Iterator Pattern

For the Complete Code, See the "Official" Head-First Design Patterns GitHub Repo:

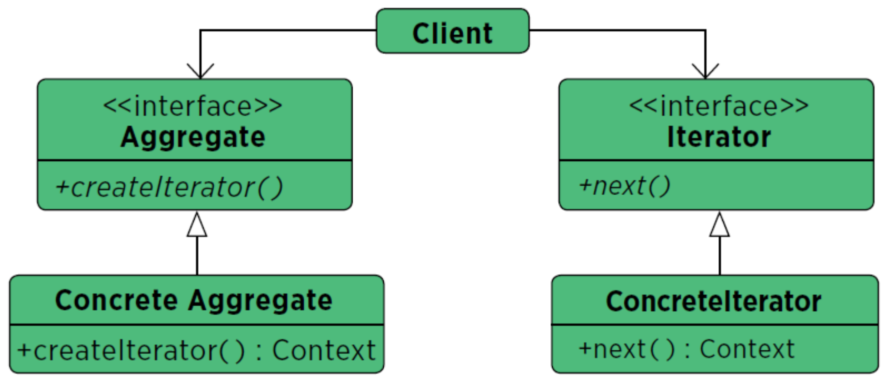
https://github.com/bethrobson/Head-First-Design-Patterns/tree/master/src/headfirst/designpatterns/

And the course SVN repo:

svn://cosc436.net:65436/Examples/trunk

ITERATOR

Object Behavioral



Purpose	Allows for access to the elements of an aggregate object without allowing access to its underlying representation.
Use When	• Access to elements is needed without access to the entire representation. • Multiple or concurrent traversals of the elements are needed. • A uniform interface for traversal is needed. • Subtle differences exist between the implementation details of various iterators.
Example	The Java implementation of the iterator pattern allows users to traverse various types of data sets without worrying about the underlying implementation of the collection. Since clients simply interact with the iterator interface, collections are left to define the appropriate iterator for themselves. Some will allow full access to the underlying data set while others may restrict certain functionalities, such as removing items.

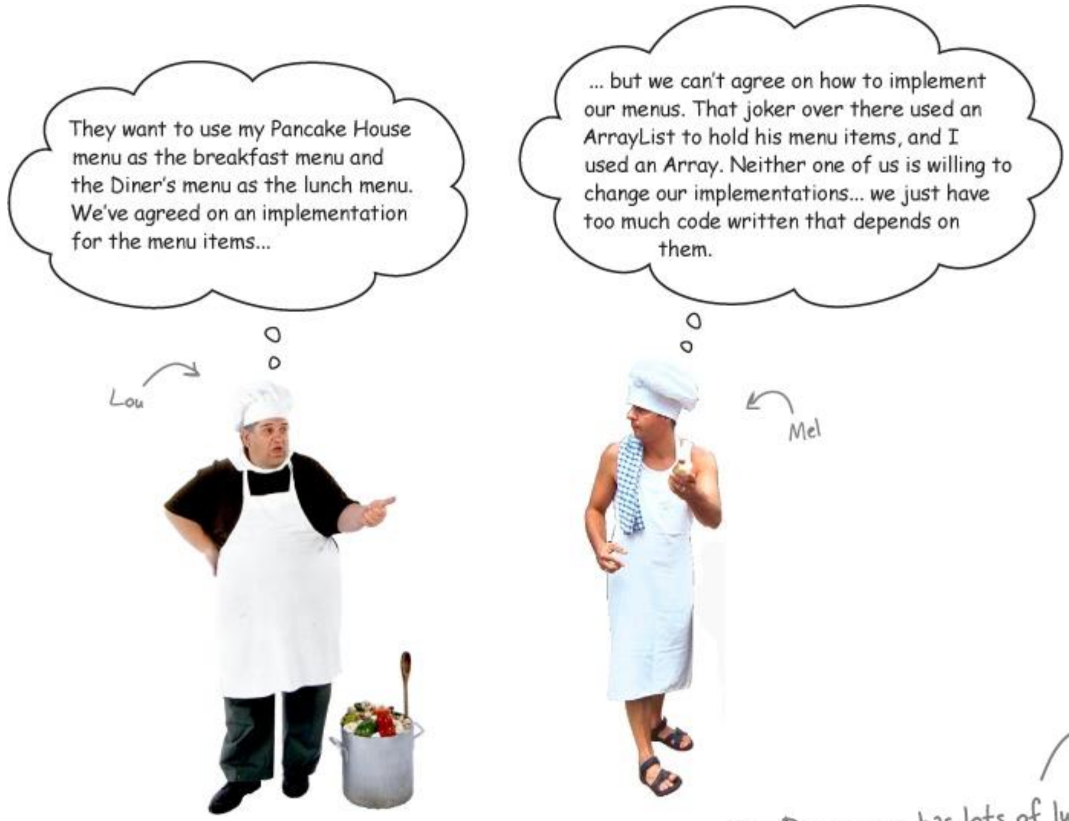
How many different types of collections in Java?

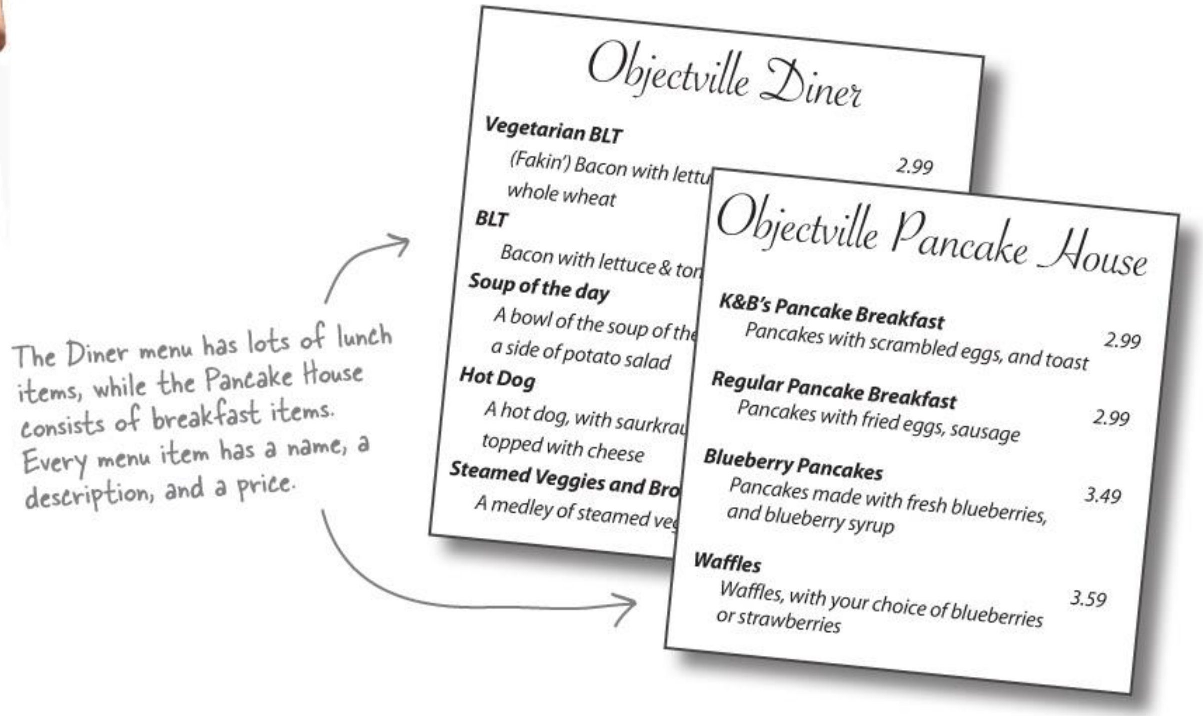
- Array, Stack, Queue, ArrayList, HashMap, etc.
- There are lots of ways to stuff objects into a collection.
- At some point your client is going to want to iterate over those objects, and when he does, are you going to show him your implementation?
- You're going to see how you can allow your clients to iterate through your objects without ever getting a peek at how you store your objects.

Problem Iterator Design Pattern Address

The means that the client code iterates over a collection of values should **not depend** on the underlying data structure of the collection.

Merge Two Restaurants


They want to use my Pancake House menu as the breakfast menu and the Diner's menu as the lunch menu. We've agreed on an implementation for the menu items...
... but we can't agree on how to implement our menus. That joker over there used an ArrayList to hold his menu items, and I used an Array. Neither one of us is willing to change our implementations... we just have too much code written that depends on them.
Lou
Mel


The Diner menu has lots of lunch items, while the Pancake House consists of breakfast items. Every menu item has a name, a description, and a price.
Objectville Diner
Vegetarian BLT
(Fakin') Bacon with lettu
whole wheat
2.99
BLT
Bacon with lettuce & tom
Soup of the day
A bowl of the soup of the
a side of potato salad
Hot Dog
A hot dog, with saurkrau
topped with cheese
Steamed Veggies and Bro
A medley of steamed ve
Objectville Pancake House
K&B's Pancake Breakfast
Pancakes with scrambled eggs, and toast
2.99
Regular Pancake Breakfast
Pancakes with fried eggs, sausage
2.99
Blueberry Pancakes
Pancakes made with fresh blueberries, and blueberry syrup
3.49
Waffles
Waffles, with your choice of blueberries or strawberries
3.59

```
public class MenuItem {
    String name;
    String description;
    boolean vegetarian;
    double price;

    public MenuItem(String name,
                    String description,
                    boolean vegetarian,
                    double price)
    {
        this.name = name;
        this.description = description;
        this.vegetarian = vegetarian;
        this.price = price;
    }

    public String getName() {
        return name;
    }

    public String getDescription() {
        return description;
    }

    public double getPrice() {
        return price;
    }

    public boolean isVegetarian() {
        return vegetarian;
    }
}
```

A MenuItem consists of a name, a description, a flag to indicate if the item is vegetarian, and a price. You pass all these values into the constructor to initialize the MenuItem.

These getter methods let you access the fields of the menu item.

Here's Lou's implementation of the Pancake House menu.

```
public class PancakeHouseMenu {
    ArrayList<MenuItem> menuItems;

    public PancakeHouseMenu() {
        menuItems = new ArrayList<MenuItem>();

        addItem("K&B's Pancake Breakfast",
            "Pancakes with scrambled eggs, and toast",
            true,
            2.99);

        addItem("Regular Pancake Breakfast",
            "Pancakes with fried eggs, sausage",
            false,
            2.99);

        addItem("Blueberry Pancakes",
            "Pancakes made with fresh blueberries",
            true,
            3.49);

        addItem("Waffles",
            "Waffles, with your choice of blueberries or strawberries",
            true,
            3.59);
    }

    public void addItem(String name, String description,
                        boolean vegetarian, double price)
    {
        MenuItem menuItem = new MenuItem(name, description, vegetarian, price);
        menuItems.add(menuItem);
    }

    public ArrayList<MenuItem> getMenuItems() {
        return menuItems;
    }

    // other menu methods here
}
```

Lou's using an ArrayList to store his menu items.

Each menu item is added to the ArrayList here, in the constructor.

Each MenuItem has a name, a description, whether or not it's a vegetarian item, and the price.

To add a menu item, Lou creates a new MenuItem object, passing in each argument, and then adds it to the ArrayList.

The getMenuItems() method returns the list of menu items.

Lou has a bunch of other menu code that depends on the ArrayList implementation. He doesn't want to have to rewrite all that code!

And here's Mel's implementation of the Diner menu.

```
public class DinerMenu {
    static final int MAX_ITEMS = 6;
    int numberOfItems = 0;
    MenuItem[] menuItems;

    public DinerMenu() {
        menuItems = new MenuItem[MAX_ITEMS];

        addItem("Vegetarian BLT",
            "(Fakin') Bacon with lettuce & tomato on whole wheat", true, 2.99);
        addItem("BLT",
            "Bacon with lettuce & tomato on whole wheat", false, 2.99);
        addItem("Soup of the day",
            "Soup of the day, with a side of potato salad", false, 3.29);
        addItem("Hotdog",
            "A hot dog, with saurkraut, relish, onions, topped with cheese",
            false, 3.05);
        // a couple of other Diner Menu items added here
    }

    public void addItem(String name, String description,
                         boolean vegetarian, double price)
    {
        MenuItem menuItem = new MenuItem(name, description, vegetarian, price);
        if (numberOfItems >= MAX_ITEMS) {
            System.err.println("Sorry, menu is full!  Can't add item to menu");
        } else {
            menuItems[numberOfItems] = menuItem;
            numberOfItems = numberOfItems + 1;
        }
    }

    public MenuItem[] getMenuItems() {
        return menuItems;
    }

    // other menu methods here
}
```

Mel takes a different approach; he's using an Array so he can control the max size of the menu.

Like Lou, Mel creates his menu items in the constructor, using the addItem() helper method.

addItem() takes all the parameters necessary to create a MenuItem and instantiates one. It also checks to make sure we haven't hit the menu size limit.

Mel specifically wants to keep his menu under a certain size (presumably so he doesn't have to remember too many recipes).

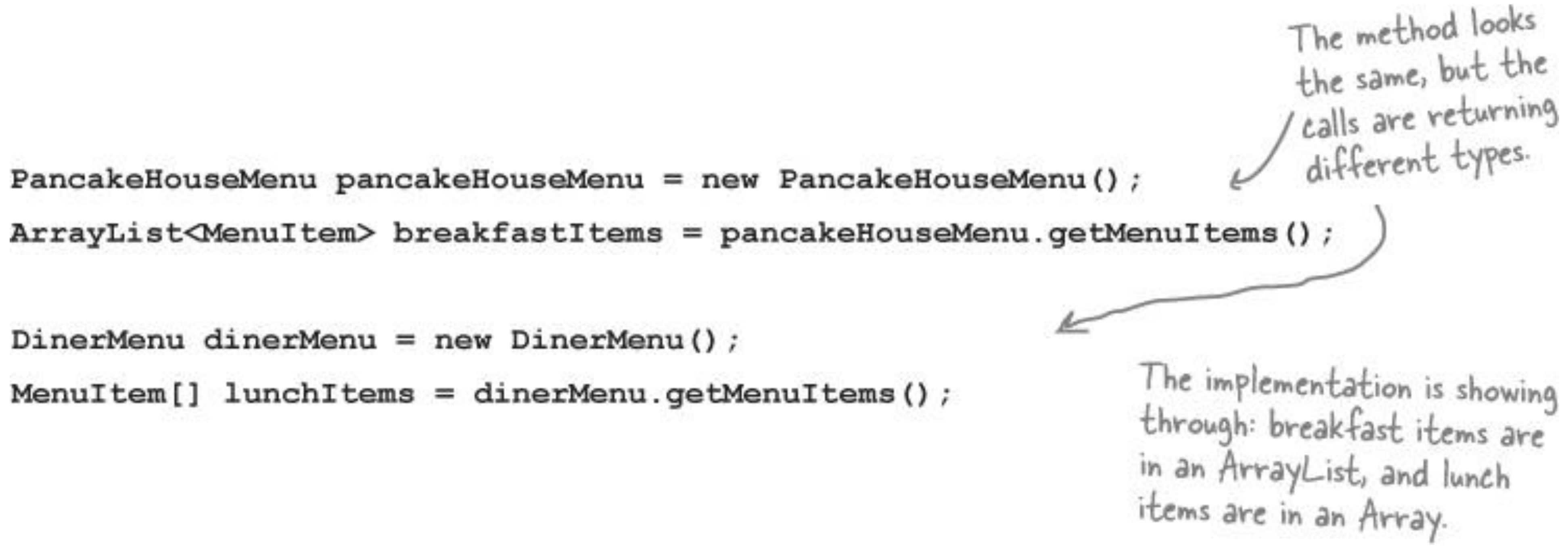
getMenuItems() returns the array of menu items.

Like Lou, Mel has a bunch of code that depends on the implementation of his menu being an Array. He's too busy cooking to rewrite all of this.

What’s the problem with having two different menu representations?

- Try implementing a client that uses the two menus.
- Imagine you have been hired by the new company formed by the merger of the Diner and the Pancake House to create a Java-enabled waitress. The spec for the Java-enabled waitress specifies that she can print a custom menu for customers on demand, and even tell you if a menu item is vegetarian without having to ask the cook—now that’s an innovation!
- Let’s check out the spec, and then step through what it might take to implement her...

- To print all the items on each menu, you'll need to call the getMenuItems() method on the PancakeHouseMenu and the DinerMenu to retrieve their respective menu items. Note that each returns a different type:



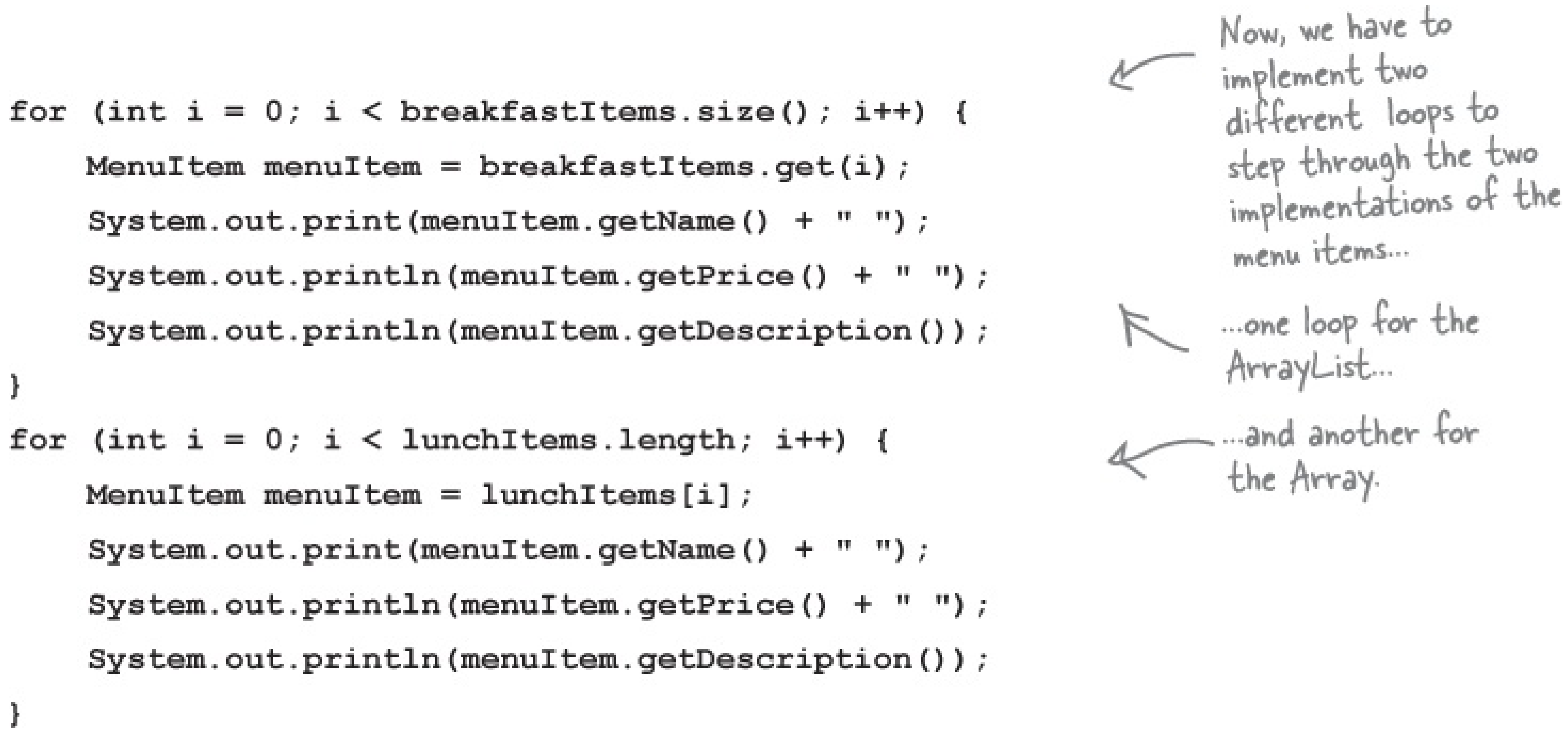
```
PancakeHouseMenu pancakeHouseMenu = new PancakeHouseMenu();
ArrayList<MenuItem> breakfastItems = pancakeHouseMenu.getMenuItems();

DinerMenu dinerMenu = new DinerMenu();
MenuItem[] lunchItems = dinerMenu.getMenuItems();
```

The method looks the same, but the calls are returning different types.

The implementation is showing through: breakfast items are in an ArrayList, and lunch items are in an Array.

- Now, to print out the items from the PancakeHouseMenu, we'll loop through the items on the breakfastItems ArrayList. And to print out the Diner items we'll loop through the Array.



```
for (int i = 0; i < breakfastItems.size(); i++) {
    MenuItem menuItem = breakfastItems.get(i);
    System.out.print(menuItem.getName() + " ");
    System.out.println(menuItem.getPrice() + " ");
    System.out.println(menuItem.getDescription());
}
for (int i = 0; i < lunchItems.length; i++) {
    MenuItem menuItem = lunchItems[i];
    System.out.print(menuItem.getName() + " ");
    System.out.println(menuItem.getPrice() + " ");
    System.out.println(menuItem.getDescription());
}
```

Now, we have to implement two different loops to step through the two implementations of the menu items...

...one loop for the ArrayList...

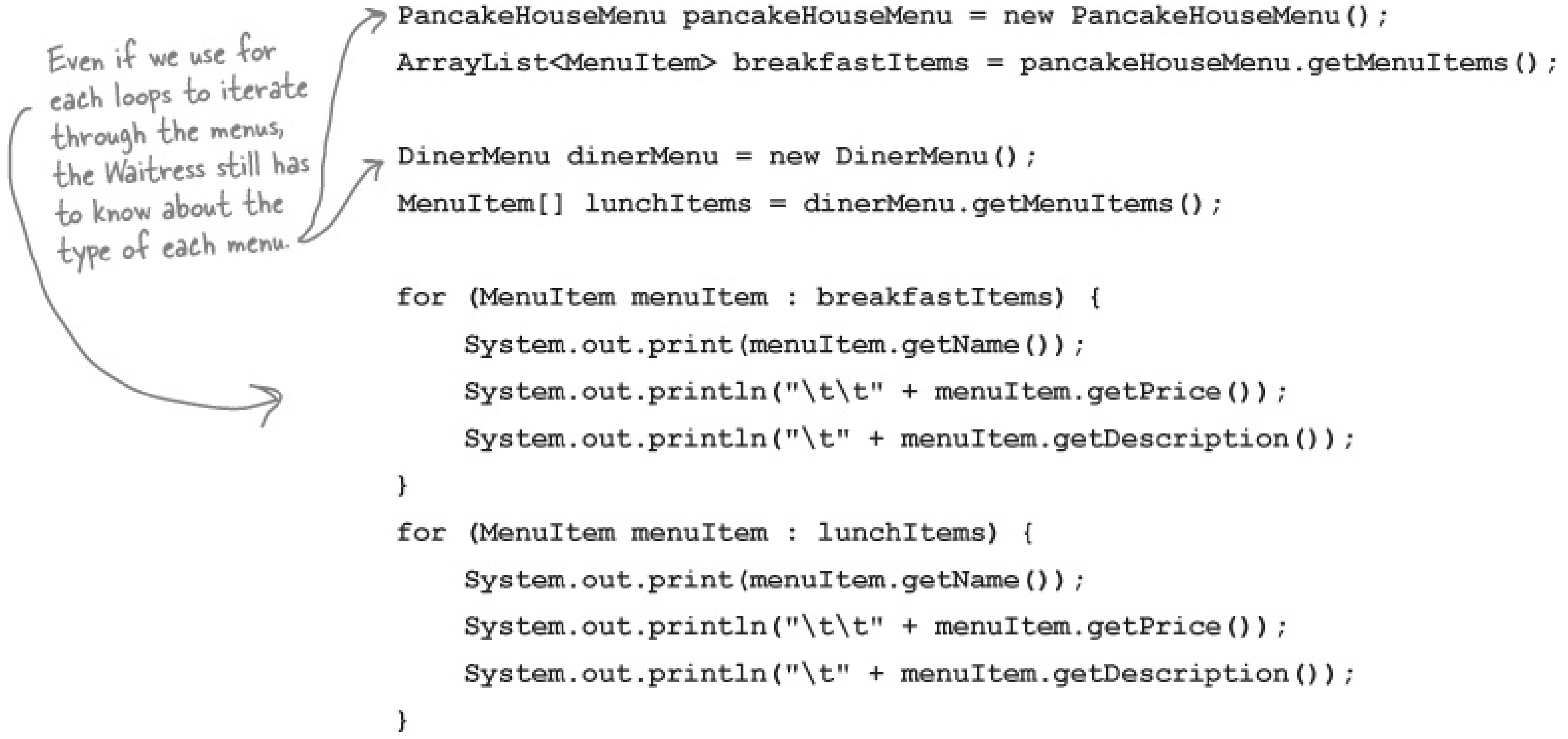
...and another for the Array.

- Implementing every other method in the Waitress is going to be a variation of this theme. We’re always going to need to get both menus and use two loops to iterate through their items.
- If another restaurant with a different implementation is acquired then we’ll have *three* loops.

- Both restaurants don’t want to change their implementations because it would mean rewriting a lot of code that is in each respective menu class.
- If one of them doesn’t give in, then we’re going to have the job of implementing a Waitress that is going to be hard to maintain and extend.
- What can we do?

For each loop?

- Our goal is to decouple the Waitress from the concrete implementations of the menus completely



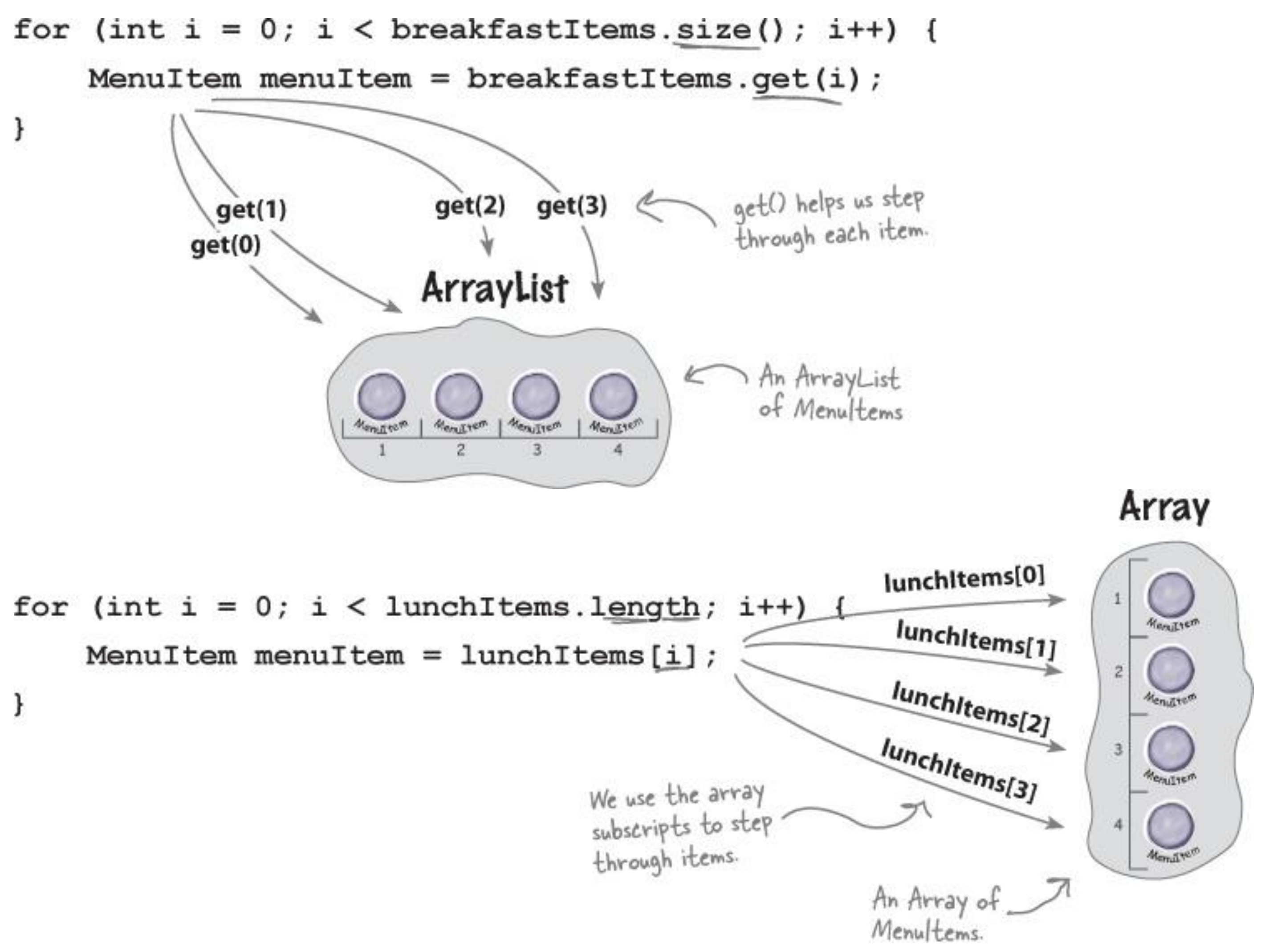
Even if we use for each loops to iterate through the menus, the Waitress still has to know about the type of each menu.

```
PancakeHouseMenu pancakeHouseMenu = new PancakeHouseMenu();
ArrayList<MenuItem> breakfastItems = pancakeHouseMenu.getMenuItems();

DinerMenu dinerMenu = new DinerMenu();
MenuItem[] lunchItems = dinerMenu.getMenuItems();

for (MenuItem menuItem : breakfastItems) {
    System.out.print(menuItem.getName());
    System.out.println("\t\t" + menuItem.getPrice());
    System.out.println("\t" + menuItem.getDescription());
}
for (MenuItem menuItem : lunchItems) {
    System.out.print(menuItem.getName());
    System.out.println("\t\t" + menuItem.getPrice());
    System.out.println("\t" + menuItem.getDescription());
}
```

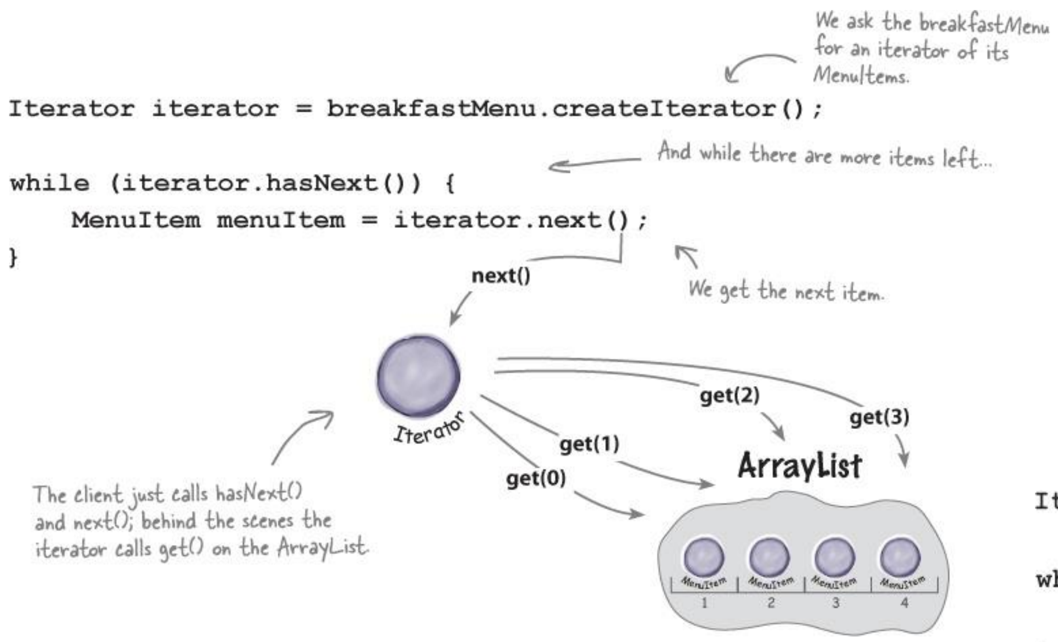
Original Way



```
for (int i = 0; i < breakfastItems.size(); i++) {
    MenuItem menuItem = breakfastItems.get(i);
}
```

```
for (int i = 0; i < lunchItems.length; i++) {
    MenuItem menuItem = lunchItems[i];
}
```

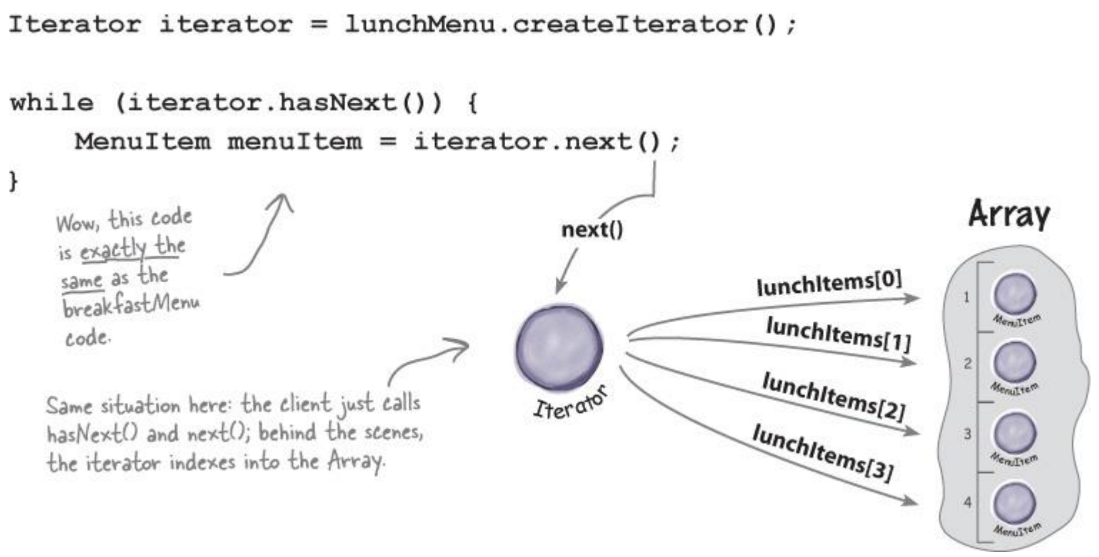
New Way: Encapsulation



```
Iterator iterator = breakfastMenu.createIterator();

while (iterator.hasNext()) {
    MenuItem menuItem = iterator.next();
}
```

We ask the breakfastMenu for an iterator of its MenuItems.
And while there are more items left...
We get the next item.
next()
Iterator
get(0)
get(1)
get(2)
get(3)
ArrayList
MenuItem
1
2
3
4
The client just calls hasNext() and next(); behind the scenes the iterator calls get() on the ArrayList.



```
Iterator iterator = lunchMenu.createIterator();

while (iterator.hasNext()) {
    MenuItem menuItem = iterator.next();
}
```

Wow, this code is exactly the same as the breakfastMenu code.
next()
Iterator
Same situation here: the client just calls hasNext() and next(); behind the scenes, the iterator indexes into the Array.
Array
lunchItems[0]
lunchItems[1]
lunchItems[2]
lunchItems[3]
MenuItem
1
2
3
4

The Iterator Design Pattern

Intent

To provide a way to access the elements of an aggregate object **sequentially without exposing its underlying representation**.

The Iterator Design Pattern

Motivation

An aggregate object (e.g., List, Set, Map) should allow access to its elements **without exposing its internal structure**.

An aggregate object might need to be **traversed over in different ways**.

Do not want to add various traversals to the aggregate object's interface, since it would acquire too much responsibility.

Desire **more than one traversal on the same aggregate object at the same time.**

The Iterator Design Pattern

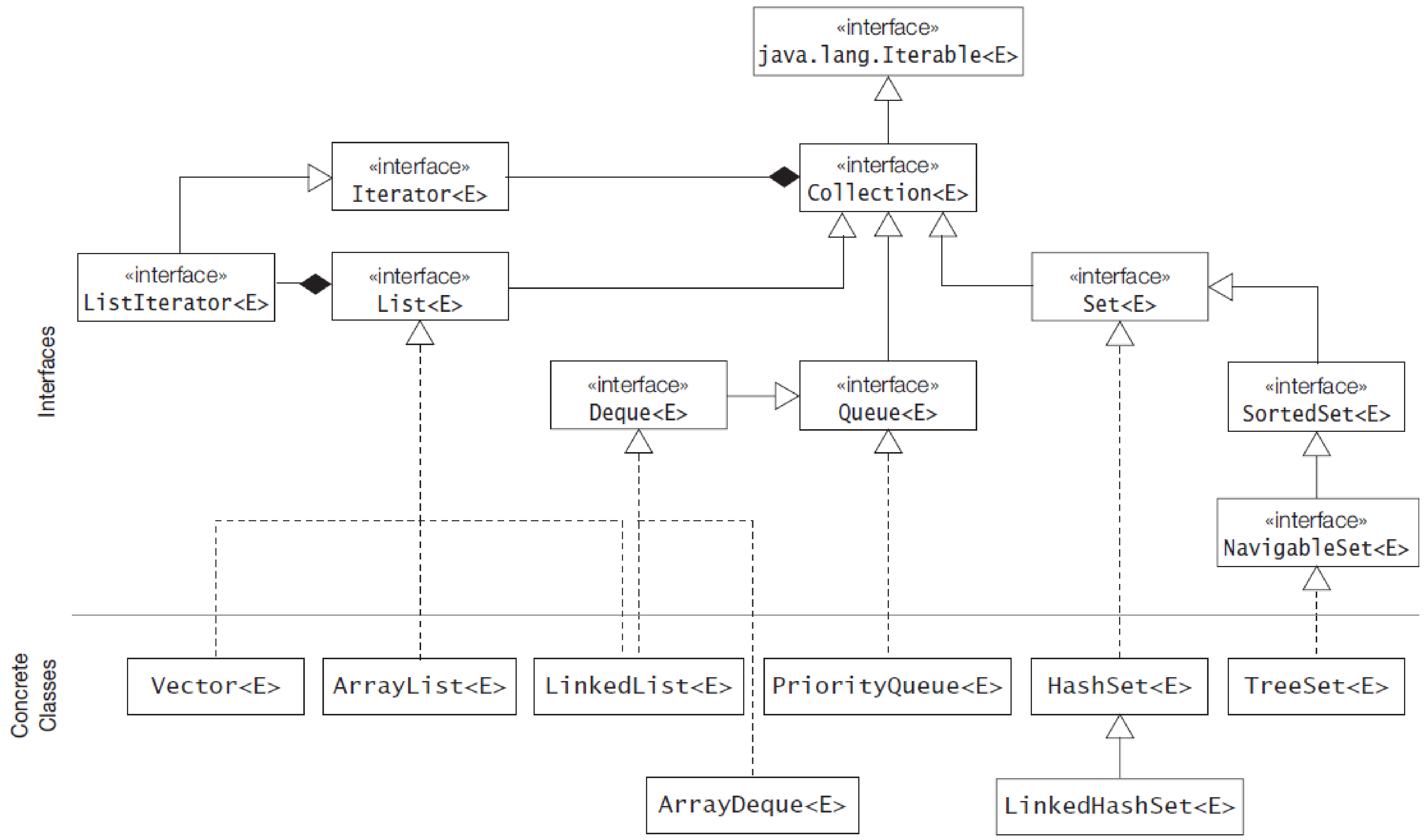
The Iterator pattern takes responsibility for **access** and **traversal** of a given aggregate object, and places in an associated Iterator object. The Iterator object keeps **track** of the current element, and knows which elements have already been traversed.

In order to instantiate a given Iterator, must instantiate aggregate object to traverse.

The Iterator Design Pattern

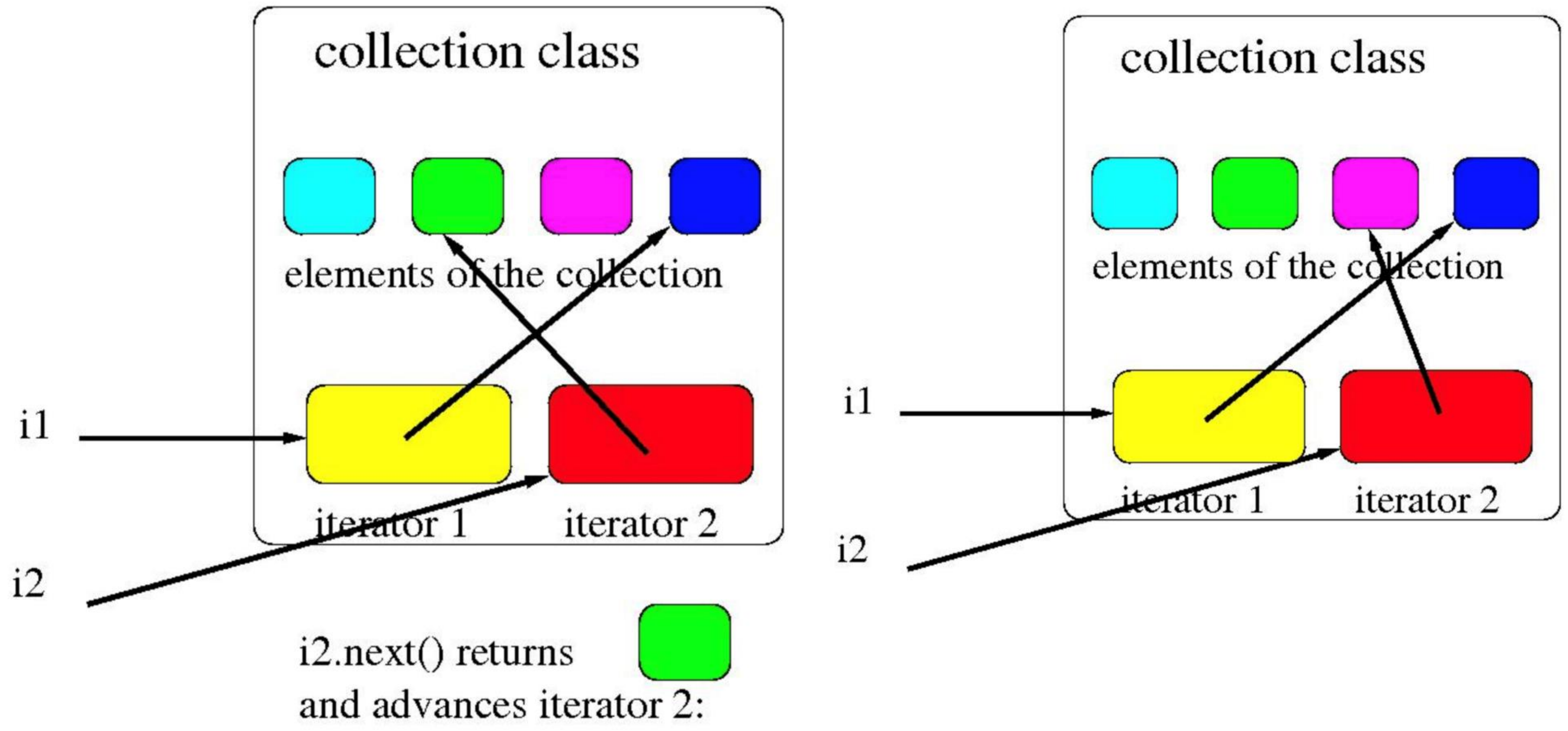
Applicability

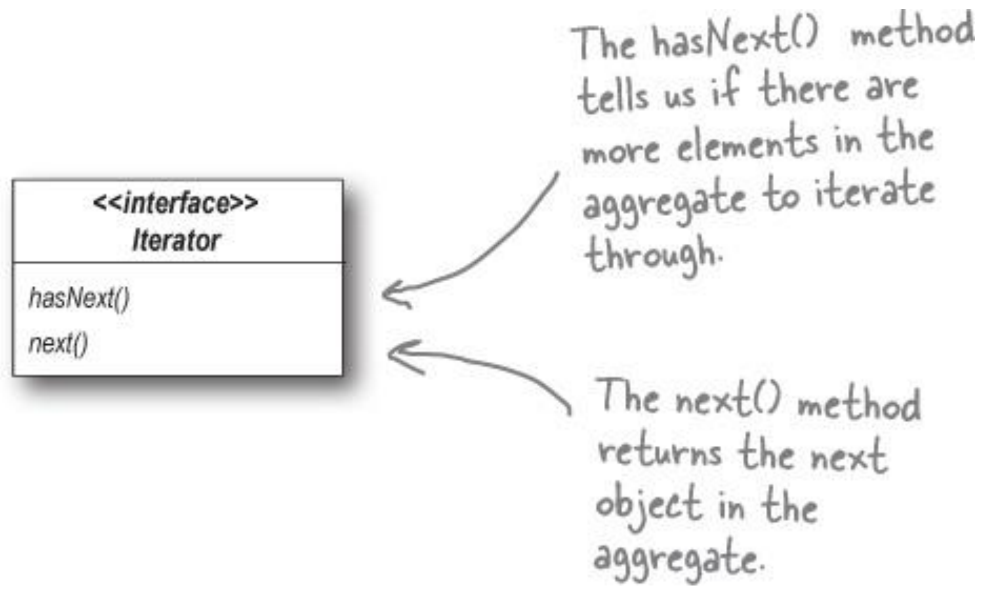
- To access an aggregate object’s contents **without exposing its internal representation**.
- To support **multiple traversals** of aggregate objects.
- To provide a uniform interface for traversing different aggregate structures (i.e., “polymorphic iteration”).



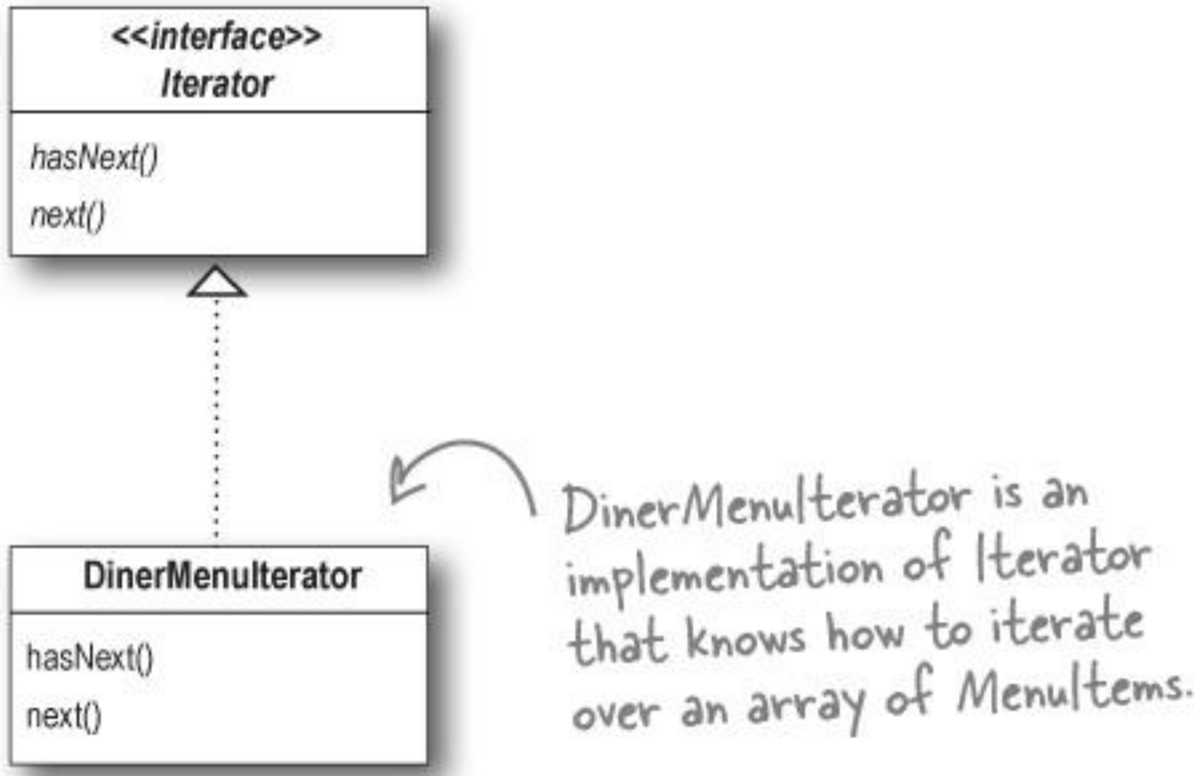
Java Collections Framework

Multiple iterators





<<interface>>
Iterator
hasNext()
next()
The hasNext() method tells us if there are more elements in the aggregate to iterate through.
The next() method returns the next object in the aggregate.



<<interface>>
Iterator
hasNext()
next()
DinerMenuIterator
hasNext()
next()
DinerMenuIterator is an implementation of Iterator that knows how to iterate over an array of MenuItems.

Here are our two methods:

```
public interface Iterator {
    boolean hasNext();
    Object next();
}
```

The hasNext() method returns a boolean indicating whether or not there are more elements to iterate over...

...and the next() method returns the next element.

```
public class DinerMenuIterator implements Iterator {
    MenuItem[] items;
    int position = 0;

    public DinerMenuIterator(MenuItem[] items) {
        this.items = items;
    }

    public MenuItem next() {
        MenuItem menuItem = items[position];
        position = position + 1;
        return menuItem;
    }

    public boolean hasNext() {
        if (position >= items.length || items[position] == null) {
            return false;
        } else {
            return true;
        }
    }
}
```

We implement the Iterator interface.

position maintains the current position of the iteration over the array.

The constructor takes the array of menu items we are going to iterate over.

The next() method returns the next item in the array and increments the position.

The hasNext() method checks to see if we've seen all the elements of the array and returns true if there are more to iterate through.

Because the diner chef went ahead and allocated a max sized array, we need to check not only if we are at the end of the array, but also if the next item is null, which indicates there are no more items.

```
public class DinerMenu {
    static final int MAX_ITEMS = 6;
    int numberOfItems = 0;
    MenuItem[] menuItems;

    // constructor here

    // addItem here

    ~~public MenuItem[] getMenuItems() {~~
        ~~return menuItems;~~
    ~~}~~

    public Iterator createIterator() {
        return new DinerMenuIterator(menuItems);
    }

    // other menu methods here
}
```

We're not going to need the getMenuItems() method anymore and in fact, we don't want it because it exposes our internal implementation!

Here's the createIterator() method. It creates a DinerMenuIterator from the menuItems array and returns it to the client.

We're returning the Iterator interface. The client doesn't need to know how the menuItems are maintained in the DinerMenu, nor does it need to know how the DinerMenuIterator is implemented. It just needs to use the iterators to step through the items in the menu.

```
public class MenuTestDrive {
    public static void main(String args[]) {
        PancakeHouseMenu pancakeHouseMenu = new PancakeHouseMenu();
        DinerMenu dinerMenu = new DinerMenu();

        Waitress waitress = new Waitress(pancakeHouseMenu, dinerMenu);

        waitress.printMenu();
    }
}
```

First we create the new menus.

Then we create a Waitress and pass her the menus.

Then we print them.

```
:lass Waitress {
:akeHouseMenu pancakeHouseMenu;
:rMenu dinerMenu;

  public Waitress(PancakeHouseMenu pancakeHouseMenu, DinerMenu dinerMenu) {
      this.pancakeHouseMenu = pancakeHouseMenu;
      this.dinerMenu = dinerMenu;
  }

  public void printMenu() {
      Iterator pancakeIterator = pancakeHouseMenu.createIterator();
      Iterator dinerIterator = dinerMenu.createIterator();

      System.out.println("MENU\n----\nBREAKFAST");
      printMenu(pancakeIterator);
      System.out.println("\nLUNCH");
      printMenu(dinerIterator);
  }

  private void printMenu(Iterator iterator) {
      while (iterator.hasNext()) {
          MenuItem menuItem = iterator.next();
          System.out.print(menuItem.getName() + ", ");
          System.out.print(menuItem.getPrice() + " -- ");
          System.out.println(menuItem.getDescription());
      }
  }

  // other methods here
}
```

In the constructor the Waitress takes the two menus.

The printMenu() method now creates two iterators, one for each menu.

And then calls the overloaded printMenu() with each iterator.

Test if there are any more items.

Get the next item.

The overloaded printMenu() method uses the Iterator to step through the menu items and print them.

Use the item to get name, price, and description and print them.

Note that we're down to one loop.

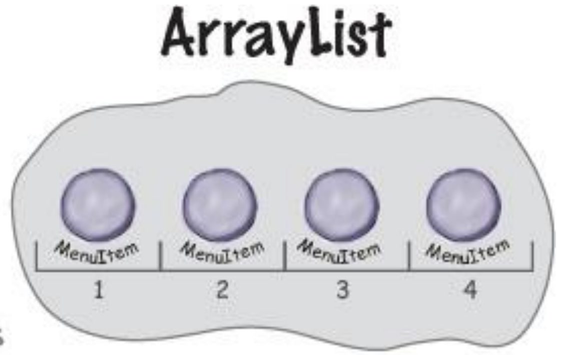
Woohoo! No code changes other than adding the createIterator() method.
Veggie burger

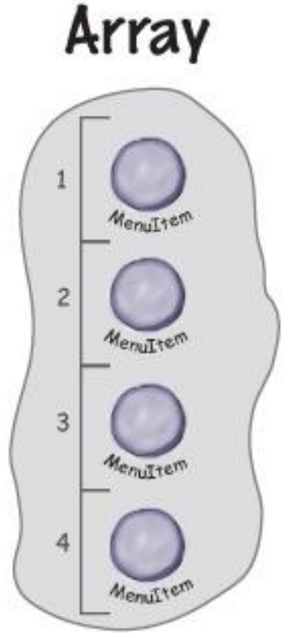
Hard to Maintain Waitress Implementation	New, Hip Waitress Powered by Iterator
The Menus are not well encapsulated; we can see the Diner is using an ArrayList and the Pancake House an Array.	The Menu implementations are now encapsulated. The Waitress has no idea how the Menus hold their collection of menu items.
We need two loops to iterate through the MenuItems.	All we need is a loop that polymorphically handles any collection of items as long as it implements Iterator.
The Waitress is bound to concrete classes (MenuItem[] and ArrayList).	The Waitress now uses an interface (Iterator).
The Waitress is bound to two different concrete Menu classes, despite their interfaces being almost identical.	The Menu interfaces are now exactly the same and, uh oh, we still don't have a common interface, which means the Waitress is still bound to two concrete Menu classes. We'd better fix that.

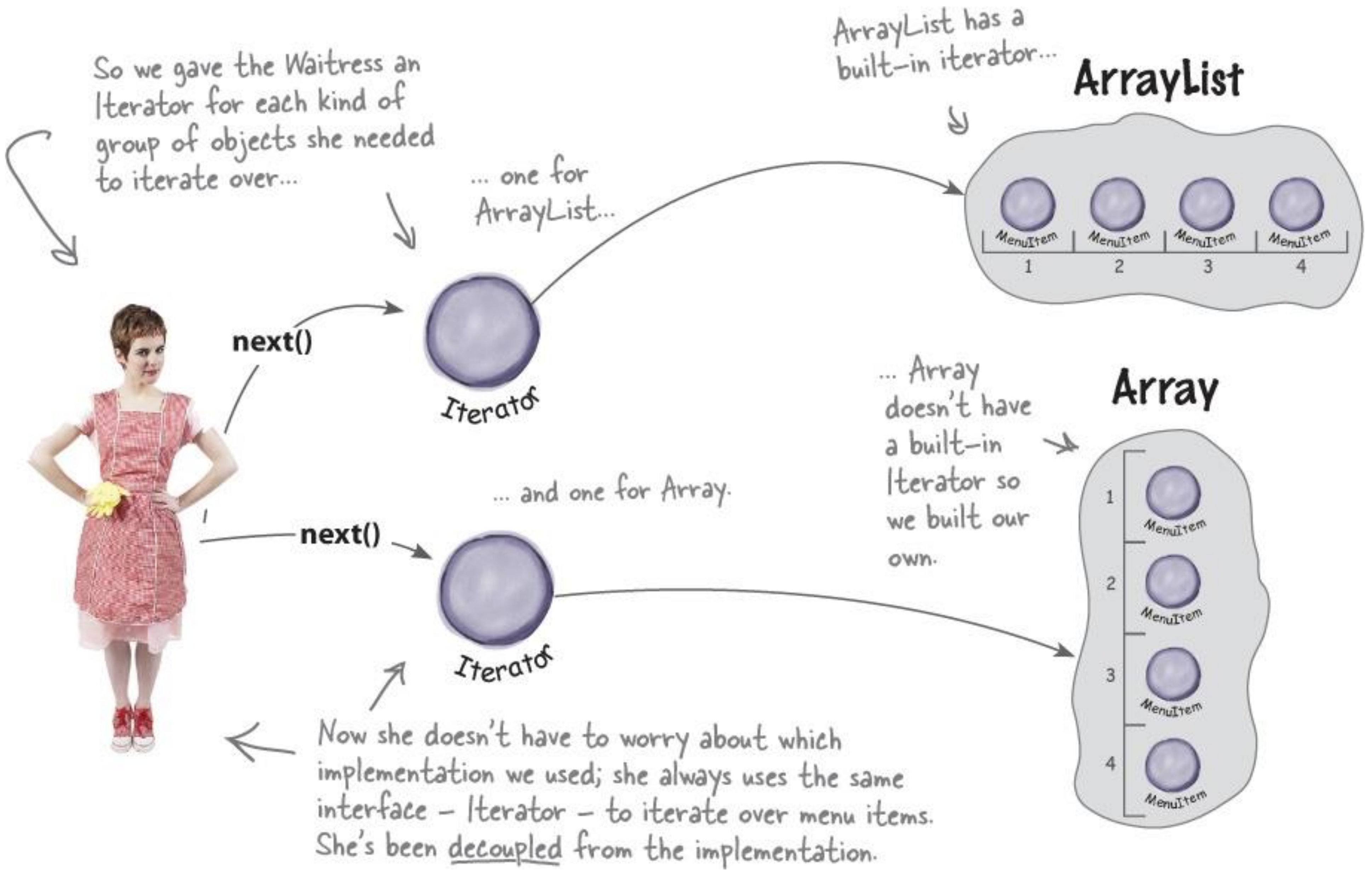
We wanted to give the Waitress an easy way to iterate over menu items...

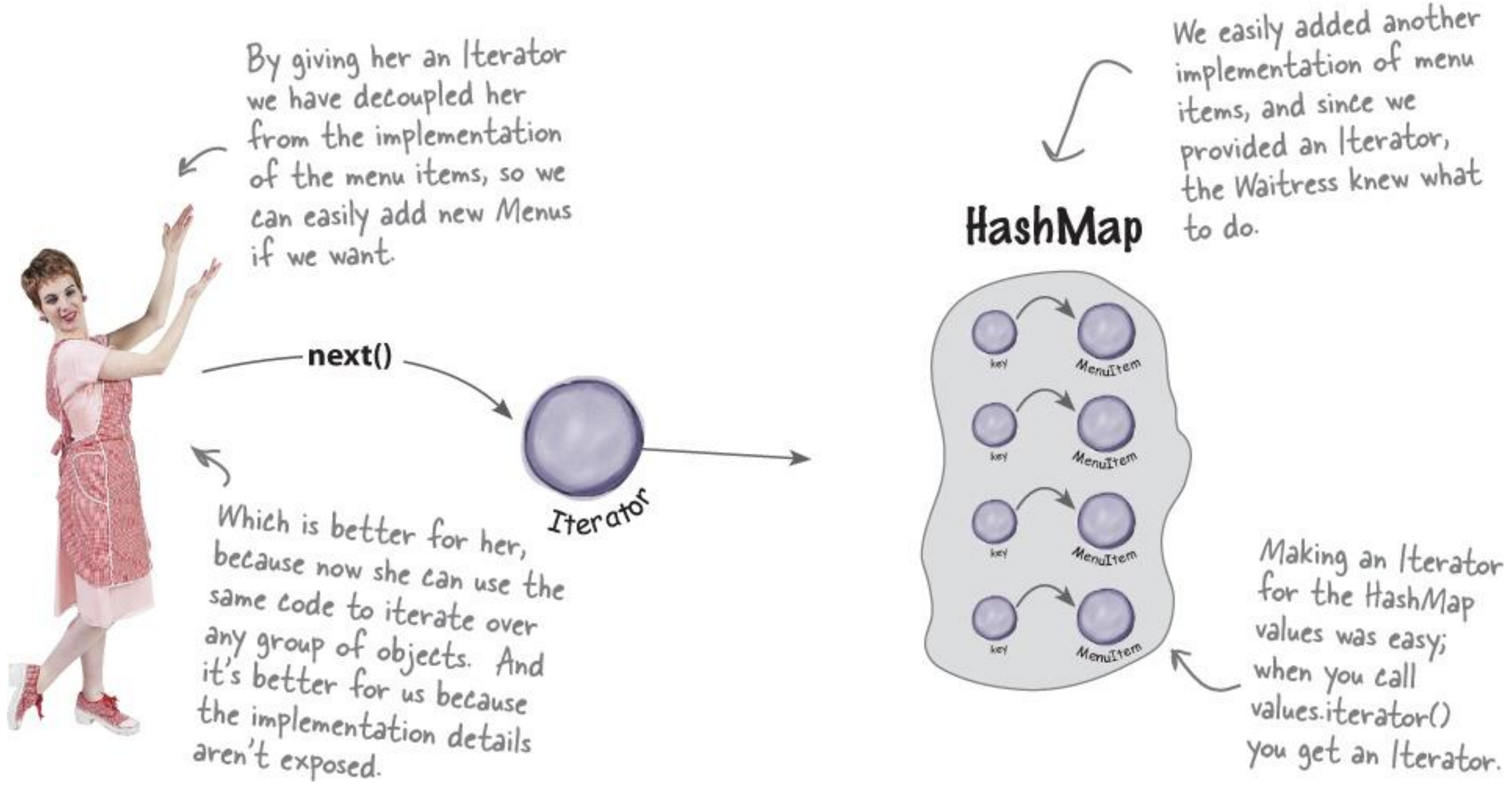
... and we didn't want her to know about how the menu items are implemented.

Our menu items had two different implementations and two different interfaces for iterating.






So we gave the Waitress an Iterator for each kind of group of objects she needed to iterate over...
... one for ArrayList...
ArrayList has a built-in iterator...
ArrayList
MenuItem
1
MenuItem
2
MenuItem
3
MenuItem
4
next()
Iterator
... Array doesn't have a built-in Iterator so we built our own.
Array
... and one for Array.
next()
Iterator
1
MenuItem
2
MenuItem
3
MenuItem
4
MenuItem
Now she doesn't have to worry about which implementation we used; she always uses the same interface – Iterator – to iterate over menu items. She's been decoupled from the implementation.



By giving her an Iterator we have decoupled her from the implementation of the menu items, so we can easily add new Menus if we want.

Which is better for her, because now she can use the same code to iterate over any group of objects. And it's better for us because the implementation details aren't exposed.

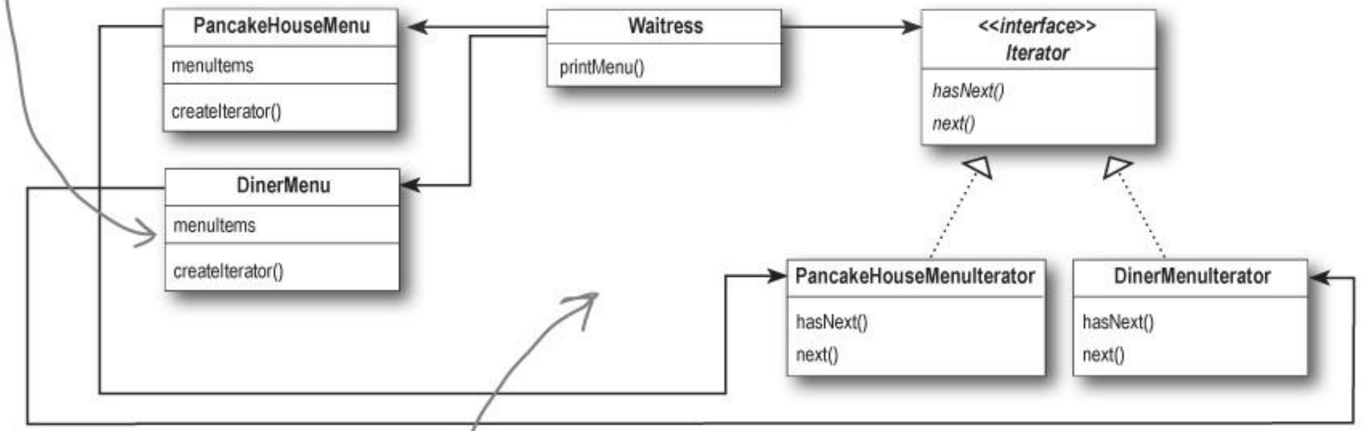
We easily added another implementation of menu items, and since we provided an Iterator, the Waitress knew what to do.

Making an Iterator for the HashMap values was easy; when you call values.iterator() you get an Iterator.

These two menus implement the same exact set of methods, but they aren't implementing the same interface. We're going to fix this and free the Waitress from any dependencies on concrete Menus.

The Iterator allows the Waitress to be decoupled from the actual implementation of the concrete classes. She doesn't need to know if a Menu is implemented with an Array, an ArrayList, or with Post-it® notes. All she cares is that she can get an Iterator to do her iterating.

We're now using a common Iterator interface and we've implemented two concrete classes.

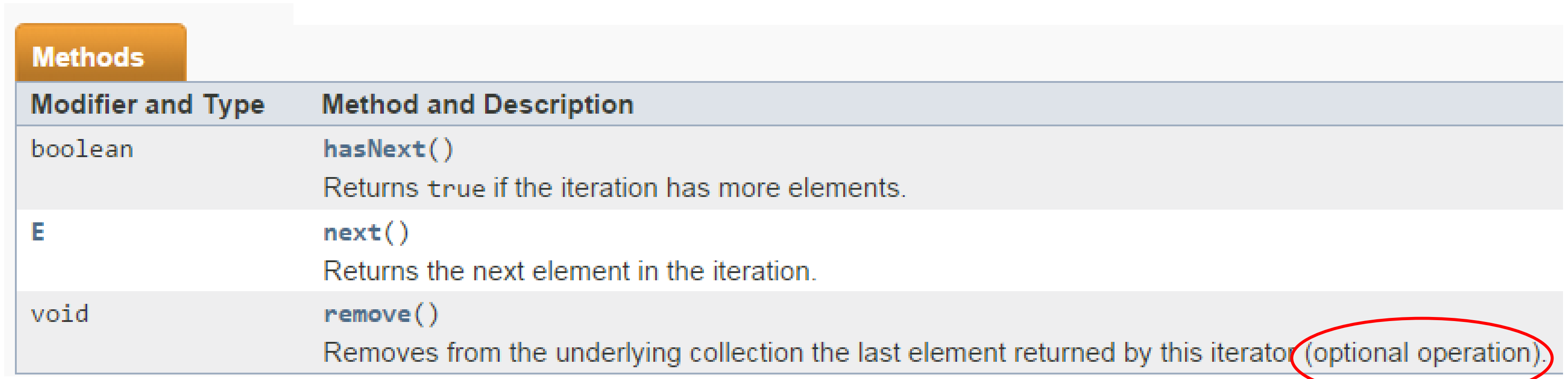


Note that the iterator gives us a way to step through the elements of an aggregate without forcing the aggregate to clutter its own interface with a bunch of methods to support traversal of its elements. It also allows the implementation of the iterator to live outside of the aggregate; in other words, we've encapsulated the interation.

PancakeHouseMenu and DinerMenu implement the new createIterator() method; they are responsible for creating the iterator for their respective menu items' implementations.

The Iterator Interface in Java

Interface Iterator<E>

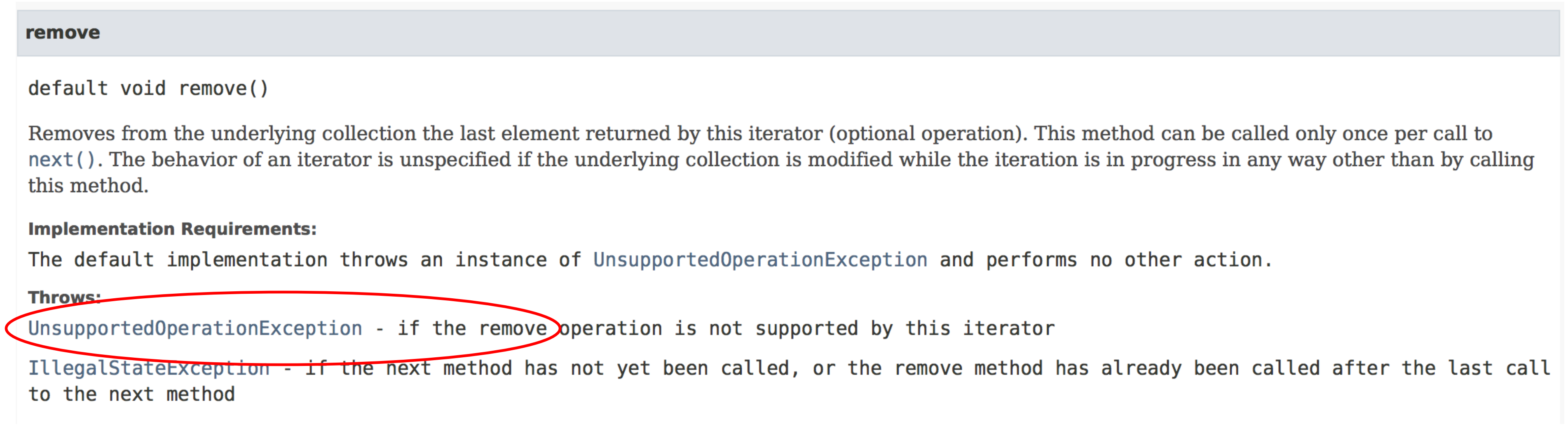


Methods

Modifier and Type	Method and Description
boolean	`hasNext()` Returns `true` if the iteration has more elements.
E	`next()` Returns the next element in the iteration.
void	`remove()` Removes from the underlying collection the last element returned by this iterator (optional operation).

What does "optional operation" mean for an interface?

The Iterator Interface in Java



remove

```
default void remove()
```

Removes from the underlying collection the last element returned by this iterator (optional operation). This method can be called only once per call to `next()`. The behavior of an iterator is unspecified if the underlying collection is modified while the iteration is in progress in any way other than by calling this method.

Implementation Requirements:

```
The default implementation throws an instance of UnsupportedOperationException and performs no other action.
```

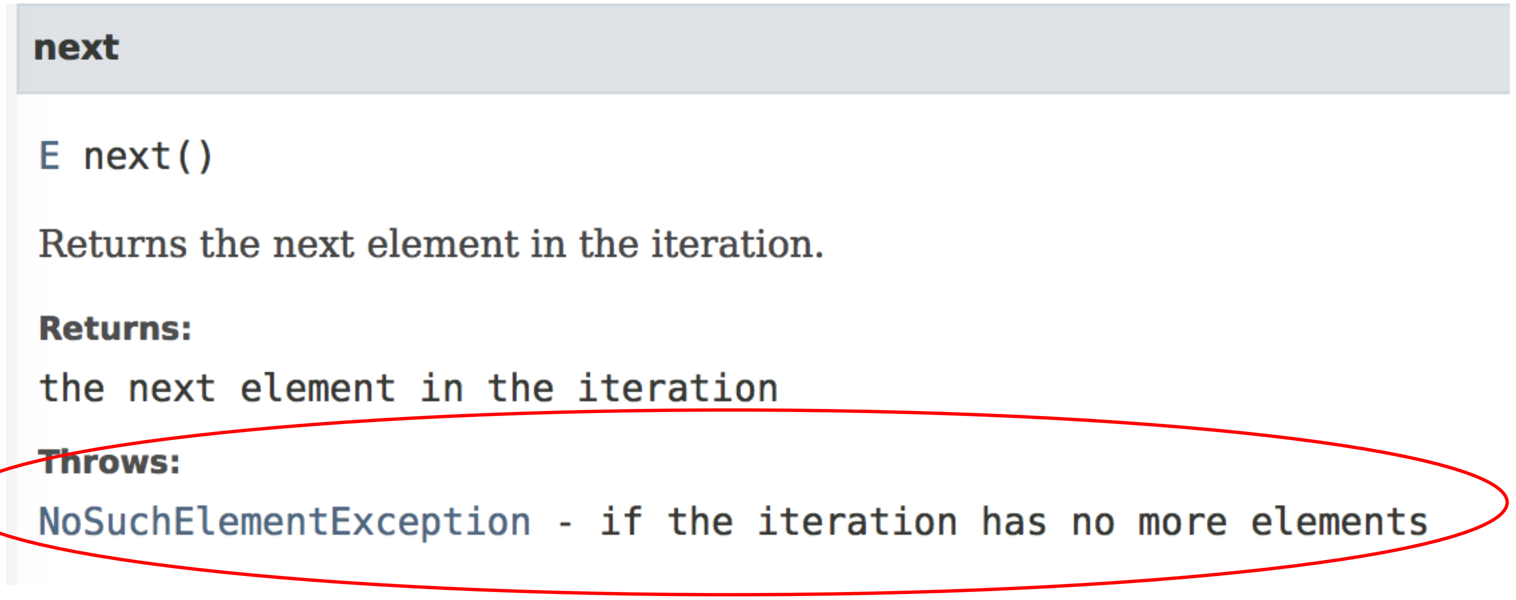
Throws:

```
UnsupportedOperationException - if the remove operation is not supported by this iterator
IllegalStateException - if the next method has not yet been called, or the remove method has already been called after the last call
to the next method
```

The method can be implemented to throw an UnsupportedOperationException if not appropriate for the collection iterating over.

NOTE: The remove method example in the textbook towards the bottom of page that is implemented as an empty method should be written to throw this exception.

The Iterator Interface in Java



next

```
E next()
```

Returns the next element in the iteration.

Returns:

```
the next element in the iteration
```

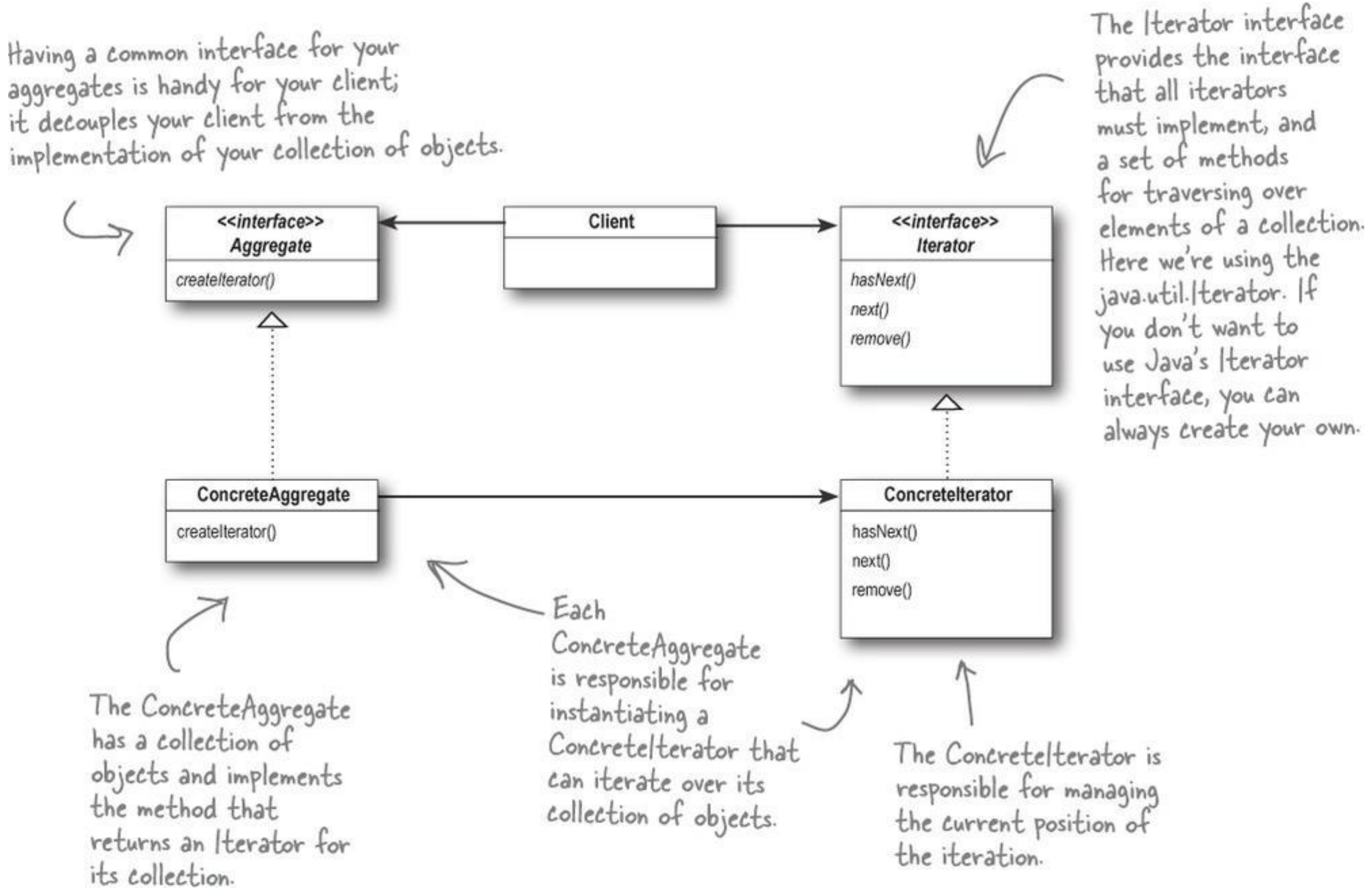
Throws:

```
NoSuchElementException - if the iteration has no more elements
```

It is NOT required to call hasNext() every time before calling next().

What if someone keep calling next() without checking hasNext()?

Need to throw **NoSuchElementException**



Having a common interface for your aggregates is handy for your client; it decouples your client from the implementation of your collection of objects.

The Iterator interface provides the interface that all iterators must implement, and a set of methods for traversing over elements of a collection. Here we're using the java.util.Iterator. If you don't want to use Java's Iterator interface, you can always create your own.

The ConcreteAggregate has a collection of objects and implements the method that returns an Iterator for its collection.

Each ConcreteAggregate is responsible for instantiating a ConcreteIterator that can iterate over its collection of objects.

The ConcreteIterator is responsible for managing the current position of the iteration.