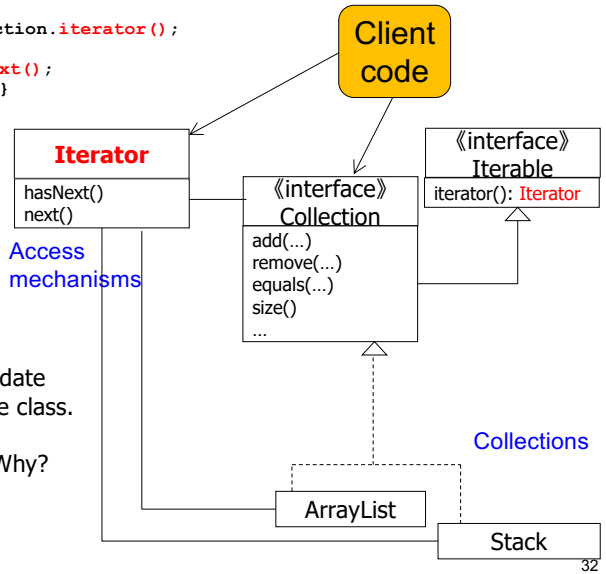


Modifying Existing Access Mechanisms



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 );}
```



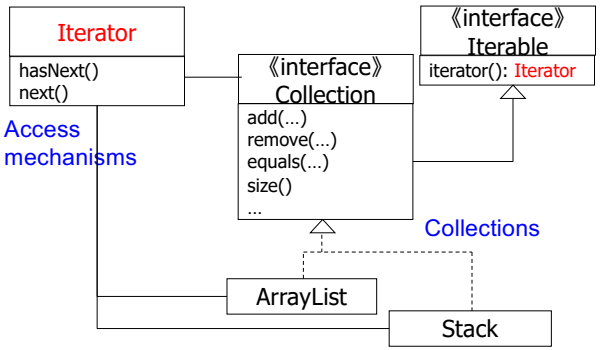
Iterator is defined as a class.

Java API designers could consolidate all access mechanisms in a single class.

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

- `Iterator` becomes error-prone (not that maintainable).
  - Iterator's methods need to have a long sequence of conditionals.
    - What if a new collection class is added or an existing collection class is modified?

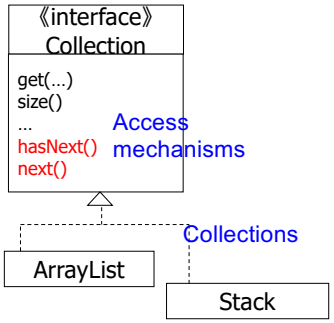
- This design is okay for collection users, but not good for collection API designers.
- Several books on design patterns use this design as an example of *Iterator*…



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

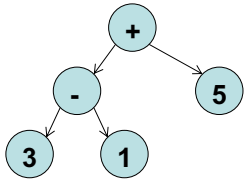
In fact, the right one is better in that it does not have conditionals in hasNext() and next().

In both designs, you cannot define collections and 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)) }
```