Singleton Design Pattern

Singleton Design Pattern

- Guarantees that a class has only one instance.
  - More like a programming idiom.

```
public class Singleton{
    private Singleton(){};
    private static Singleton instance = null;

    public static Singleton getInstance(){
        if(instance==null)
            instance = new Singleton ();
        return instance;
    }
}
```

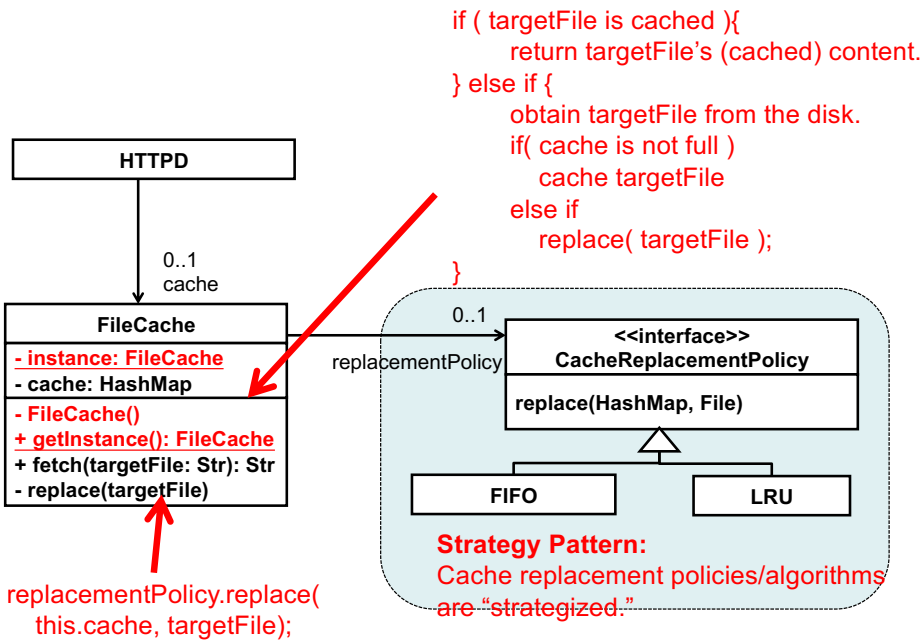
- You should not define public constructors.
- If you do not define constructors...

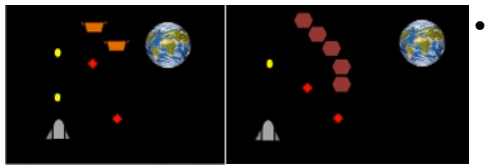

```
Singleton instance = Singleton.getInstance();
instance.hashCode();
Singleton instance = Singleton.getInstance();
instance.hashCode();
```

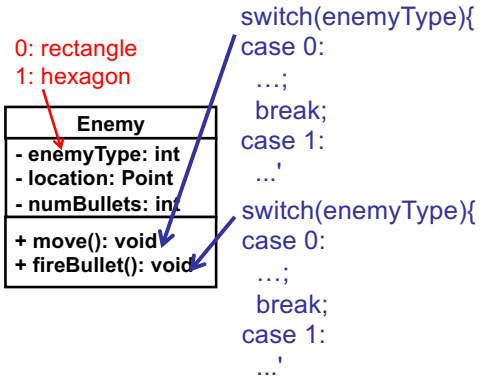
- hashCode() returns a unique ID for an instance.
  - Different instances of a class have different IDs.

What Can be a Singleton?

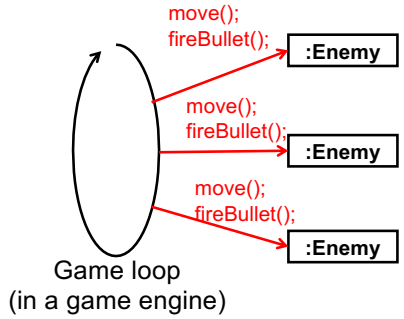
- Object pools
- Game loop
- Logger
- Plugin manager
- Access counter
- …, etc.







- Each type of enemies has its own attack pattern.
  - e.g. How to move, when to fire bullets, how to fire bullets…



Alternative Implementation with Null Checking API in Java 7

- java.util.Objects, extending java.lang.Object
  - A utility class (i.e., a set of static methods) for the instances of java.lang.Object and its subclasses.

```
class Foo{
    private String str;
    public Foo()(String str){
        this.str = Objects.requireNonNull(str);
    } }
```

  - `requireNonNull()` throws a `NullPointerException` if `str==null`. Otherwise, it simply returns `str`.

```
class Foo{
    private String str;
    public Foo()(String str){
        this.str = Objects.requireNonNull(
                    str, "str must be non-null!!!");
    } }
```

  - `requireNonNull()` can accept an error message, which is to be contained in a `NullPointerException`.

- Traditional null checking
  -
```
if(str == null)
    throw new NullPointerException();
this.str = str;
```

- With `Objects.requireNonNull()`
  - `this.str = Objects.requireNonNull(str);`
- Can eliminate an explicit conditional statement and make code simpler.

Singleton Impl. with `Objects.requireNonNull()`

```
public class SingletonNullCheckingJava7{
      private SingletonNullCheckingJava7(){};
      private static SingletonNullCheckingJava7 instance = null;

      public static SingletonNullCheckingJava7 getInstance(){
        try{
            return Objects.requireNonNull(instance);
        }
        catch(NullPointerException ex){
            instance = new SingletonNullCheckingJava7();
            return instance;
      } }
```

```
public class Singleton{
      private Singleton(){};
      private static Singleton instance = null;

      public static Singleton getInstance(){
             if(instance==null)
                    instance = new Singleton ();
             return instance;
      } }
```

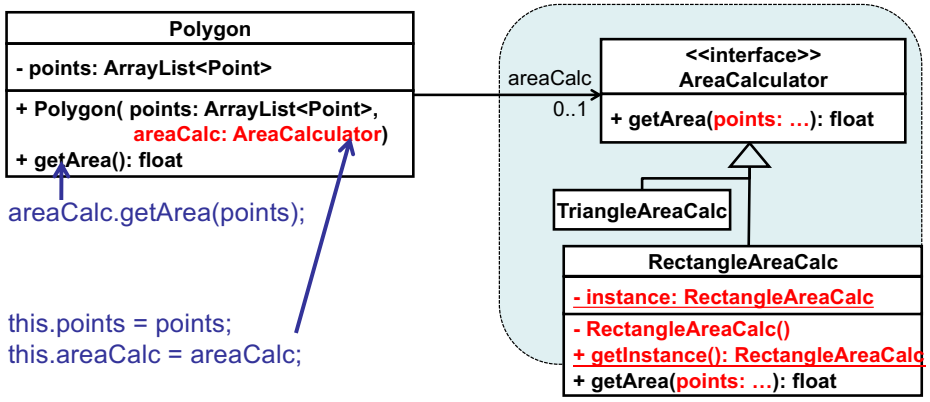

Singleton as *Factory Method*

- `getInstance()` is a factory method

```
public class Singleton{
     private Singleton(){};
     private static Singleton instance = null;

     public static Singleton getInstance(){ // factory method
            if(instance==null)
                   instance = new Singleton ();
            return instance;
     }
}
```


Singleton in *Strategy*

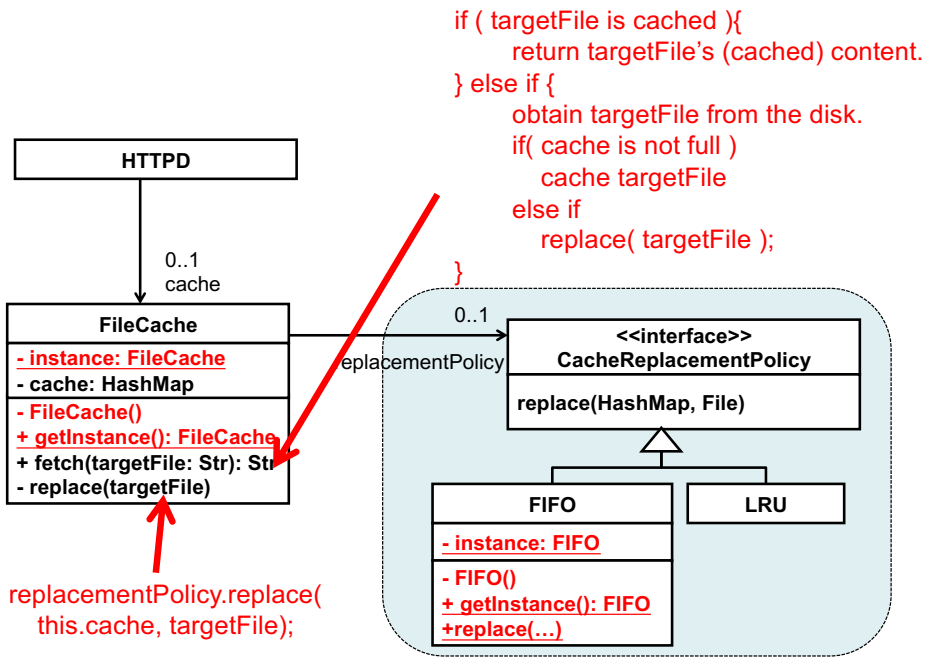


User/client of Polygon:

```
ArrayList<Point> al = new ArrayList<Point>();
al.add( new Point(…) ); al.add( new Point(…) ); al.add( new Point(…) );
Polygon p = new Polygon( al, RectangleAreaCalc.getInstance() );
p.getArea();
```


HW 10

- Implement FileCache with *Singleton* and *Strategy*.
  - Implement FIFO as a replacement policy.
  - [Optional] Implement another policy as well.
  - Implement setCacheReplacementPolicy() in FileCache to pass an instance of FIFO to FileCache.

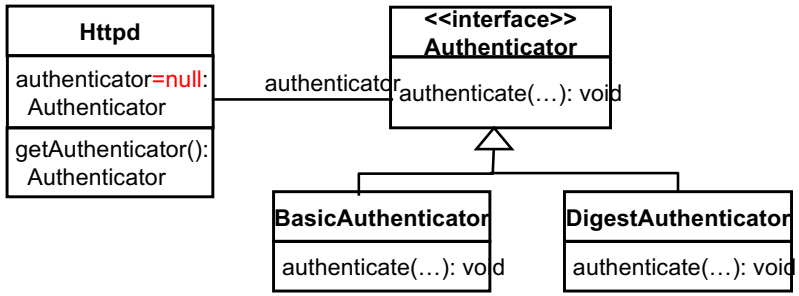


Null Object Design Pattern

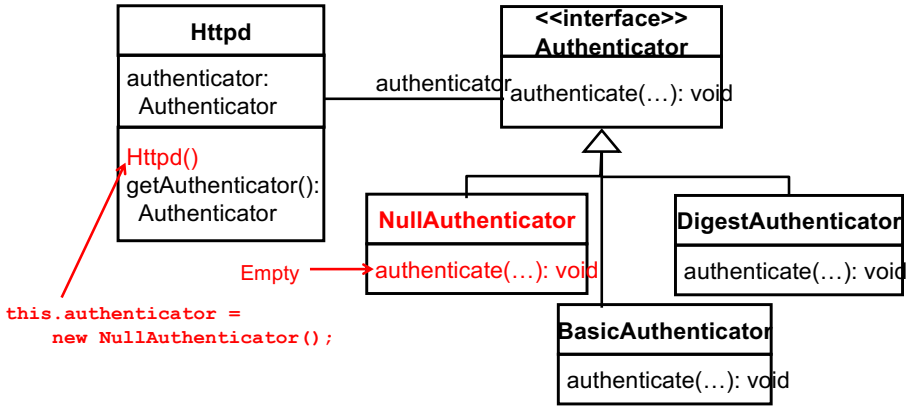
Null Object Design Pattern

- Intent
  - Encapsulate the implementation decisions of how to do nothing and hide those details from clients
  - Replace a *null-checking* (i.e., if statement) with a neutral/default object that does nothing.
  - B. Woolf, "Null Object," Chapter 1, PLoP 3, Addison-Wesley, 1998.
  - Refactoring: Introduce Null Object
    - http://sourcemaking.com/refactoring/introduce-null-object

An Example: Authentication in HTTP




```
if( getAuthenticator() != null ){
    try{
        getAuthenticator().authenticate(...);
    }catch(AuthenticationFailedException afe){
        makeErrorResponse(afe);
    }
}
makeSuccessfulResponse(...);
```

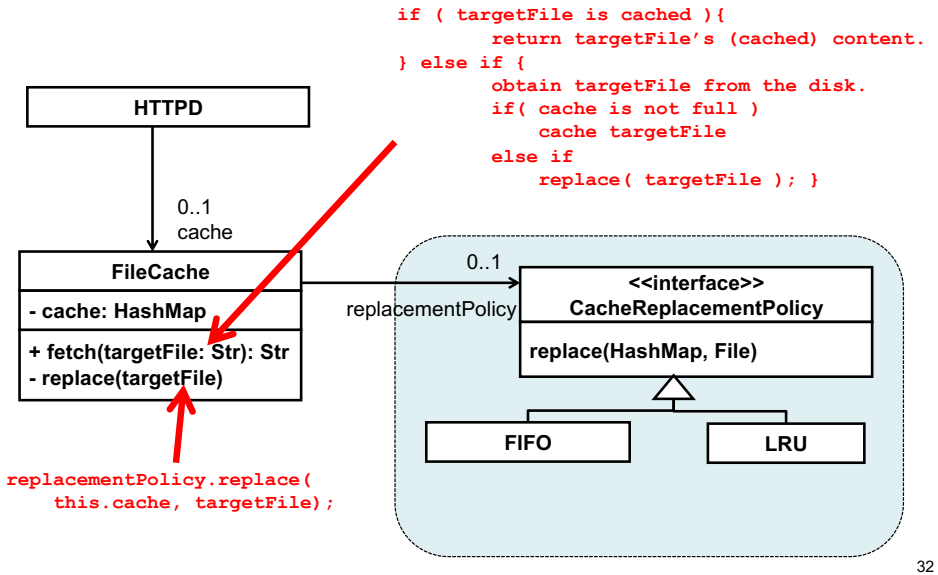


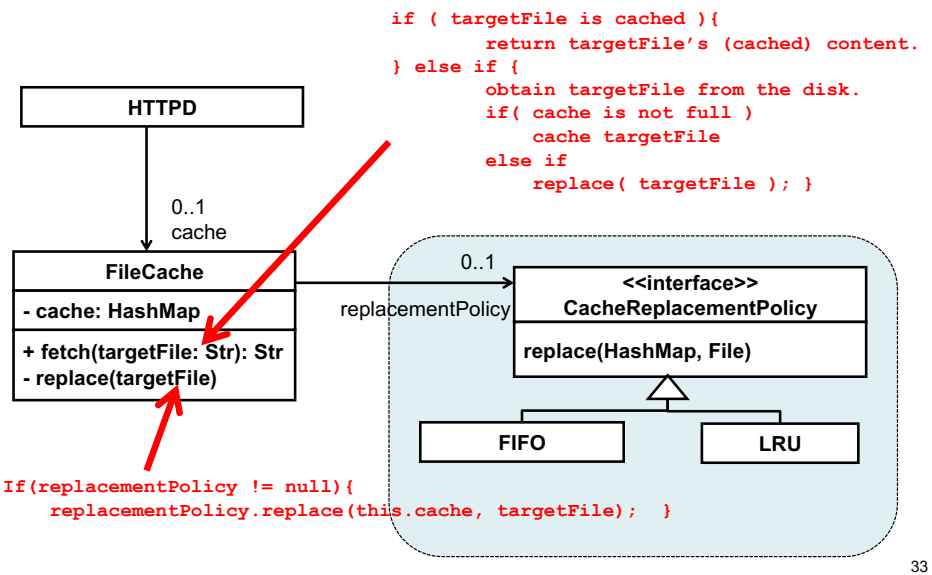

```
try{
    getAuthenticator().authenticate(...);
}catch(AuthenticationFailedException afe){
    makeErrorResponse(afe);
}
makeSuccessfulResponse(...);
```


Null Object as *Strategy*

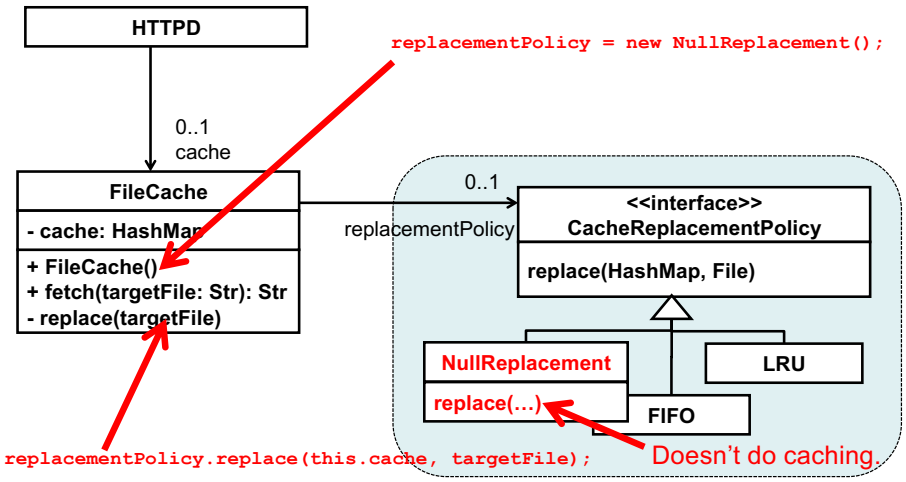
- Null object
  - A variant/application of *Strategy* that focuses on "doing nothing" by default.

Another Example: File Caching





File Caching with *Null Object*



HW 11

- Extend your HW 10 solution by adding NullReplacement.
  - Implement NullReplacement as a singleton class.
  - Keep FileCache as a singleton class.
  - Use NullReplacement by default.
  - Change it to FIFO dynamically.

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 (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

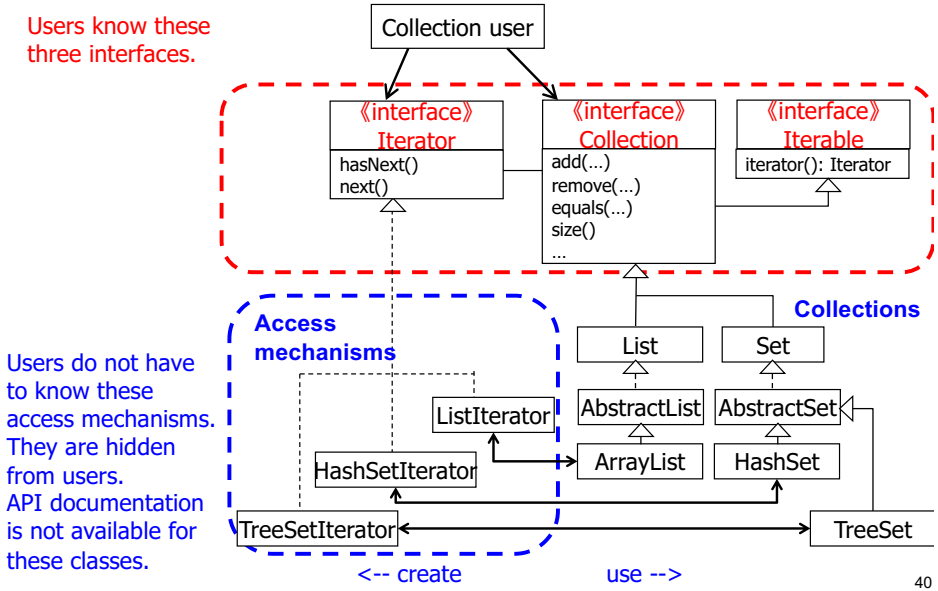
An Example in Java

```
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 );}
```

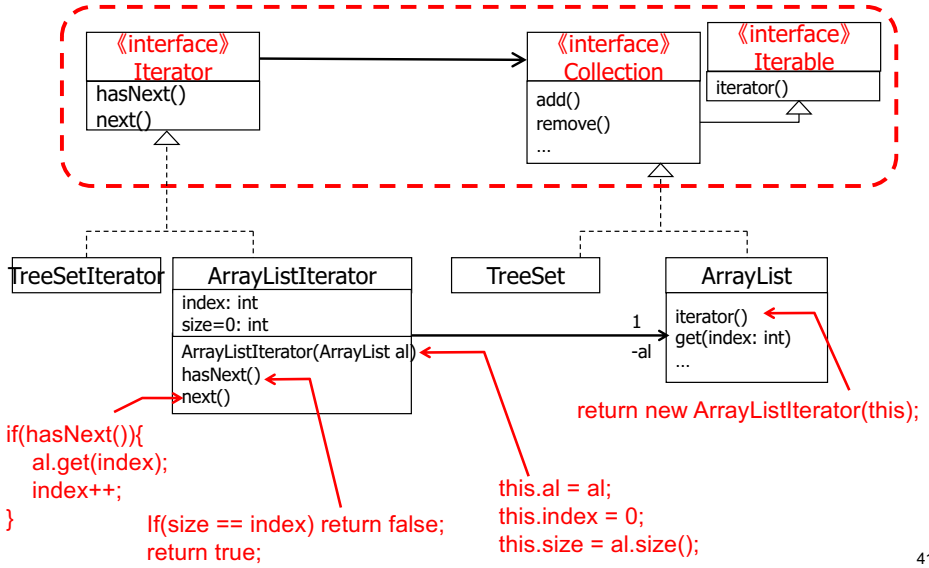
```
ArrayList<Integer> collection = new ArrayList<Integer>();
...
java.util.Iterator<Integer> iterator = collection.iterator();
while ( iterator.hasNext() ) {
     Object o = iterator.next();
     System.out.print( o ); }
```

- 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



What's Hidden from Users?

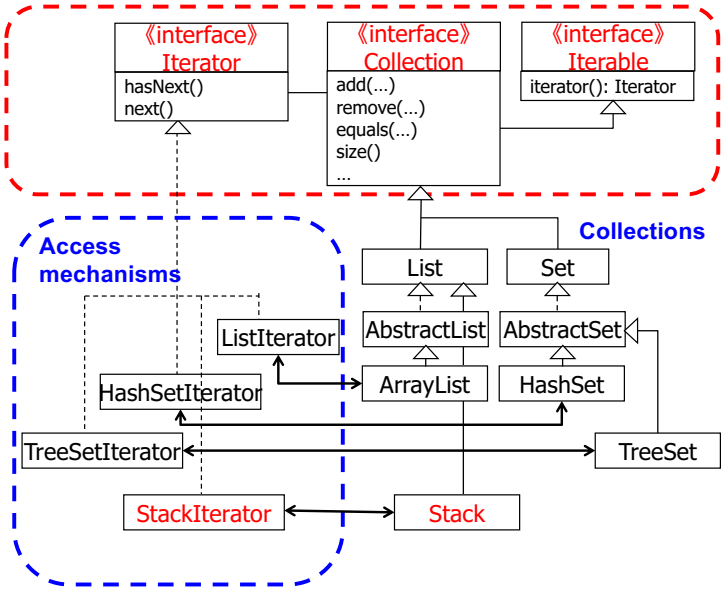


Key Points

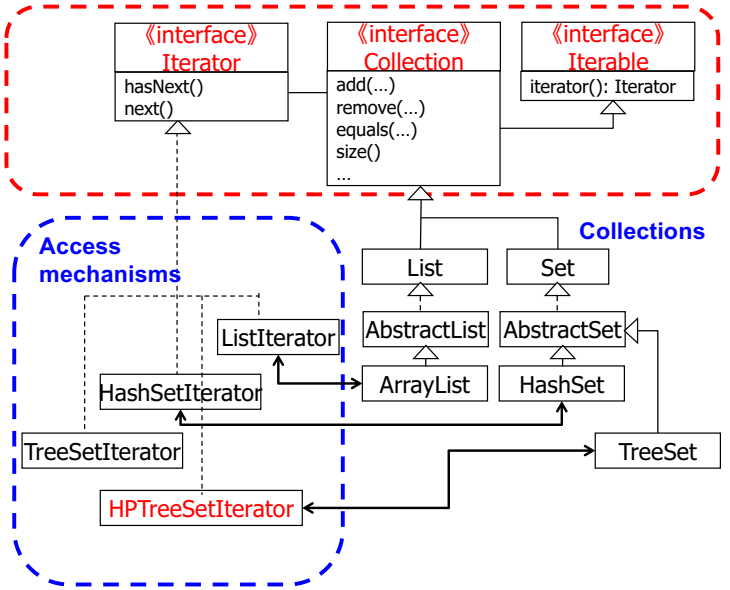
- In user's point of view
  - java.util.**Iterator** iterator = collection.**iterator()**;
  - An iterator always implement the Iterator interface.
  - No need to know what specific implementation class is returned/used.
    - In fact, ArrayListIterator does not appear in the Java API documentation.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

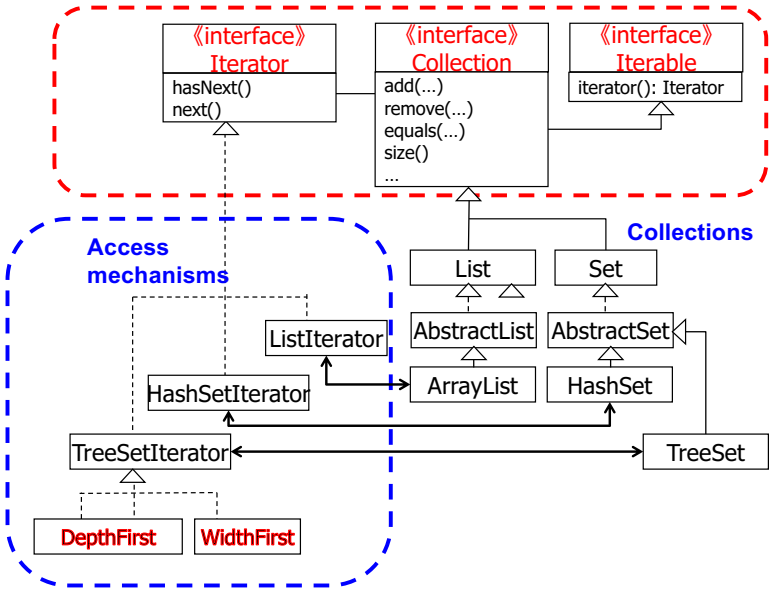
- In collection developer's (API designer's) point of view
  - No need to change
    - Iterator and Iterable interfaces
    - existing access mechanism classes
  - even if…
    - a new collection type is added.
    - existing collections (their method bodies) need to be modified.
- Important principle: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

Adding a New Collection (Stack)



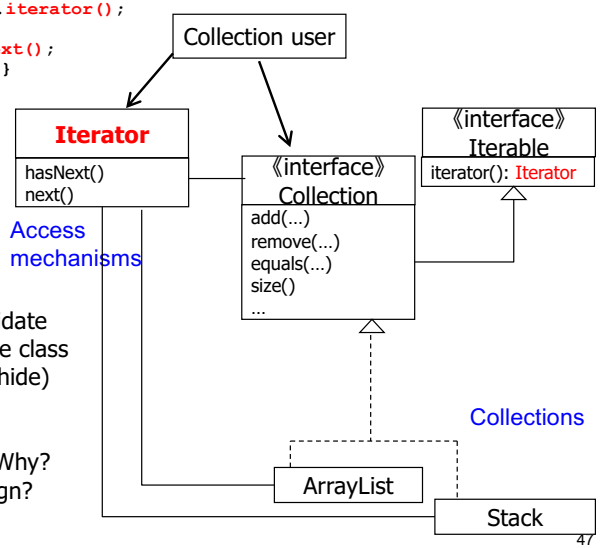
Adding New Access Mechanisms





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 );}
```

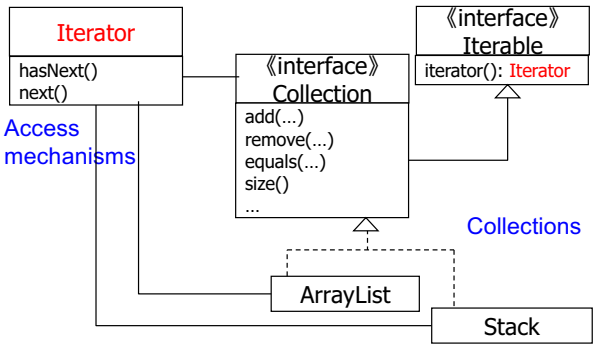


Iterator is defined as a class.

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?

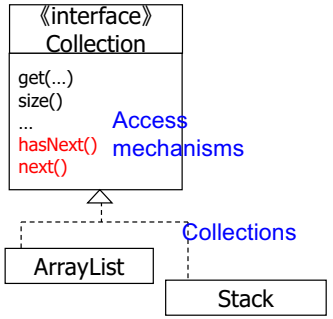
- Iterator becomes error-prone (not that maintainable).
  - Iterator's methods need to have a long sequence of conditional statements.
    - What if a new collection class is added or an existing collection class is modified?
    - What if a collection class's access methods are modified?
- This design 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*…



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

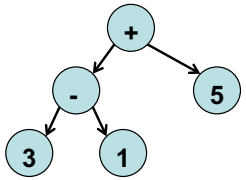
In fact, the right one is better in that it does not have conditional statements in hasNext() and next().

However, no way to define different iterators in a "pluggable" way.



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



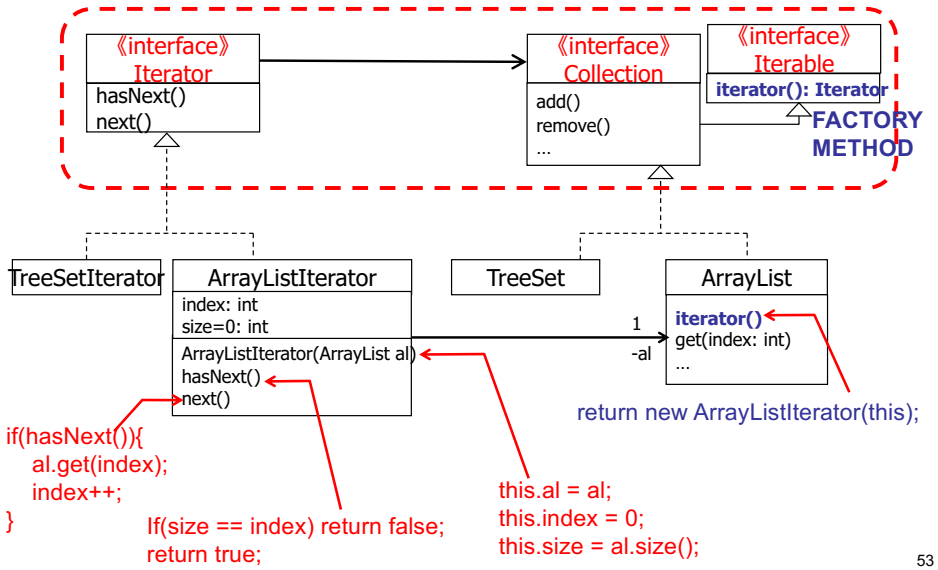
By the way…: *for-each* Expression

- 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

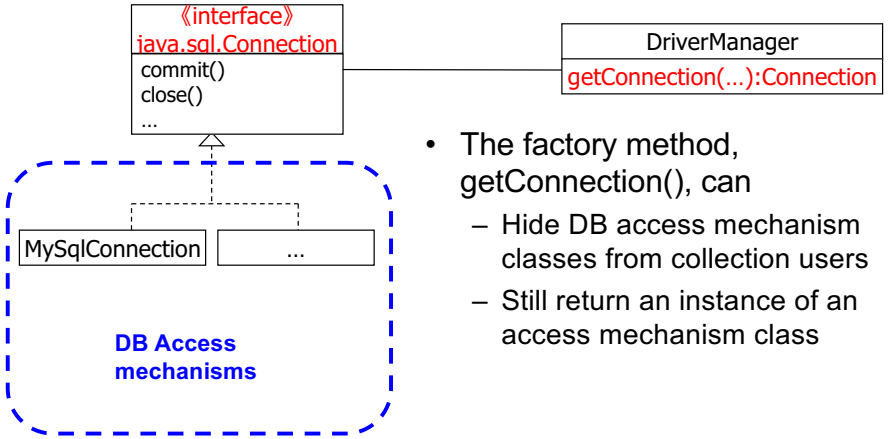


What's the Point?

- 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