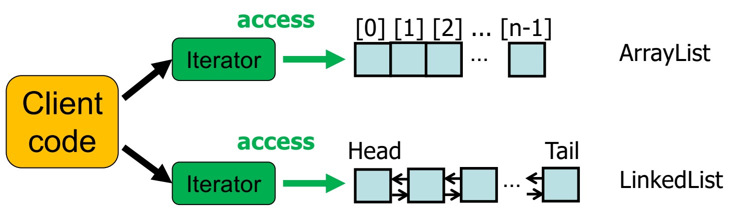
Iterator Design Pattern

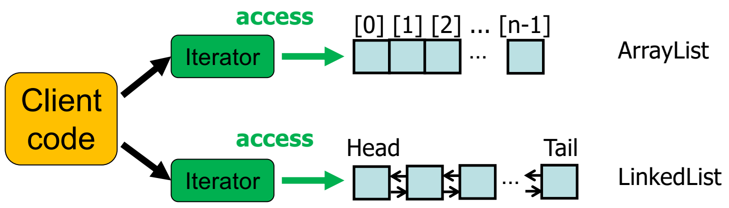
Iterator Design Pattern

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

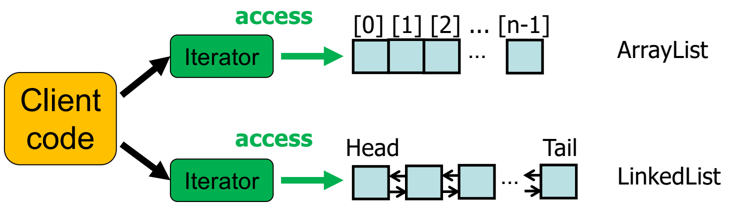
- Provides a uniform way to sequentially access collection elements without exposing its underlying representation (data structure).
  - Offers the same way (i.e., same set of methods) to access different types of collection elements
    - e.g., lists, queues, sets, maps, stacks, trees, graphs…



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

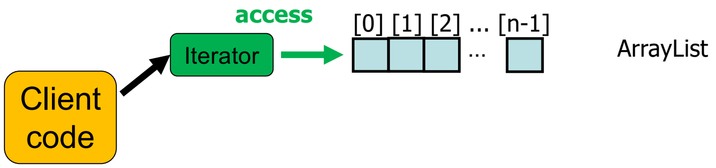


- Provides a uniform way to sequentially access collection elements without exposing its underlying representation (data structure).
  - Abstracts away different access mechanisms for different collection types.
    - Separates a collection's data structure and its access mechanism (i.e., how to get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



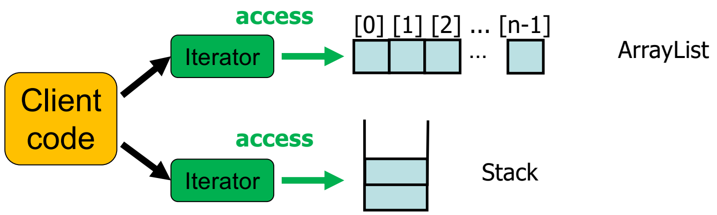
An Example in Java

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



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

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

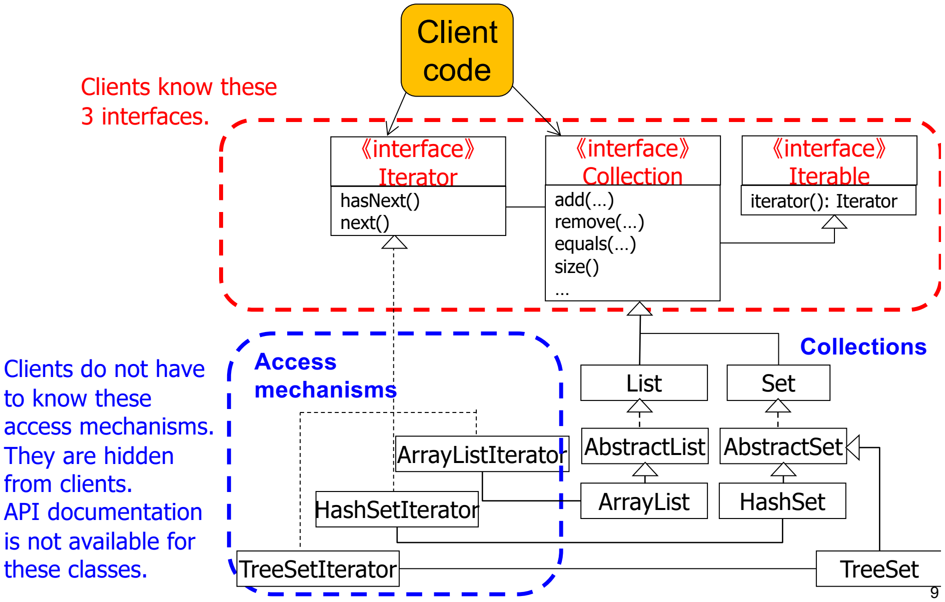


```
ArrayList<Integer> collection = new ArrayList<>();
...
java.util.Iterator<Integer> iterator = collection.iterator();
while ( iterator.hasNext() ) {
     Integer o = iterator.next();
     System.out.print( o ); }
```

```
Stack<String> collection = new Stack<>();
...
java.util.Iterator<String> iterator = collection.iterator();
while ( iterator.hasNext() ) {
     String 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.
- Actual access mechanisms (i.e., how to get collection elements) are hidden by iterators.

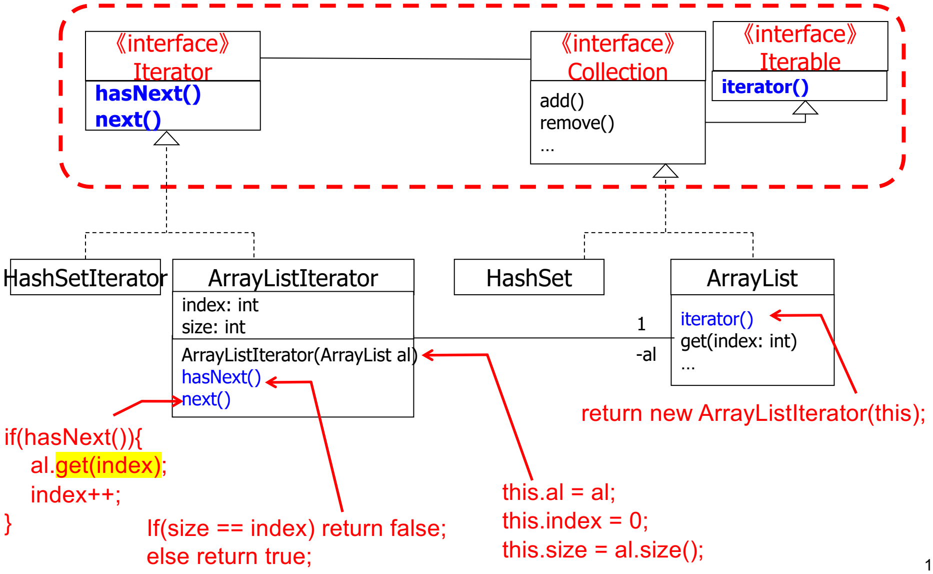
Class Structure



Clients know these 3 interfaces.

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

What’s Hidden from Clients?



return new ArrayListIterator(this);

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

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

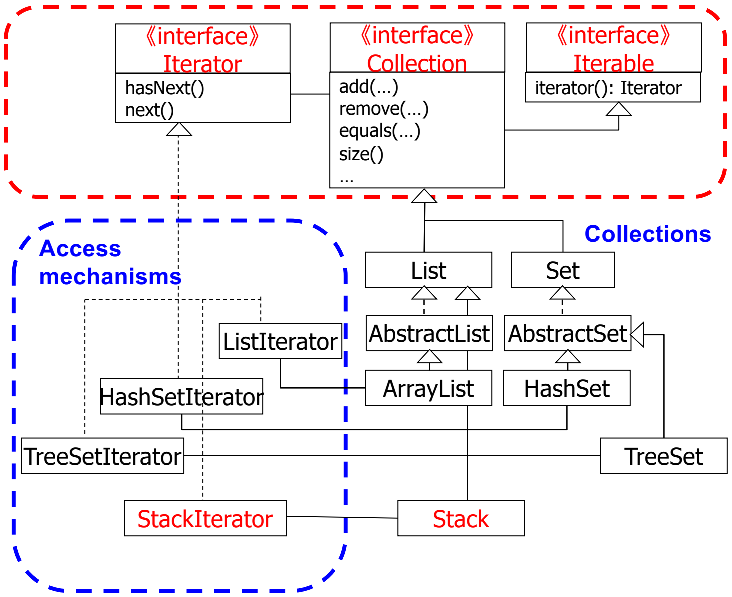
this.al = al;
this.index = 0;
this.size = al.size();

Key Points

- In 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()`
    - 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` even if collection classes are added/removed.
  - No need to change client code even if access mechanisms are modified.

Adding a New Collection



《interface》 Iterator
hasNext()
next()

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

《interface》 Iterable
iterator(): Iterator

Access mechanisms

Collections

List

Set

AbstractList

AbstractSet

ListIterator

ArrayList

HashSet

HashSetIterator

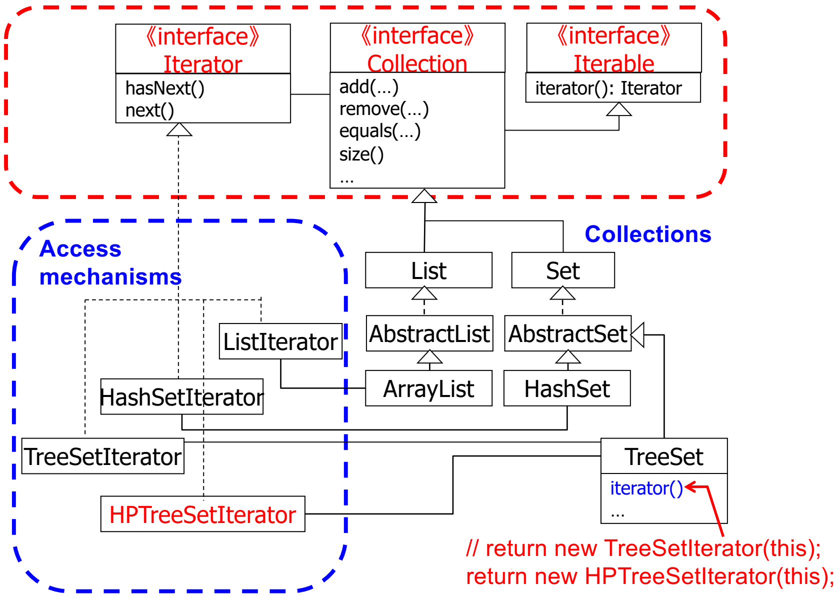
TreeSetIterator

TreeSet

StackIterator

Stack

Adding New Access Mechanisms



《interface》 Iterator
hasNext()
next()

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

《interface》 Iterable
iterator(): Iterator

Access mechanisms

Collections

List

Set

AbstractList

AbstractSet

ListIterator

ArrayList

HashSet

HashSetIterator

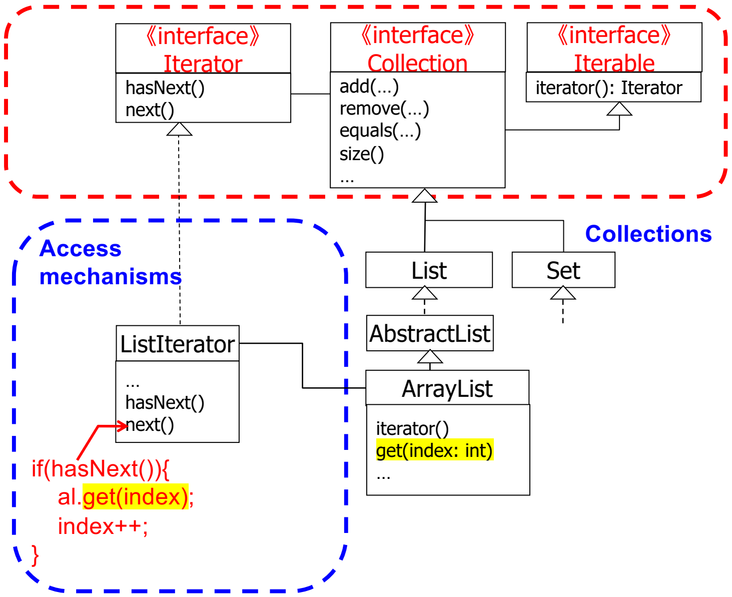
TreeSetIterator

TreeSet
iterator()
...

HPTreeSetIterator

// return new TreeSetIterator(this);
return new HPTreeSetIterator(this);

Modifying Existing Access Mechanisms



《interface》 Iterator
hasNext()
next()

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

《interface》 Iterable
iterator(): Iterator

Access mechanisms

Collections

List

Set

AbstractList

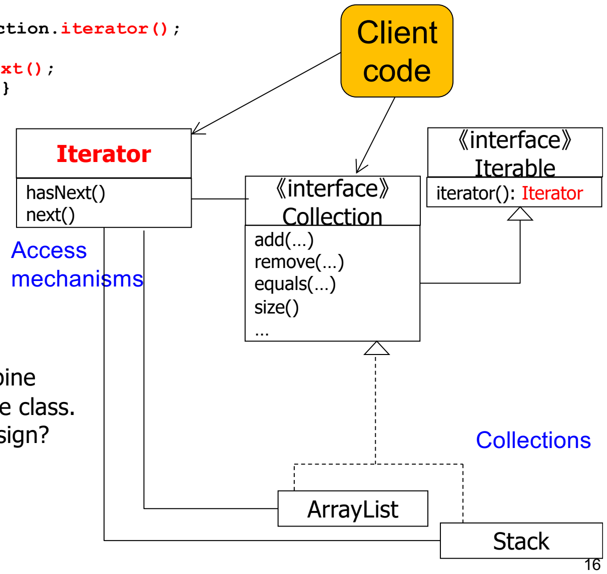
ListIterator
...
hasNext()
next()

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

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

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 );}
```



Client code

Iterator
hasNext()
next()

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

《interface》 Iterable
iterator(): Iterator

Access mechanisms

Collections

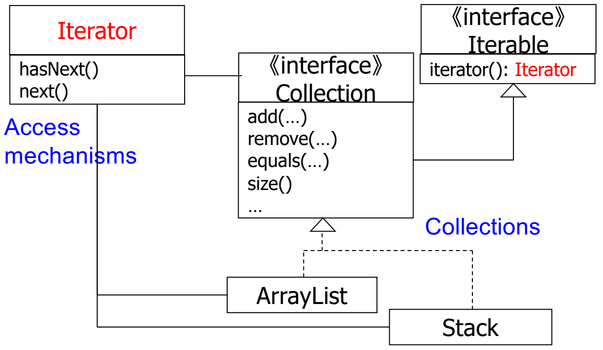
ArrayList

Stack

Iterator is defined as a class.

Java API designers did not combine all access mechanisms in a single class. 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*…



Iterator
hasNext()
next()
Access mechanisms

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

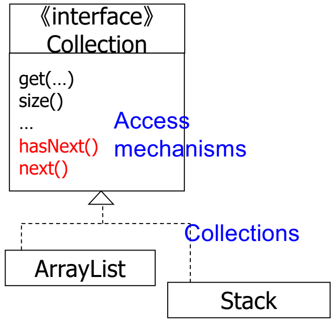
《interface》
Iterable
iterator(): Iterator

Collections
ArrayList
Stack

These two designs are same in that both do not separate 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 access mechanisms in a "pluggable" way.

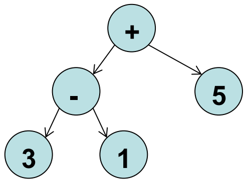


《interface》
Collection
get(…)
size()
…
hasNext()
next()
Access mechanisms

Collections
ArrayList
Stack

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()`
- "leaf-to-root" width-first policy



By the way… *for-each* Expression

- Java 5 introduced ***for-each*** expressions.

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

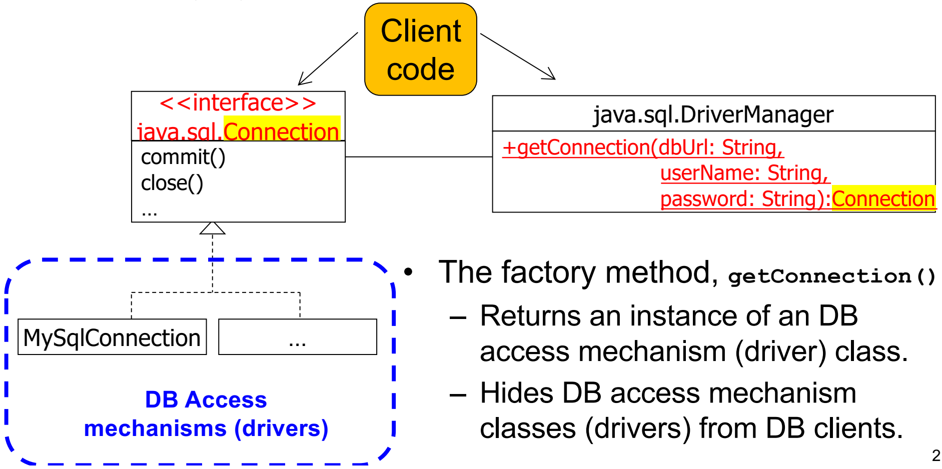
  - No need to explicitly use an iterator.
- Note that "for-each" is a *syntactic sugar* for iterator-based code.
  - The above code is automatically transformed to the following code during a compilation:

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

A Similar Example:

DriverManager.getConnection() in JDBC API

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


Client code

<<interface>>
java.sql.Connection
commit()
close()
...

java.sql.DriverManager
+getConnection(dbUrl: String,
userName: String,
password: String):Connection

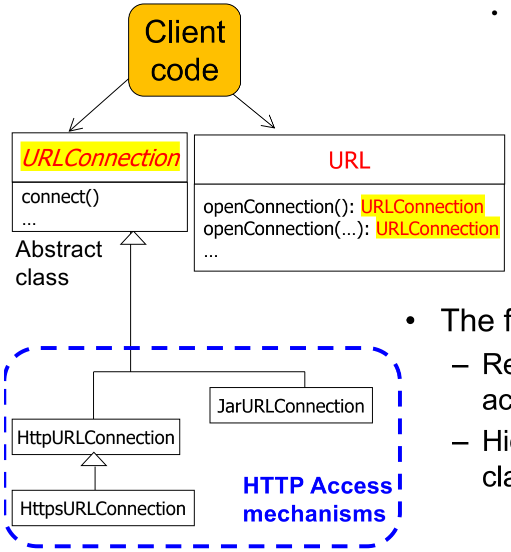
MySqlConnection
...

DB Access mechanisms (drivers)

- The factory method, getConnection()
  - Returns an instance of an DB access mechanism (driver) class.
  - Hides DB access mechanism classes (drivers) from DB clients.

Another Example:

URL and URLConnection in Java API


Client code

URLConnection
connect()
...

Abstract class

URL
openConnection(): URLConnection
openConnection(...): URLConnection
...

JarURLConnection
HttpURLConnection
HttpsURLConnection

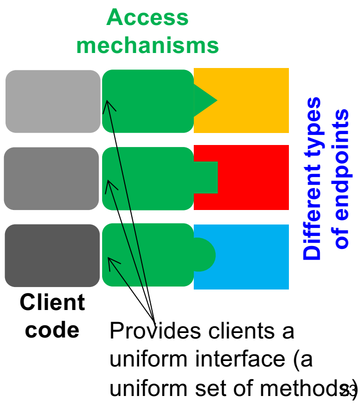
HTTP Access mechanisms

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

- The factory method, openConnection()
  - Returns an instance of a HTTP access mechanism class
  - Hides HTTP access mechanism classes from HTTP clients

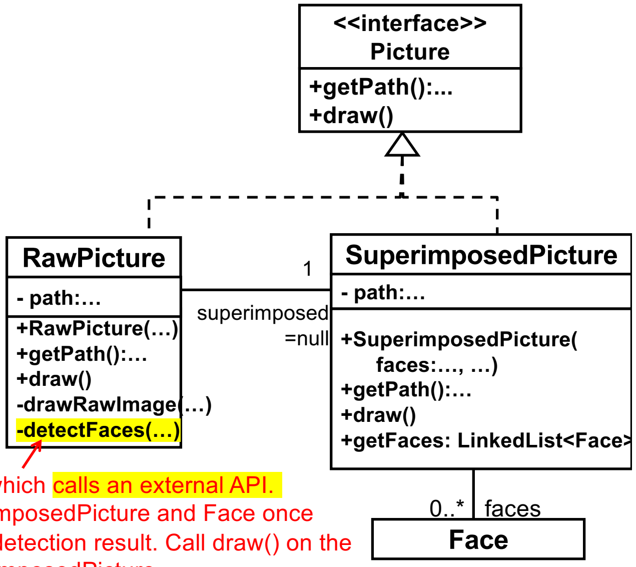
This Design Pattern's Name…

- Has been outdated
  - Now that most languages have implemented iterators.
- This pattern's design principle is still important.
  - It is not limited to the development of iterators.
- Alternative/better name
  - *Abstract access mechanism*?
  - *Pluggable driver*??
  - *Glue*???



Provides clients a uniform interface (a uniform set of methods)

Recap: Face Detection with *Proxy*


<<interface>>
Picture
+getPath():…
+draw()

RawPicture
- path:…
+RawPicture(…)
+getPath():…
+draw()
-drawRawImage(…)
-detectFaces(…)

1
superimposed =null

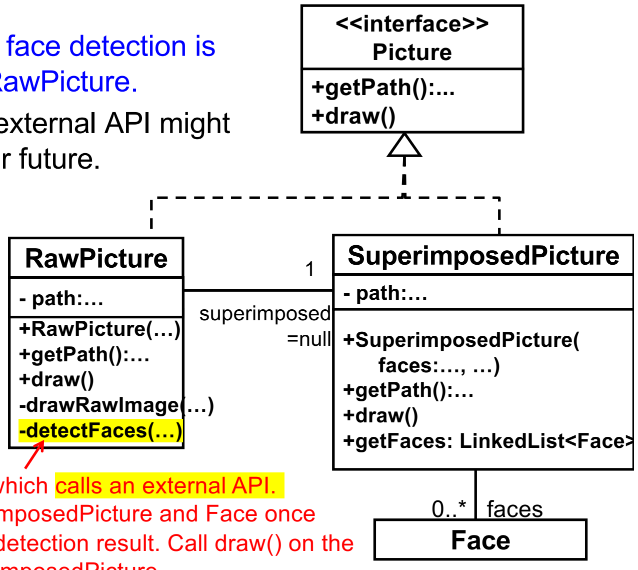
SuperimposedPicture
- path:…
+SuperimposedPicture(faces:…, …)
+getPath():…
+draw()
+getFaces: LinkedList<Face>

0..* faces

Face

Create a thread, which calls an external API. Instantiate SuperimposedPicture and Face once the API returns a detection result. Call draw() on the instance of SuperimposedPicture.

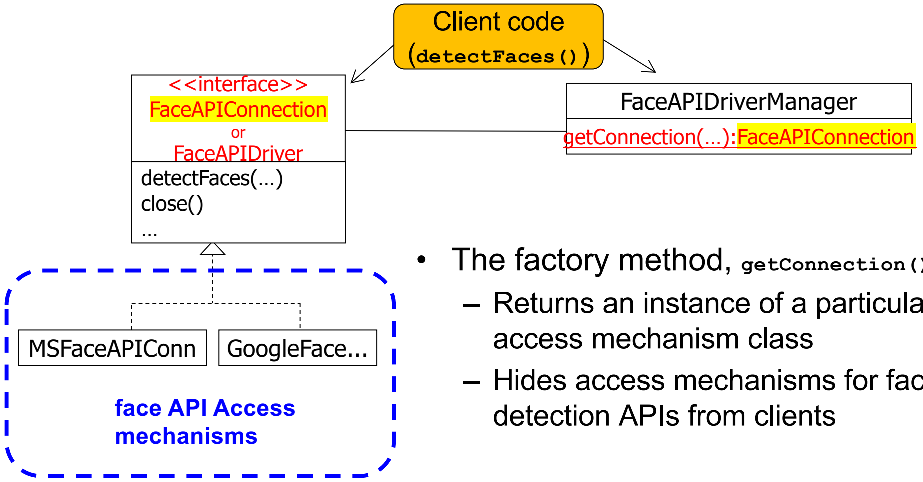
- Issue: An API call for face detection is tightly coupled with RawPicture.
  - The choice of an external API might change in the near future.



Create a thread, which calls an external API. Instantiate SuperimposedPicture and Face once the API returns a detection result. Call draw() on the instance of SuperimposedPicture.

- Have `detectFaces()` obtain an access mechanism to a particular face detection API based on *Iterator*-inspired design.

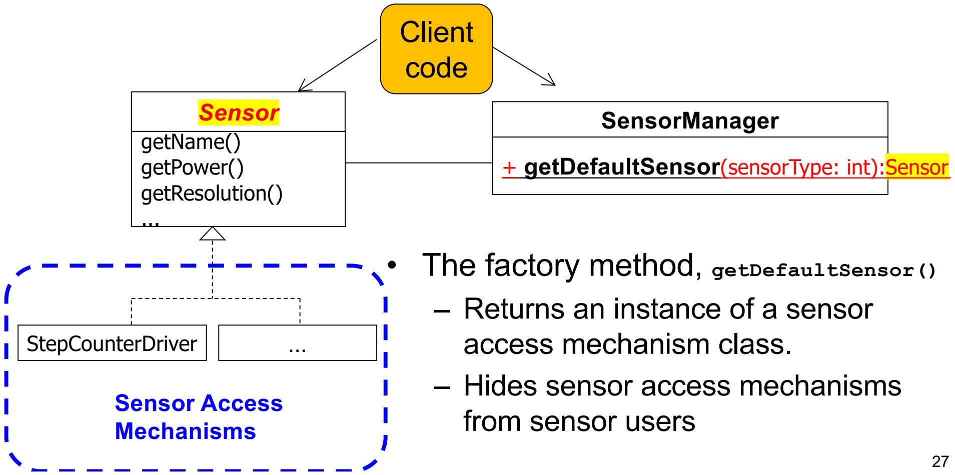
  -
    ```
    FaceAPIConnection conn =
      FaceAPIDriverManager.getConnection(...);
    conn.detectFaces(...);
    ```



- The factory method, `getConnection()`,
  - Returns an instance of a particular access mechanism class
  - Hides access mechanisms for face detection APIs from clients

One More: Slightly Modified Android Sensor API

-
  ```
  Sensor stepCounter =
    SensorManager.getDefaultSensor(Sensor.TYPE_STEP_COUNTER);
  stepCounter.getPower();
  ```



- The factory method, `getDefaultSensor()`
  - Returns an instance of a sensor access mechanism class.
  - Hides sensor access mechanisms from sensor users