

THE ITERATOR PATTERN

Chandan R. Rupakheti

Week 7-1

Today

How to allow access to members of an aggregate without exposing the aggregate's internal representation

Use polymorphism to support use of multiple aggregates without breaking the client code

You bet I keep my collections well encapsulated!



Breaking News: Objectville Diner and Objectville Pancake House Merge



That's great news! Now we can get those delicious pancake breakfasts at the Pancake House and those yummy lunches at the Diner all in one place. But, there seems to be a slight problem...

Check out the Menu Items

At least Lou and Mel agree on the implementation of the MenuItem. Let's check out the items on each menu, and also take a look at the implementation.

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.

<i>Objectville Diner</i>		
Vegetarian BLT		
(Fakin') Bacon with lettuce and tomato on whole wheat		2.99
BLT		
Bacon with lettuce & tomato		
Soup of the day		
A bowl of the soup of the day with a side of potato salad		
Hot Dog		
A hot dog, with saurkraut and mustard, topped with cheese		
Steamed Veggies and Broccoli		
A medley of steamed vegetables		
<i>Objectville Pancake House</i>		
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

Implementation of MenuItem

```
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.

Lou's Code

I used an ArrayList so I can easily expand my menu.



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!

Mel's Code



Haah! An ArrayList... I used a REAL Array so I can control the maximum size of my menu.

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

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

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

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

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

```
    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
```

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.

```
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;
    }
}
```

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

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

getMenuItems() returns the array of menu items.

```
// other menu methods here
```

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.

```
}
```


A Java-Enabled Waitress Spec.

Java-Enabled Waitress: code-name "Alice"

```
printMenu()  
- prints every item on the menu  
  
printBreakfastMenu()  
- prints just breakfast items  
  
printLunchMenu()  
- prints just lunch items  
  
printVegetarianMenu()  
- prints all vegetarian menu items  
  
isItemVegetarian(name)  
- given the name of an item, returns true  
  if the item is vegetarian, otherwise,  
  returns false
```



The Waitress is getting
Java-enabled.

The spec for
the Waitress

Implementation of printMenu()

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

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

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

The implementation is showing through: breakfast items are in an ArrayList, and lunch items are in an 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());
}
```

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

```
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());
}
```

...one loop for the ArrayList...

...and another for the Array.

What now?

Would really be nice if we could find a way to allow them to implement the same interface for their menus

That way we can minimize the concrete references in the Waitress code and also hopefully get rid of the multiple loops required to iterate over both menus



Wait, aren't you making this a lot more complicated than it needs to be? If we use for each to loop, then the way we loop is exactly the same for both menus.

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

Yes, but we've got two different implementations of the menus, and the Waitress has to know how each kind of menu is implemented.

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

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

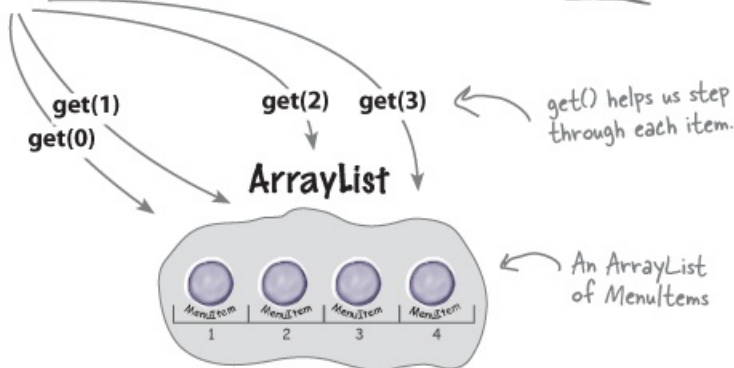
for (MenuItem menuItem : breakfastItems) {
    System.out.print(menuItem.getName());
    System.out.println("\t\t" + menuItem.getPrice());
    System.out.println("\t" + menuItem.getDescription());
}

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

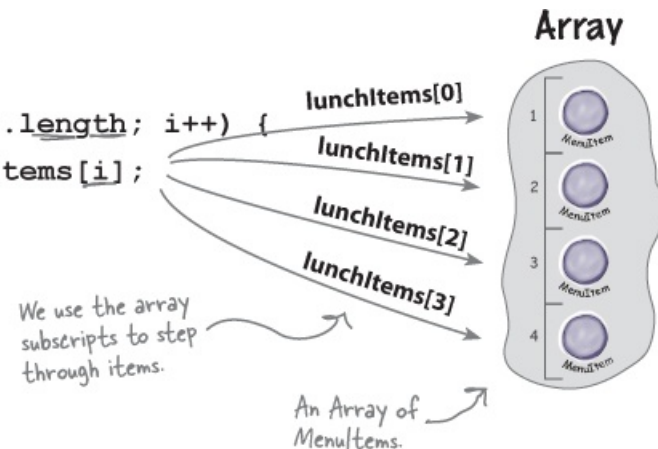
Can we encapsulate the iteration? 1/3

If we've learned one thing in this class, it's encapsulate what varies!

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

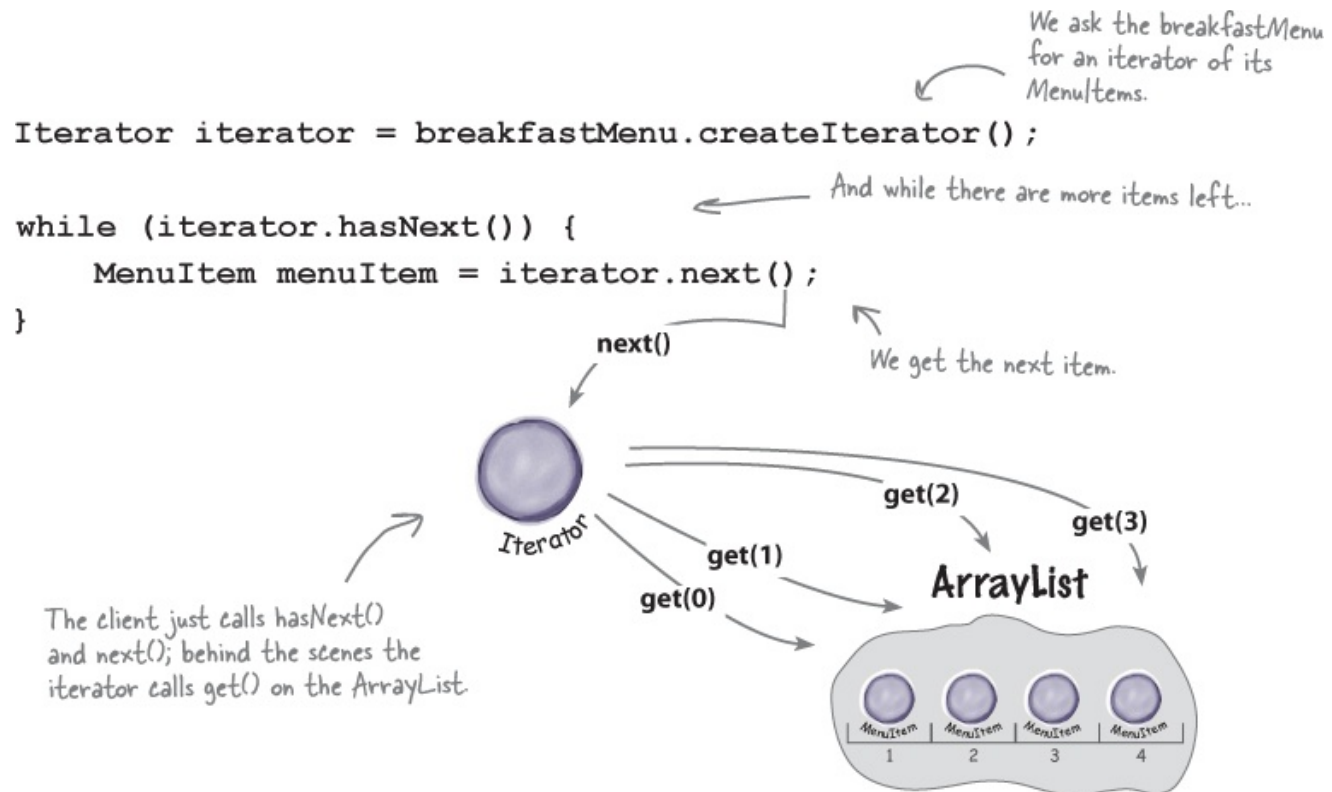


```
for (int i = 0; i < lunchItems.length; i++) {
    MenuItem menuItem = lunchItems[i];
}
```



Can we encapsulate the iteration? 2/3

Now what if we create an object, let's call it an Iterator, that encapsulates the way we iterate through a collection of objects? Let's try this on the ArrayList



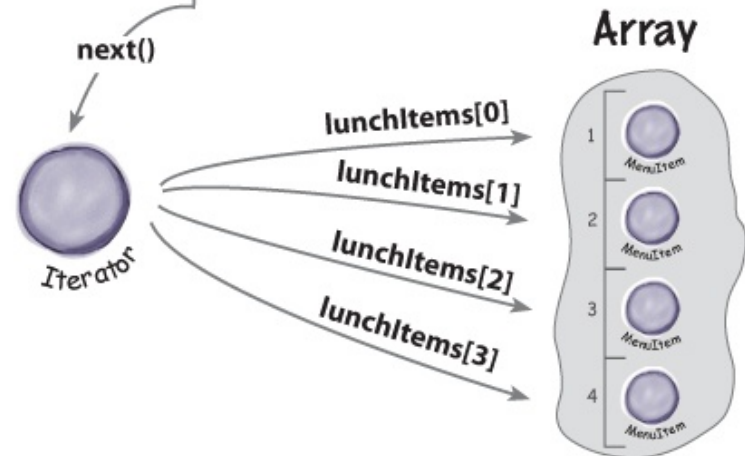
Can we encapsulate the iteration? 3/3

Let's try that on the Array too

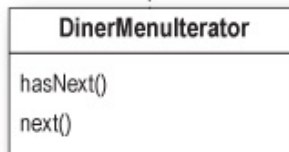
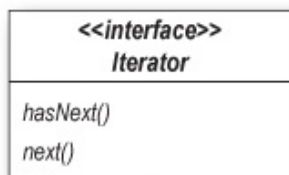
```
Iterator iterator = lunchMenu.createIterator();  
  
while (iterator.hasNext()) {  
    MenuItem menuItem = iterator.next();  
}
```

Wow, this code
is exactly the
same as the
breakfastMenu
code.

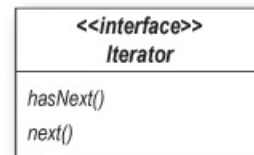
Same situation here: the client just calls
hasNext() and next(); behind the scenes,
the iterator indexes into the Array.



Meet the Iterator Pattern



DinerMenuIterator is an implementation of Iterator that knows how to iterate over an array of MenuItem's.



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.

When we say **COLLECTION** we just mean a group of objects. They might be stored in very different data structures like lists, arrays, or hashmaps, but they're still collections. We also sometimes call these **AGGREGATES**.



Adding an Iterator to DinnerMenu

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

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

We implement the
Iterator interface.

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

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

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

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

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

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

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.

Reworking the Diner Menu with Iterator

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

```
    // constructor here
```

```
    // addItem here
```

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

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

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

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

```
    // other menu methods here  
}
```

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.

Fixing up the Waitress Code

```

public class Waitress {
    PancakeHouseMenu pancakeHouseMenu;
    DinerMenu 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.

New and improved with Iterator.



Testing our code

```
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.

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



Veggie burger

```
File Edit Window Help GreenEggs&Ham
% java DinerMenuTestDrive
```

```
MENU
```

```
----
```

```
BREAKFAST
```

```
K&B's Pancake Breakfast, 2.99 -- Pancakes with scrambled eggs, and toast
```

```
Regular Pancake Breakfast, 2.99 -- Pancakes with fried eggs, sausage
```

```
Blueberry Pancakes, 3.49 -- Pancakes made with fresh blueberries
```

```
Waffles, 3.59 -- Waffles, with your choice of blueberries or strawberries
```

```
LUNCH
```

```
Vegetarian BLT, 2.99 -- (Fakin') Bacon with lettuce & tomato on whole wheat
```

```
BLT, 2.99 -- Bacon with lettuce & tomato on whole wheat
```

```
Soup of the day, 3.29 -- Soup of the day, with a side of potato salad
```

```
Hotdog, 3.05 -- A hot dog, with saurkraut, relish, onions, topped with cheese
```

```
Steamed Veggies and Brown Rice, 3.99 -- Steamed vegetables over brown rice
```

```
Pasta, 3.89 -- Spaghetti with Marinara Sauce, and a slice of sourdough bread
```

```
%
```

First we iterate through the pancake menu.

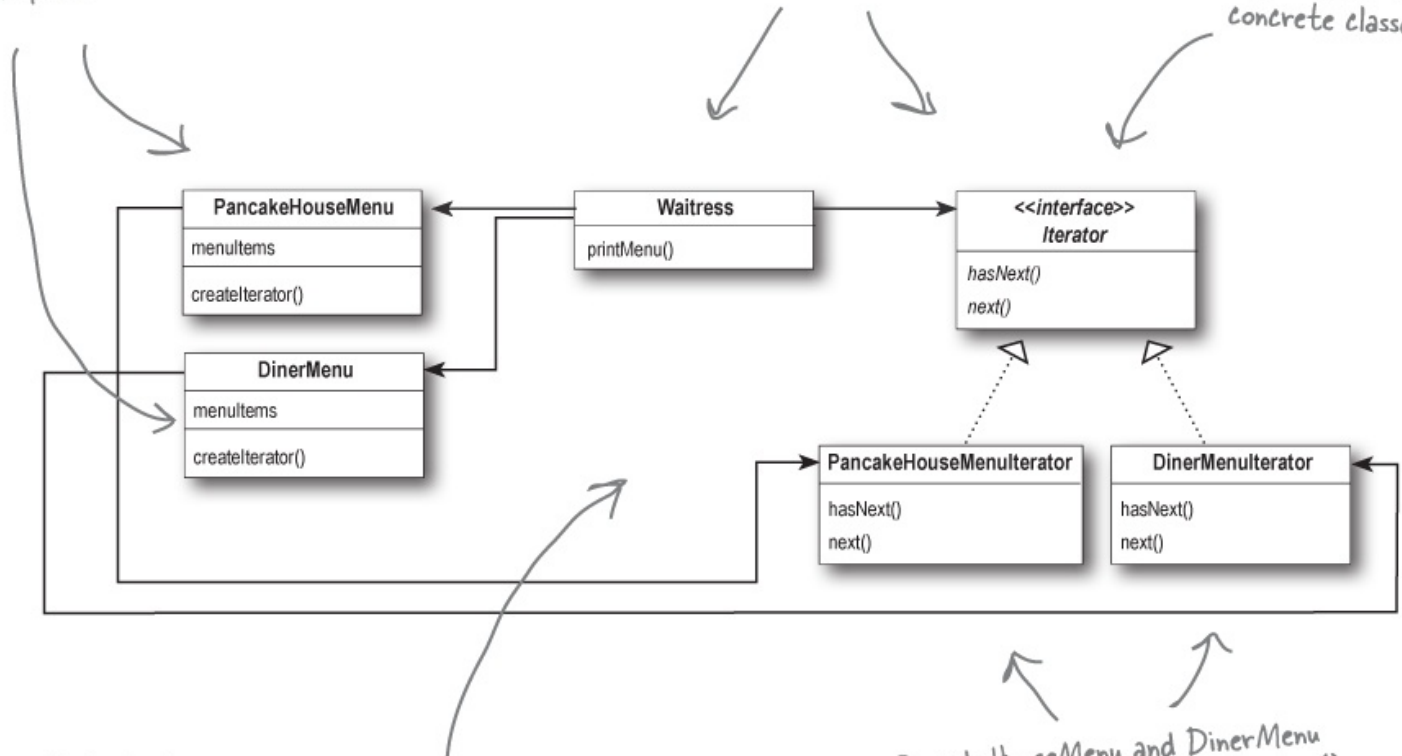
And then the lunch menu, all with the same iteration code.

What
we
have
so far
...

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.



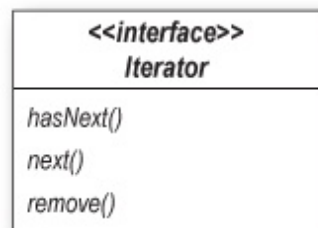
Note that the iterator gives us a way to step through the elements of an aggregate without forcing the aggregate to clutter its own interface with a bunch of methods to support traversal of its elements. It also allows the implementation of the iterator to live outside of the aggregate; in other words,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

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

Making some more improvements ...

Chandan, why are you not using Java Collection's Iterator?

So that you could see how to build an iterator from scratch. Now that we've done that, we're going to switch to using the Java Iterator interface



← This looks just like our previous definition.

← Except we have an additional method that allows us to remove the last item returned by the `next()` method from the aggregate.

Cleaning things up with java.util.Iterator

```
public Iterator<MenuItem> createIterator() {
    return menuItems.iterator();
}
```

Instead of creating our own iterator now, we just call the iterator() method on the menuItems ArrayList.

```
import java.util.Iterator;
```

First we import java.util.Iterator, the interface we're going to implement.

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

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

```
    public MenuItem next() {
        //implementation here
    }
```

None of our current implementation changes...

```
    public boolean hasNext() {
        //implementation here
    }
```

...but we do need to implement remove(). Here, because the chef is using a fixed-size Array, we just shift all the elements up one when remove() is called.

```
    public void remove() {
        if (position <= 0) {
            throw new IllegalStateException
                ("You can't remove an item until you've done at least one next()");
        }
        if (list[position-1] != null) {
            for (int i = position-1; i < (list.length-1); i++) {
                list[i] = list[i+1];
            }
            list[list.length-1] = null;
        }
    }
}
```



```
public interface Menu {
    public Iterator<MenuItem> createIterator();
}
```

← This is a simple interface that just lets clients get an iterator for the items in the menu.

```
import java.util.Iterator;
```

← Now the Waitress uses the java.util.Iterator as well.

```
public class Waitress {
    Menu pancakeHouseMenu;
    Menu dinerMenu;

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

    public void printMenu() {
        Iterator<MenuItem> pancakeIterator = pancakeHouseMenu.createIterator();
        Iterator<MenuItem> dinerIterator = dinerMenu.createIterator();
        System.out.println("MENU\n---\nBREAKFAST");
        printMenu(pancakeIterator);
        System.out.println("\nLUNCH");
        printMenu(dinerIterator);
    }
}
```

← We need to replace the concrete Menu classes with the Menu Interface.

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

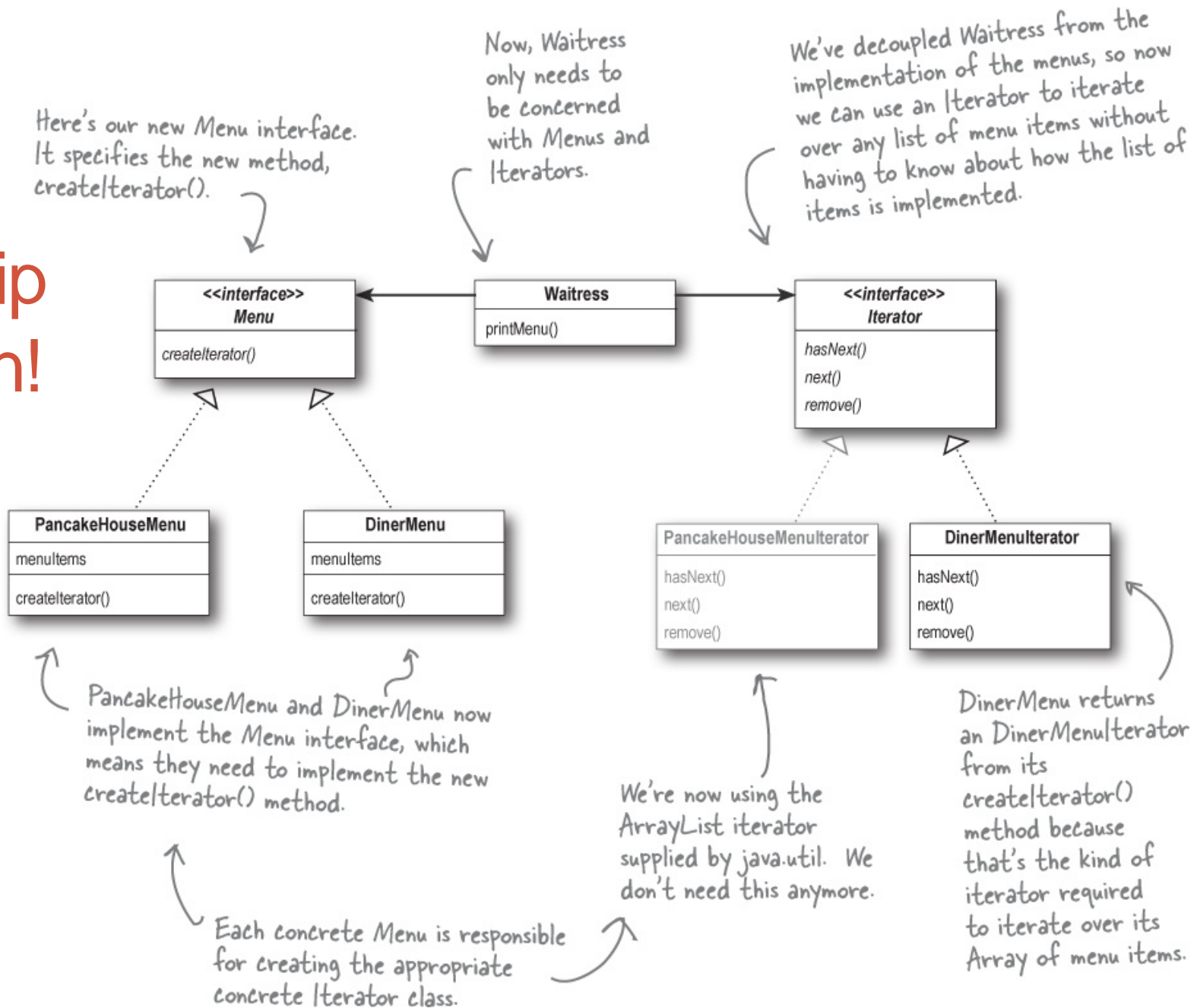
Nothing changes here.

```
// other methods here
```

```
}
```

Let's take
abstraction
a notch
further

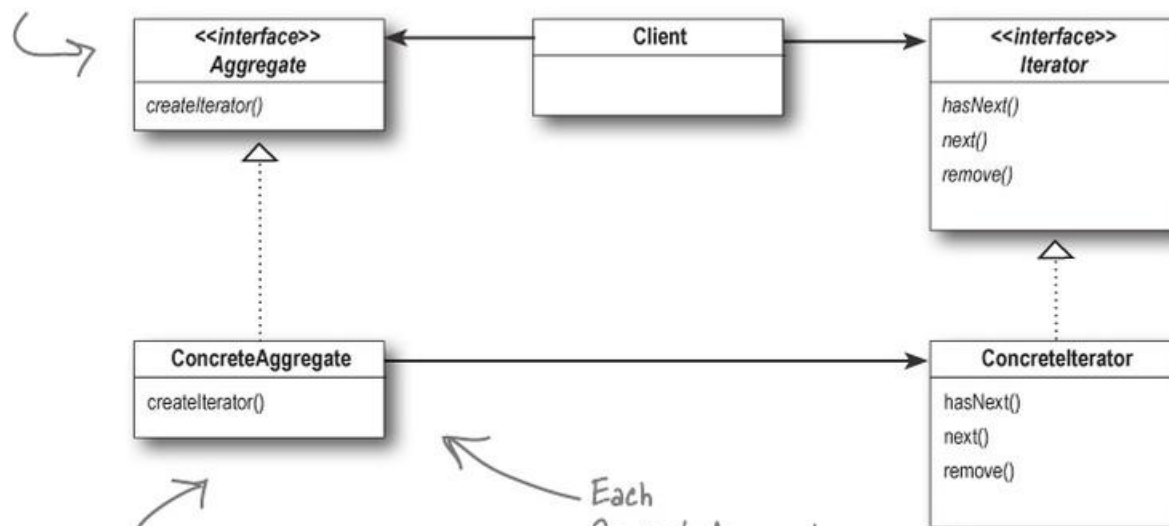
The new and hip design!



Iterator Pattern Defined

The **Iterator Pattern** provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

Having a common interface for your aggregates is handy for your clients; 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.

Recap

An Iterator allows **access to an aggregate's elements** without exposing its internal structure

When using an Iterator, we **relieve the aggregate** of the responsibility of supporting **operations for traversing its data**

An Iterator provides a **common interface for traversing** the items of an aggregate, **allowing you to use polymorphism** when writing code that makes use of the items of the aggregate