finished 1st round this whole
book on 14 - July - 09 3:49 AM

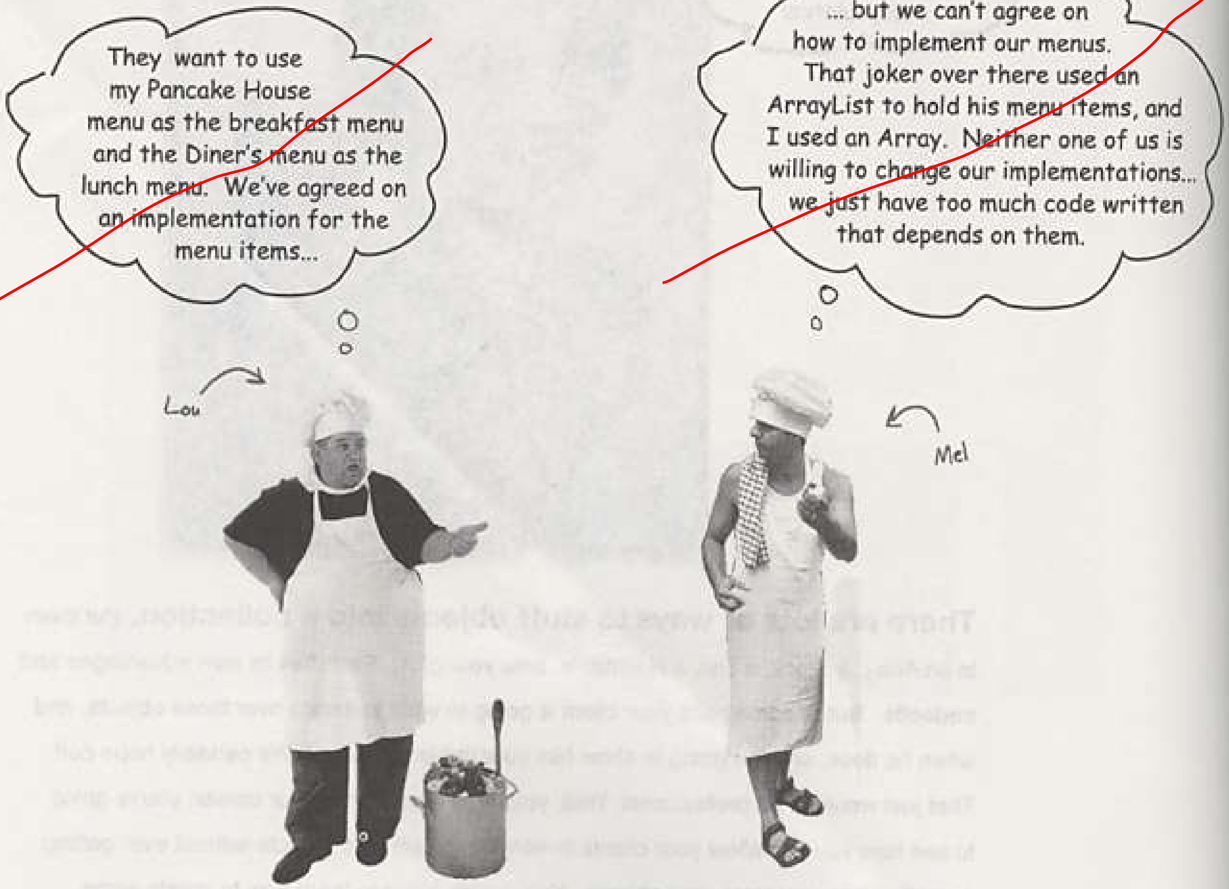
9 the Iterator and Composite Patterns

Well-Managed Collections

There are lots of ways to stuff objects into a collection. Put them in an Array, a Stack, a List, a Hashtable, take your pick. Each has its own advantages and tradeoffs. But 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? We certainly hope not! That just wouldn't be professional. Well, you don't have to risk your career; 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. You're also going to learn how to create some *super collections* of objects that can leap over some impressive data structures in a single bound. And if that's not enough, you're also going to learn a thing or two about object responsibility.

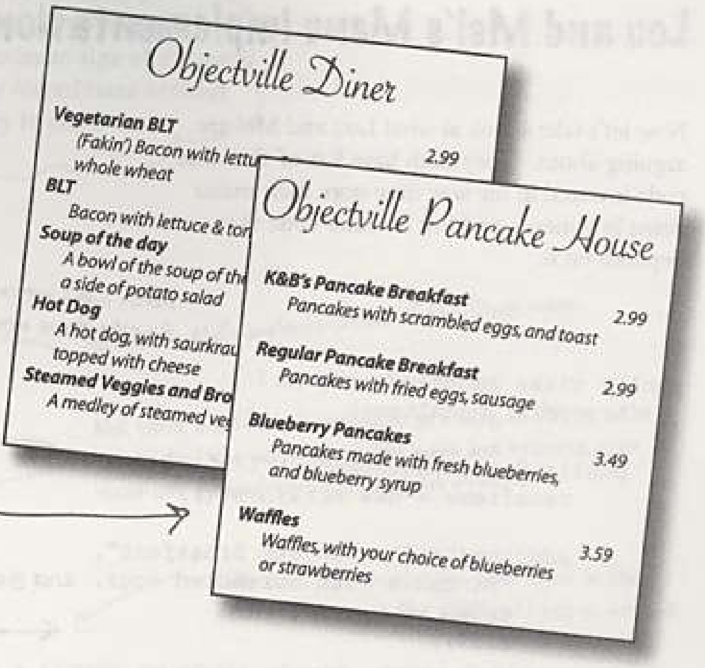
Breaking News: Objectville Diner and Objectville Pancake House Merge

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...


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

Check out the Menu Items

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



Objectville Diner

Vegetarian BLT
(Fakin') Bacon with lettu 2.99
whole wheat
BLT
Bacon with lettuce & tor
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

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

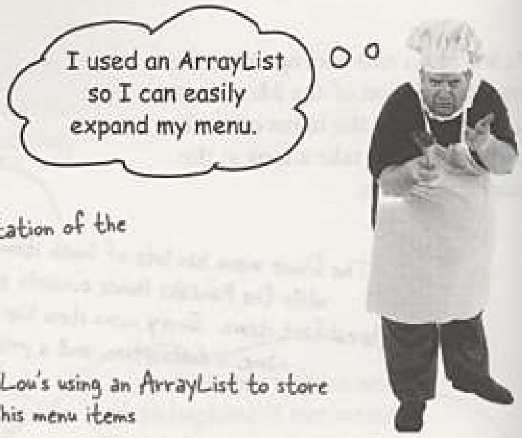
```
public class MenuItem {
    String name;
    String description;
    boolean vegetarian;
    double price;

    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 and Mel's Menu implementations

Now let's take a look at what Lou and Mel are arguing about. They both have lots of time and code invested in the way they store their menu items in a menu, and lots of other code that depends on it.



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

```
public class PancakeHouseMenu {
    ArrayList menuItems;

    public PancakeHouseMenu() {
        menuItems = new ArrayList();

        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 getMenuItems() {
        return menuItems;
    }

    // other menu methods here
}
```

Lou's using an ArrayList to store his menu items

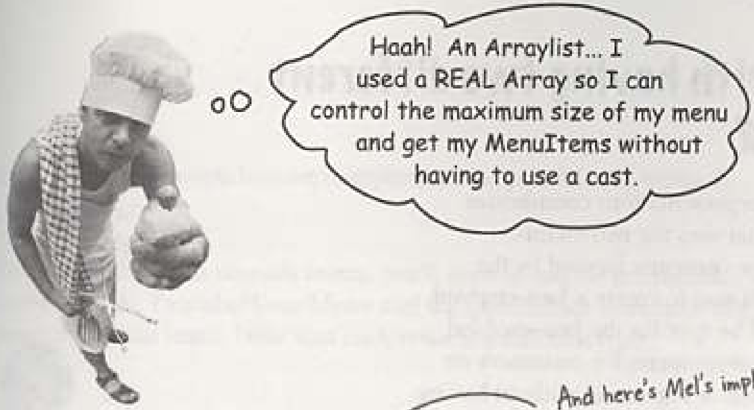
Each menu item is added to the ArrayList here, in the constructor

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!


Haah! An Arraylist... I used a REAL Array so I can control the maximum size of my menu and get my MenuItems without having to use a cast.

this is one of adv. while using array

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 and retrieve menu items out without having to cast his objects.

```
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
}
```

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.

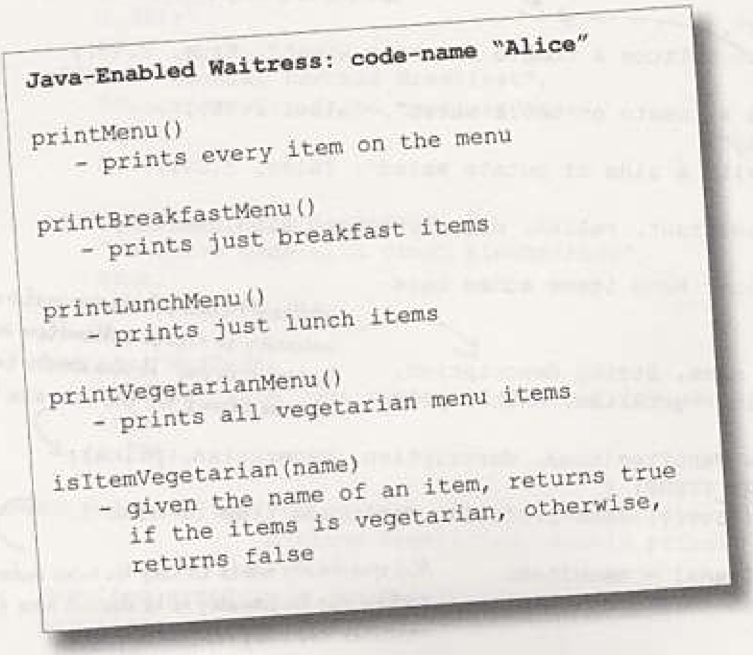
What's the problem with having two different menu representations?

To see why having two different menu representations complicates things, let's 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 (this *is* Objectville, after all). 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...

The Waitress is getting Java-enabled.

The Java-Enabled Waitress Specification



```
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 items is vegetarian, otherwise,
     returns false
```

The spec for the Waitress

Let's start by stepping through how we'd implement the printMenu() method:

1. To print all the items on each menu, you'll need to call the getMenuItem() 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 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, lunch items are in an Array.

2. 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 = (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.

3. 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.

Sharpen your pencil

Based on our implementation of printMenu(), which of the following apply?

□ A. We are coding to the PancakeHouseMenu and DinerMenu concrete implementations, not to an interface.

□ B. The Waitress doesn't implement the Java Waitress API and so she isn't adhering to a standard.

□ C. If we decided to switch from using DinerMenu to another type of menu that implemented its list of menu items with a Hashtable, we'd have to modify a lot of code in the Waitress.

□ C. The Waitress needs to know how each menu represents its internal collection of menu items; this violates encapsulation.

□ D. We have duplicate code: the printMenu() method needs two separate loops to iterate over the two different kinds of menus. And if we added a third menu, we'd have yet another loop.

□ E. The implementation isn't based on MXML (Menu XML) and so isn't as interoperable as it should be.

What now?

Mel and Lou are putting us in a difficult position. They don't want to change their implementations because it would mean rewriting a lot of code that is in each respective menu class. But 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.

It would really be nice if we could find a way to allow them to implement the same interface for their menus (they're already close, except for the return type of the getMenuItems() method). 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.

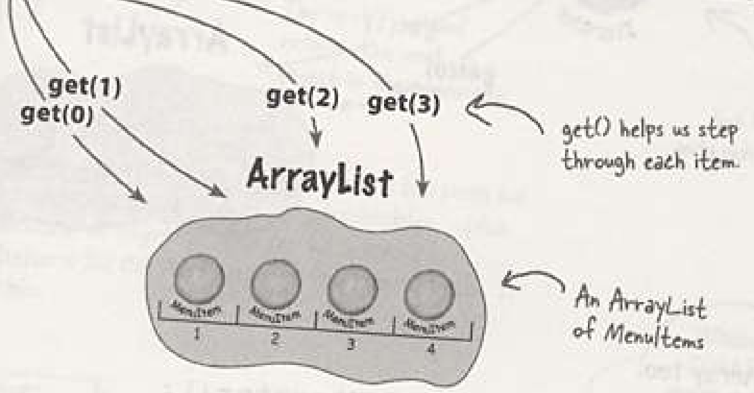
Sound good? Well, how are we going to do that?

Can we encapsulate the iteration?

If we've learned one thing in this book, it's encapsulate what varies. It's obvious what is changing here: the iteration caused by different collections of objects being returned from the menus. But can we encapsulate this? Let's work through the idea...

1. To iterate through the breakfast items we use the size() and get() methods on the ArrayList:

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

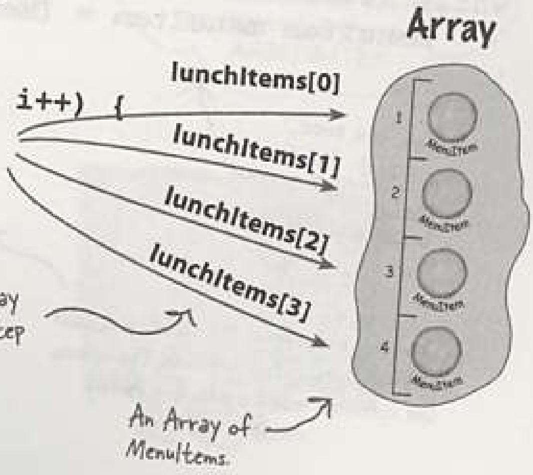


get() helps us step through each item.

An ArrayList of MenuItems

2. And to iterate through the lunch items we use the Array length field and the array subscript notation on the MenuItem Array.

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



We use the array subscripts to step through items.

An Array of MenuItems.

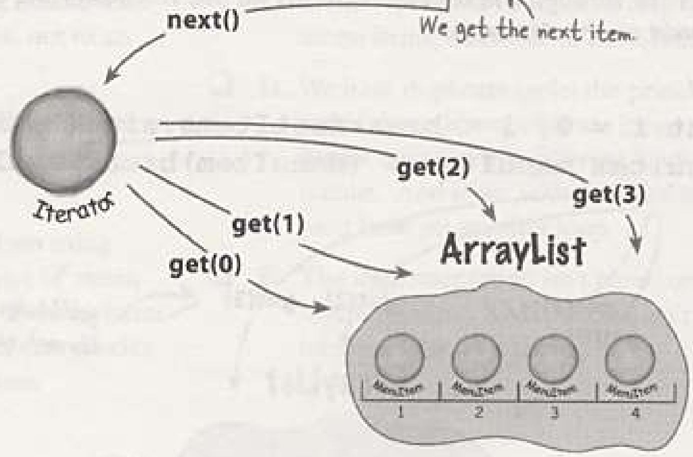
3 Now what if we create an object, let's call it an Iterator, that encapsulates the way we iterate through a collection of objects? Let's try this on the ArrayList

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

while (iterator.hasNext()) {
    MenuItem 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.

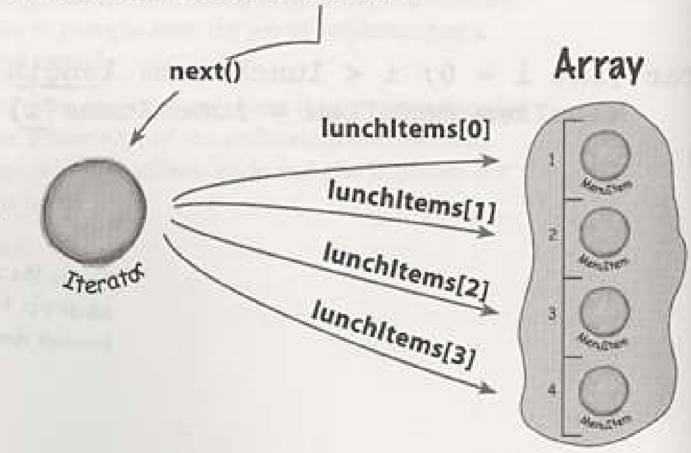


The client just calls hasNext() and next(); behind the scenes the iterator calls get() on the ArrayList.

4 Let's try that on the Array too:

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

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

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

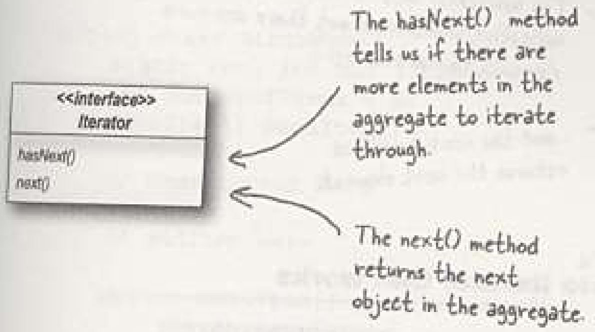


Same situation here: the client just calls hasNext() and next(); behind the scenes, the iterator indexes into the Array.

Meet the Iterator Pattern

Well, it looks like our plan of encapsulating iteration just might actually work; and as you've probably already guessed, it is a Design Pattern called the Iterator Pattern.

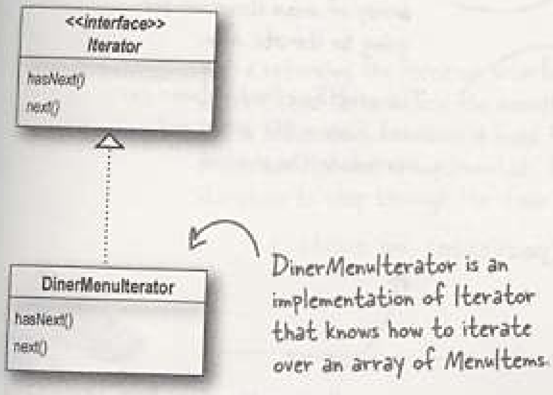
The first thing you need to know about the Iterator Pattern is that it relies on an interface called Iterator. Here's one possible Iterator interface:



The hasNext() method tells us if there are more elements in the aggregate to iterate through.

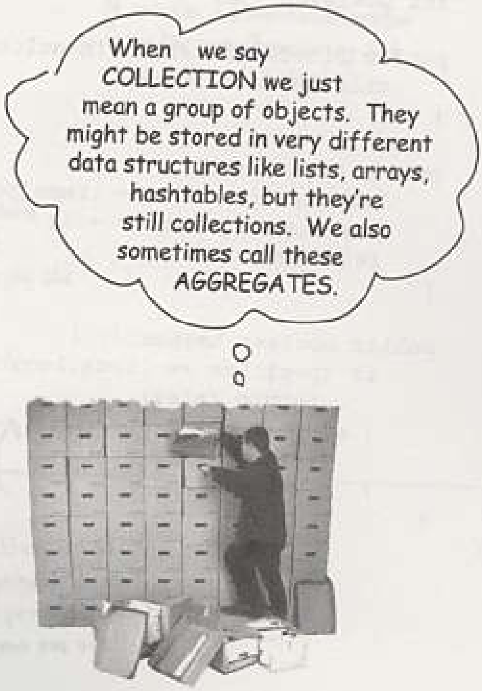
The next() method returns the next object in the aggregate.

Now, once we have this interface, we can implement Iterators for any kind of collection of objects: arrays, lists, hashtables, ...pick your favorite collection of objects. Let's say we wanted to implement the Iterator for the Array used in the DinerMenu. It would look like this:



DinerMenuIterator is an implementation of Iterator that knows how to iterate over an array of MenuItems.

Let's go ahead and implement this Iterator and hook it into the DinerMenu to see how this works...


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

Adding an Iterator to DinerMenu

To add an Iterator to the DinerMenu we first need to define the Iterator Interface:

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

Here's our two methods:

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.

super. simple form of iterator implementation

And now we need to implement a concrete Iterator that works for the Diner menu:

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

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

    public Object 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.

Reworking the Diner Menu with Iterator

Okay, we've got the iterator. Time to work it into the DinerMenu; all we need to do is add one method to create a DinerMenuIterator and return it to the client:

```
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.

Exercise

Go ahead and implement the PancakeHouseIterator yourself and make the changes needed to incorporate it into the PancakeHouseMenu.

Fixing up the Waitress code

Now we need to integrate the iterator code into the Waitress. We should be able to get rid of some of the redundancy in the process. Integration is pretty straightforward: first we create a printMenu() method that takes an Iterator, then we use the getIterator() method on each menu to retrieve the Iterator and pass it to the new method.

New and improved with Iterator.

```
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 = (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.

Testing our code

It's time to put everything to a test. Let's write some test drive code and see how the Waitress works...

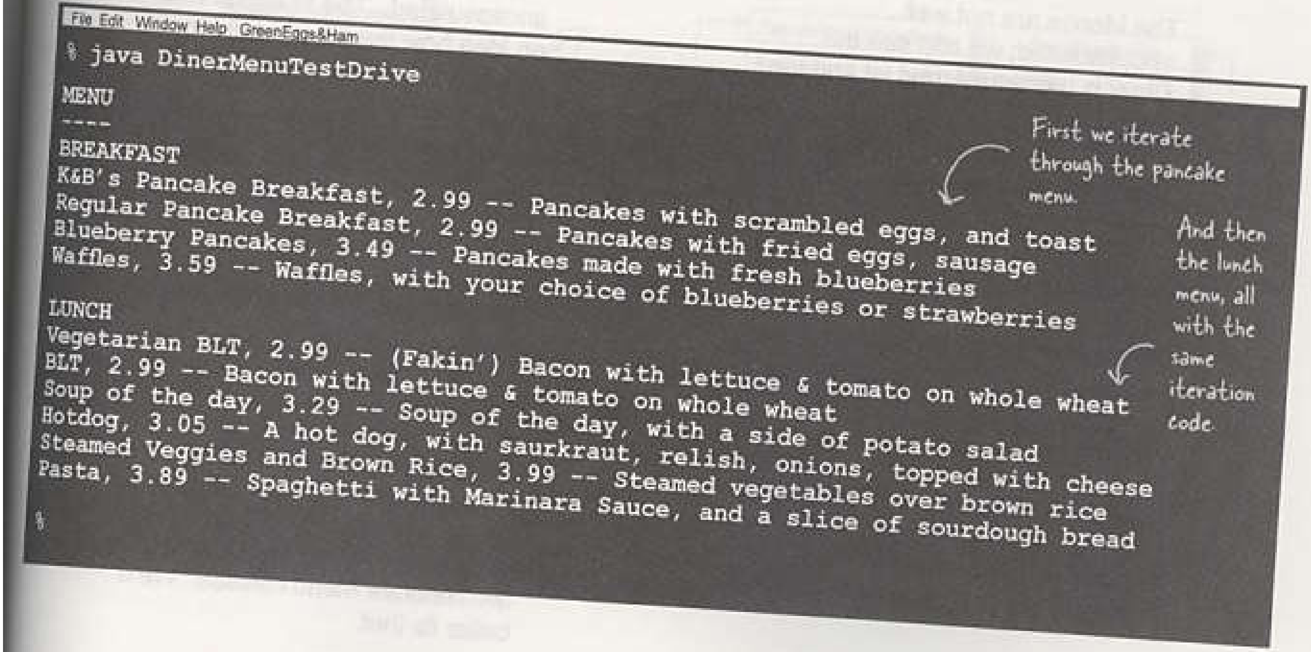
```
public class MenuTestDrive {
   public static void main(String args[]) {
       PancakeHouseMenu pancakeHouseMenu = new PancakeHouseMenu();
       DinerMenu dinerMenu = new DinerMenu();

       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.

Here's the test run...

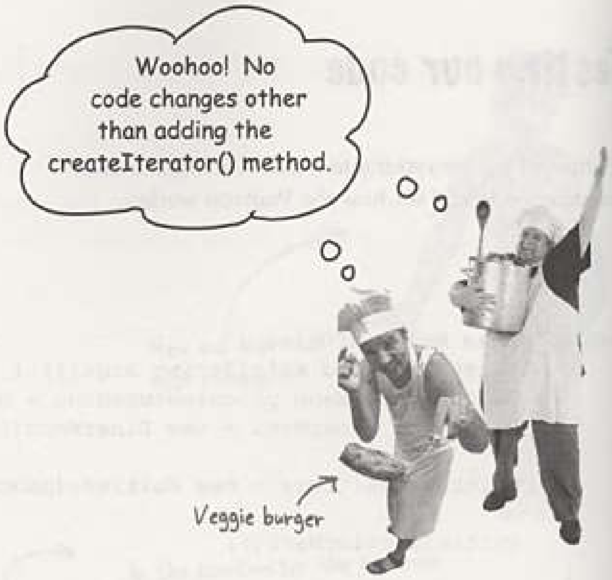


```
% java DinerMenuTestDrive
MENU
----
BREAKFAST
K&B's Pancake Breakfast, 2.99 -- Pancakes with scrambled eggs, and toast
Regular Pancake Breakfast, 2.99 -- Pancakes with fried eggs, sausage
Blueberry Pancakes, 3.49 -- Pancakes made with fresh blueberries
Waffles, 3.59 -- Waffles, with your choice of 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 have we done so far?


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

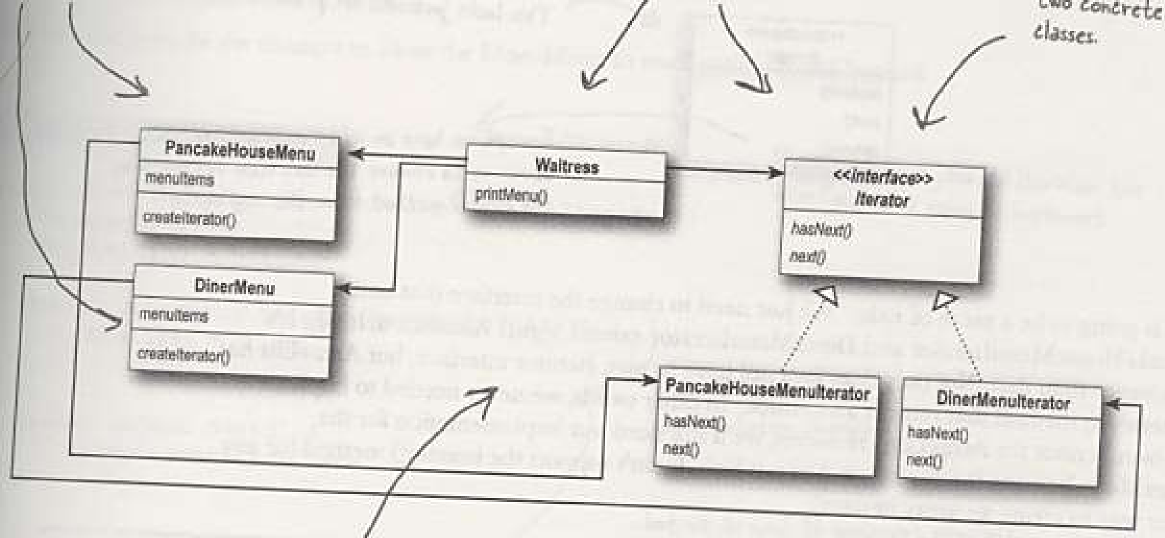
For starters, we've made our Objectville cooks very happy. They settled their differences and kept their own implementations. Once we gave them a PancakeHouseMenuIterator and a DinerMenuIterator, all they had to do was add a getIterator() method and they were finished.

We've also helped ourselves in the process. The Waitress will be much easier to maintain and extend down the road. Let's go through exactly what we did and think about the consequences:

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.

What we have so far...

Before we clean things up, let's get a bird's eye view of our current design.



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 PostIt™ 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 give 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.

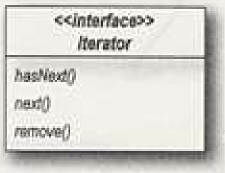
PancakeHouseMenu and DinerMenu implement the new createIterator() method; they are responsible for creating the iterator for their respective menu items implementations.

Making some improvements...

Okay, we know the interfaces of PancakeHouseMenu and DinerMenu are exactly the same and yet we haven't defined a common interface for them. So, we're going to do that and clean up the Waitress a little more.

You may be wondering why we're not using the Java Iterator interface – we did that so 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, because we'll get a lot of leverage by implementing that instead of our home grown Iterator interface. What kind of leverage? You'll soon see.

First, let's check out the java.util.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.

This is going to be a piece of cake: We just need to change the interface that both PancakeHouseMenuIterator and DinerMenuIterator extend, right? Almost... actually, it's even easier than that. Not only does java.util have its own Iterator interface, but ArrayList has an iterator() method that returns an iterator. In other words, we never needed to implement our own iterator for ArrayList. However, we'll still need our implementation for the DinerMenu because it relies on an Array, which doesn't support the iterator() method (or any other way to create an array iterator).

there are no Dumb Questions

Q: What if I don't want to provide the ability to remove something from the underlying collection of objects?

A: The remove() method is considered optional. You don't have to provide remove functionality. But, obviously you do need to provide the method because it's part of the Iterator interface. If you're not going to allow remove() in your iterator you'llwant to throw the runtime exception java.lang.UnsupportedOperationException. The Iterator API documentation specifies that this exception may be thrown from remove() and any client that is a good citizen will check for this exception when calling the remove() method.

Q: How does remove() behave under multiple threads that may be using different iterators over the same collection of objects?

A: The behavior of the remove() is unspecified if the collection changes while you are iterating over it. So you should be careful in designing your own multithreaded code when accessing a collection concurrently.

i saw, it will throw, IllegalState exception

Cleaning things up with java.util.Iterator

Let's start with the PancakeHouseMenu, changing it over to java.util.Iterator is going to be easy. We just delete the PancakeHouseMenuIterator class, add an import java.util.Iterator to the top of PancakeHouseMenu and change one line of the PancakeHouseMenu:

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

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

And that's it, PancakeHouseMenu is done.

Now we need to make the changes to allow the DinerMenu to work with java.util.Iterator.

```
import java.util.Iterator;

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

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

    public Object next() {
        //implementation here
    }

    public boolean hasNext() {
        //implementation here
    }

    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;
        }
    }
}
```

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

None of our current implementation changes...

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

We are almost there...

We just need to give the Menus a common interface and rework the Waitress a little. The Menu interface is quite simple: we might want to add a few more methods to it eventually, like addItem(), but for now we will let the chefs control their menus by keeping that method out of the public interface:

```
public interface Menu {
    public Iterator createIterator();
}
```

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

Now we need to add an `implements Menu` to both the PancakeHouseMenu and the DinerMenu class definitions and update the Waitress:

```
import java.util.Iterator;

public class Waitress {
    Menu pancakeHouseMenu;
    Menu dinerMenu;

    public Waitress(Menu pancakeHouseMenu, Menu 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 = (MenuItem)iterator.next();
            System.out.print(menuItem.getName() + ", ");
            System.out.print(menuItem.getPrice() + " -- ");
            System.out.println(menuItem.getDescription());
        }
    }

    // other methods here
}
```

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

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

Nothing changes here.

Wh

The I
Menu
the co
the co

The r
by Pa
respo
imple

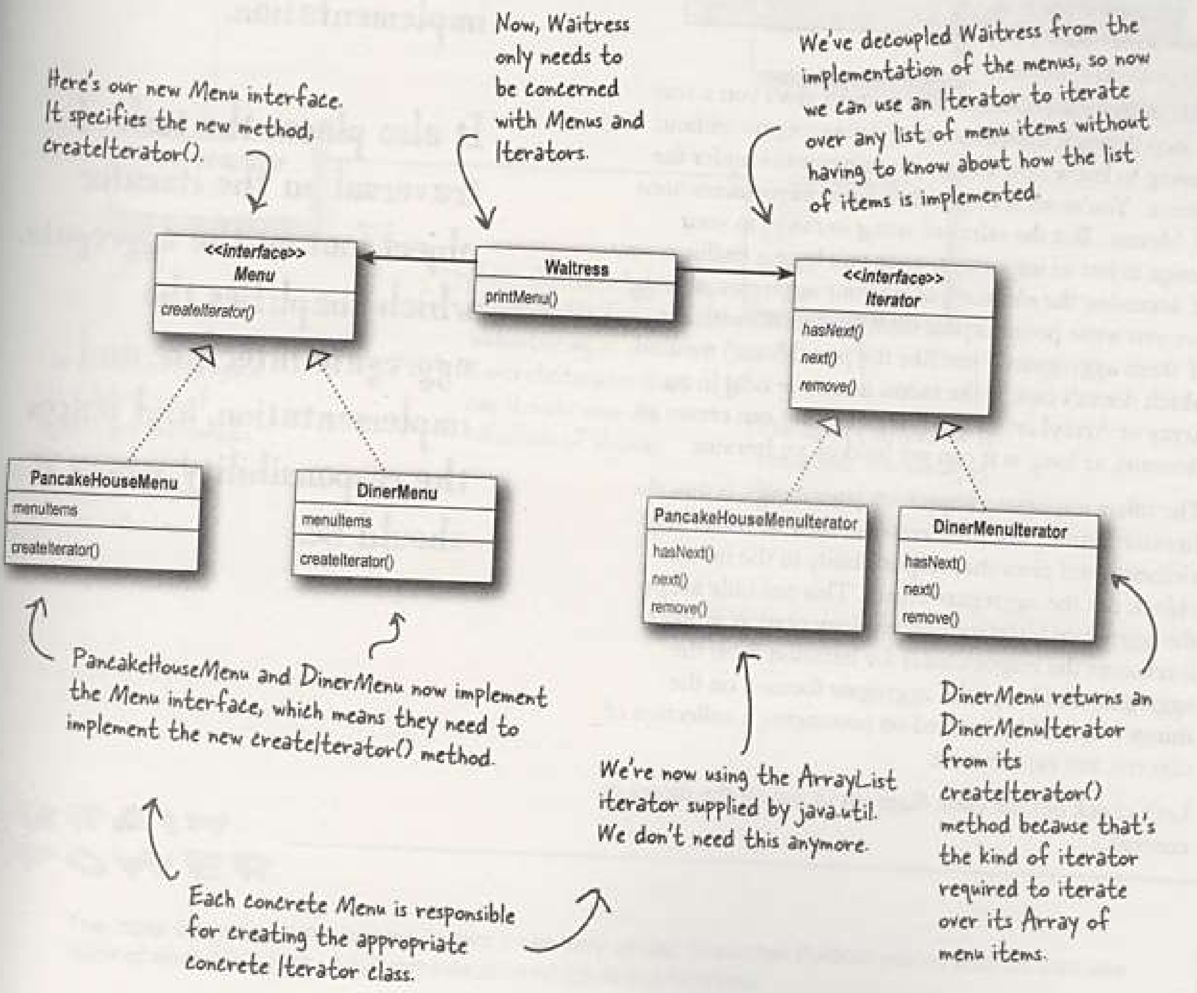
What does this get us?

The PancakeHouseMenu and DinerMenu classes implement an interface, Menu. Waitress can refer to each menu object using the interface rather than the concrete class. So, we're reducing the dependency between the Waitress and the concrete classes by "programming to an interface, not an implementation."

This solves the problem of the Waitress depending on the concrete Menus.

The new Menu interface has one method, createIterator(), that is implemented by PancakeHouseMenu and DinerMenu. Each menu class assumes the responsibility of creating a concrete Iterator that is appropriate for its internal implementation of the menu items.

This solves the problem of the Waitress depending on the implementation of the MenuItems.



Here's our new Menu interface. It specifies the new method, createIterator().

Now, Waitress only needs to be concerned with Menus and Iterators.

We've decoupled Waitress from the implementation of the menus, so now we can use an Iterator to iterate over any list of menu items without having to know about how the list of items is implemented.

PancakeHouseMenu and DinerMenu now implement the Menu interface, which means they need to implement the new createIterator() method.

We're now using the ArrayList iterator supplied by java.util. We don't need this anymore.

DinerMenu returns an DinerMenuIterator from its createIterator() method because that's the kind of iterator required to iterate over its Array of menu items.

Each concrete Menu is responsible for creating the appropriate concrete Iterator class.

Iterator Pattern defined

You've already seen how to implement the Iterator Pattern with your very own iterator. You've also seen how Java supports iterators in some of its collection oriented classes (the ArrayList). Now it's time to check out the official definition of the pattern:

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

~~This makes a lot of sense: the pattern gives you a way to step through the elements of an aggregate without having to know how things are represented under the covers. You've seen that with the two implementations of Menus. But the effect of using iterators in your design is just as important: once you have a uniform way of accessing the elements of all your aggregate objects, you can write polymorphic code that works with *any* of these aggregates – just like the printMenu() method, which doesn't care if the menu items are held in an Array or ArrayList (or anything else that can create an Iterator), as long as it can get hold of an Iterator.~~

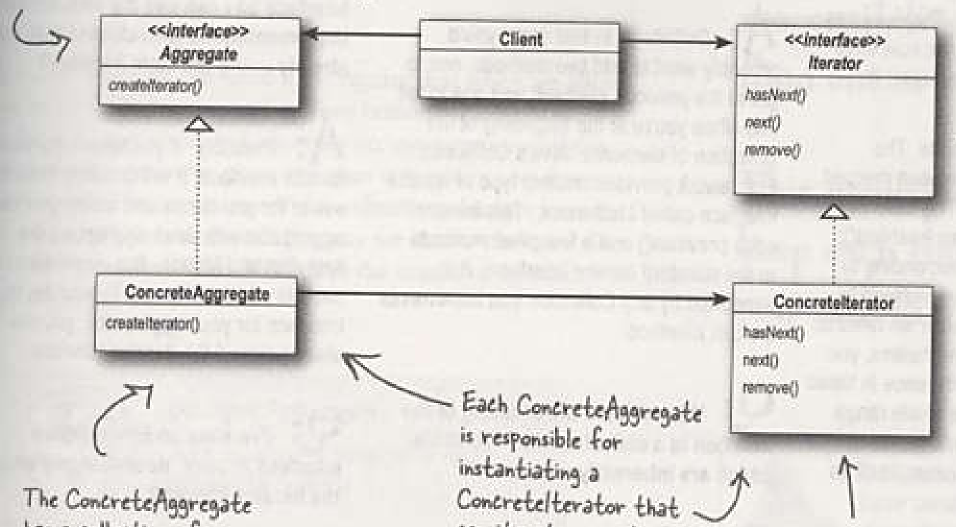
The other important impact on your design is that the Iterator Pattern takes the responsibility of traversing elements and gives that responsibility to the iterator object, not the aggregate object. This not only keeps the aggregate interface and implementation simpler, it removes the responsibility for iteration from the aggregate and keeps the aggregate focused on the things it should be focused on (managing a collection of objects), not on iteration.

Let's check out the class diagram to put all the pieces in context...

if need, we can implement the iterator interface any non-built in collection ourselves

The Iterator Pattern allows traversal of the elements of an aggregate without exposing the underlying implementation.

It also places the task of traversal on the iterator object, not on the aggregate, which simplifies the aggregate interface and implementation, and places the responsibility where it should be.



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.

BRAIN POWER

The class diagram for the Iterator Pattern looks very similar to another Pattern you've studied; can you think of what it is? Hint: A subclass decides which object to create.

there are no Dumb Questions

Q: I've seen other books show the Iterator class diagram with the methods first(), next(), isDone() and currentItem(). Why are these methods different?

A: Those are the "classic" method names that have been used. These names have changed over time and we now have next(), hasNext() and even remove() in java.util.Iterator.

Let's look at the classic methods. The next() and currentItem() have been merged into one method in java.util. The isDone() method has obviously become hasNext(); but we have no method corresponding to first(). That's because in Java we tend to just get a new iterator whenever we need to start the traversal over. Nevertheless, you can see there is very little difference in these interfaces. In fact, there is a whole range of behaviors you can give your iterators. The remove() method is an example of an extension in java.util.Iterator.

Q: I've heard about "internal" iterators and "external" iterators. What are they? Which kind did we implement in the example?

A: We implemented an external iterator, which means that the client controls the iteration by calling next() to get the next element. An internal iterator is controlled by the iterator itself. In that case, because it's the iterator that's stepping through the elements, you have to tell the iterator what to do with those elements as it goes through them. That means you need a way to pass an operation to an iterator. Internal iterators are less flexible that external iterators because the client doesn't have control of the iteration. However, some might argue that they are easier to use because you just hand them an operation and tell them to iterate, and they do all the work for you.

Q: Could I implement an Iterator that can go backwards as well as backwards?

A: Definitely. In that case, you'd probably want to add two methods, one to get to the previous element, and one to tell you when you're at the beginning of the collection of elements. Java's Collection Framework provides another type of iterator interface called ListIterator. This iterator adds previous() and a few other methods to the standard Iterator interface. It is supported by any Collection that implements the List interface.

Q: Who defines the ordering of the iteration in a collection like Hashtable, which are inherently unordered?

A: Iterators imply no ordering. The underlying collections may be unordered as in a hashtable or in a bag; they may even contain duplicates. So ordering is related to both the properties of the underlying collection and to the implementation. In general, you should make no assumptions about ordering unless the Collection documentation indicates otherwise.

Q: You said we can write "polymorphic code" using an iterator; can you explain that more?

A: When we write methods that take Iterators as parameters, we are using polymorphic iteration. That means we are creating code that can iterate over any collection as long as it supports Iterator. We don't care about how the collection is implemented, we can still write code to iterate over it.

Q: If I'm using Java, won't I always want to use the java.util.Iterator interface so I can use my own iterator implementations with classes that are already using the Java iterators?

A: Probably. If you have a common Iterator interface, it will certainly make it easier for you to mix and match your own aggregates with Java aggregates like ArrayList and Vector. But remember, if you need to add functionality to your Iterator interface for your aggregates, you can always extend the Iterator interface.

Q: I've seen an Enumeration interface in Java; does that implement the Iterator Pattern?

A: We talked about this in the Adapter Chapter. Remember? The java.util.Enumeration is an older implementation of Iterator that has since been replaced by java.util.Iterator. Enumeration has two methods, hasMoreElements(), corresponding to hasNext(), and nextElement(), corresponding to next(). However, you'll probably want to use Iterator over Enumeration as more Java classes support it. If you need to convert from one to another, review the Adapter Chapter again where you implemented the adapter for Enumeration and Iterator.

Single Responsibility

What if we allowed our aggregates to implement their internal collections and related operations AND the iteration methods? Well, we already know that would expand the number of methods in the aggregate, but so what? Why is that so bad?

Well, to see why, you first need to recognize that when we allow a class to not only take care of its own business (managing some kind of aggregate) but also take on more responsibilities (like iteration) then we've given the class two reasons to change. Two? Yup, two: it can change if the collection changes in some way, and it can change if the way we iterate changes. So once again our friend CHANGE is at the center of another design principle:

> **Design Principle**
>
> *A class should have only one reason to change.*

We know we want to avoid change in a class like the plague – modifying code provides all sorts of opportunities for problems to creep in. Having two ways to change increases the probability the class will change in the future, and when it does, it's going to affect two aspects of your design.

The solution? The principle guides us to assign each responsibility to one class, and only one class.

That's right, it's as easy as that, and then again it's not: separating responsibility in design is one of the most difficult things to do. Our brains are just too good at seeing a set of behaviors and grouping them together even when there are actually two or more responsibilities. The only way to succeed is to be diligent in examining your designs and to watch out for signals that a class is changing in more than one way as your system grows.

Every responsibility of a class is an area of potential change. More than one responsibility means more than one area of change.

This principle guides us to keep each class to a single responsibility.

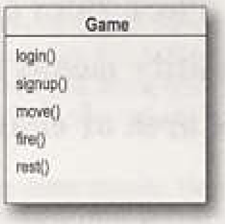
Cohesion is a term you'll hear used as a measure of how closely a class or a module supports a single purpose or responsibility.

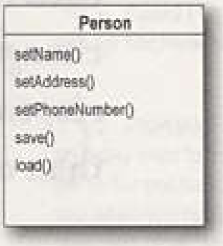
We say that a module or class has *high cohesion* when it is designed around a set of related functions, and we say it has *low cohesion* when it is designed around a set of unrelated functions.

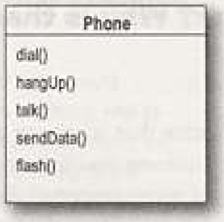
Cohesion is a more general concept than the Single Responsibility Principle, but the two are closely related. Classes that adhere to the principle tend to have high cohesion and are more maintainable than classes that take on multiple responsibilities and have low cohesion.

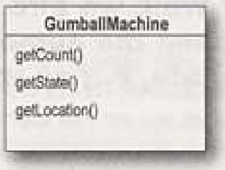
BRAIN POWER

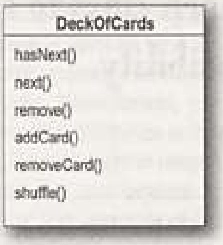
Examine these classes and determine which ones have multiple responsibilities.

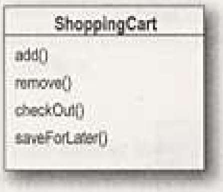


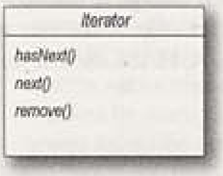








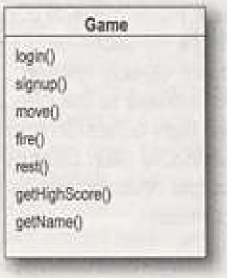


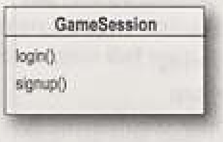


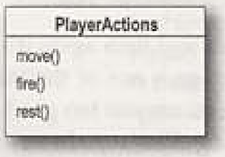
HARD HAT AREA, WATCH OUT FOR FALLING ASSUMPTIONS

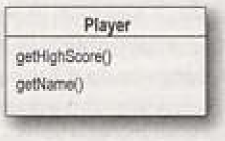
BRAIN^2 POWER

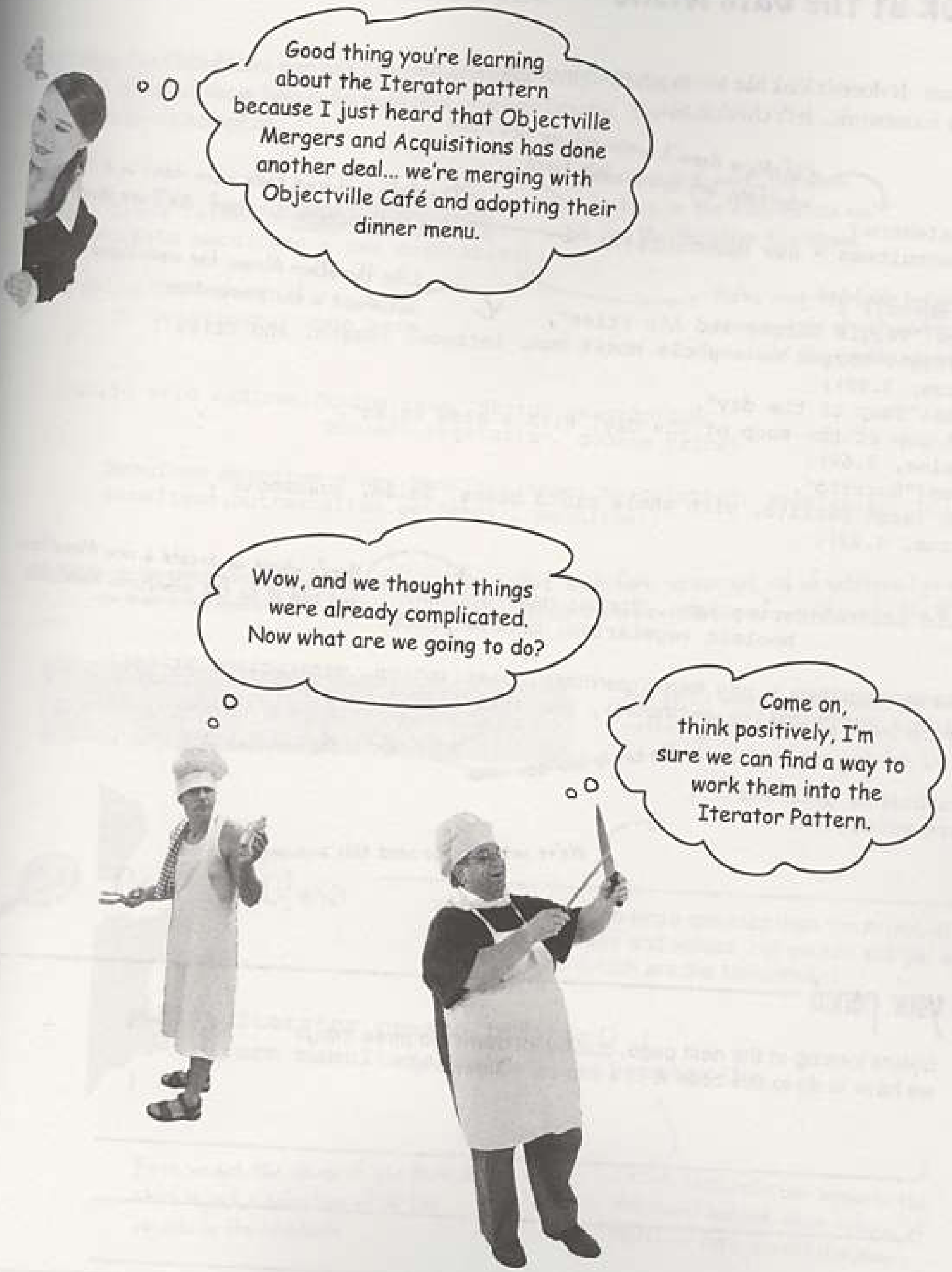
Determine if these classes have low or high cohesion.










Good thing you're learning about the Iterator pattern because I just heard that Objectville Mergers and Acquisitions has done another deal... we're merging with Objectville Café and adopting their dinner menu.
Wow, and we thought things were already complicated. Now what are we going to do?
Come on, think positively, I'm sure we can find a way to work them into the Iterator Pattern.

Taking a look at the Café Menu

Here's the Café Menu. It doesn't look like too much trouble to integrate the Cafe Menu into our framework... let's check it out.

CafeMenu doesn't implement our new Menu interface, but this is easily fixed.

The Café is storing their menu items in a Hashtable. Does that support Iterator? We'll see shortly...

Like the other Menus, the menu items are initialized in the constructor.

```
public class CafeMenu {
    Hashtable menuItems = new Hashtable();

    public CafeMenu() {
        addItem(“Veggie Burger and Air Fries”,
            “Veggie burger on a whole wheat bun, lettuce, tomato, and fries”,
            true, 3.99);
        addItem(“Soup of the day”,
            “A cup of the soup of the day, with a side salad”,
            false, 3.69);
        addItem(“Burrito”,
            “A large burrito, with whole pinto beans, salsa, guacamole”,
            true, 4.29);
    }

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

    public Hashtable getItems() {
        return menuItems;
    }
}
```

Here's where we create a new MenuItem and add it to the menuItems hashtable.

the key is the item name.

the value is the menuItem object.

We're not going to need this anymore.

Sharpen your pencil

Before looking at the next page, quickly jot down the three things we have to do to this code to fit it into our framework:

1. ______________________________

2. ______________________________

3. ______________________________

Reworking the Café Menu code

Integrating the Cafe Menu into our framework is easy. Why? Because Hashtable is one of those Java collections that supports Iterator. But it's not quite the same as ArrayList...

```
public class CafeMenu implements Menu {
    Hashtable menuItems = new Hashtable();

    public CafeMenu() {
        // constructor code here
    }

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

    ~~public Hashtable getItems() {~~
        ~~return menuItems;~~
    ~~}~~

    public Iterator createIterator() {
        return menuItems.values().iterator();
    }
}
```

CafeMenu implements the Menu interface, so the Waitress can use it just like the other two Menus.

We're using Hashtable because it's a common data structure for storing values; you could also use the newer HashMap.

Just like before, we can get rid of getItems() so we don't expose the implementation of menuItems to the Waitress.

And here's where we implement the createIterator() method. Notice that we're not getting an Iterator for the whole Hashtable, just for the values.

Code Up Close

Hashtable is a little more complex than the ArrayList because it supports both keys and values, but we can still get an Iterator for the values (which are the MenuItems).

```
public Iterator createIterator() {
    return menuItems.values().iterator();
}
```

First we get the values of the Hashtable, which is just a collection of all the objects in the hashtable.

Luckily that collection supports the iterator() method, which returns a object of type java.util.Iterator.

Adding the Café Menu to the Waitress

That was easy; how about modifying the Waitress to support our new Menu? Now that the Waitress expects Iterators, that should be easy too.

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

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

    public void printMenu() {
        Iterator pancakeIterator = pancakeHouseMenu.createIterator();
        Iterator dinerIterator = dinerMenu.createIterator();
        Iterator cafeIterator = cafeMenu.createIterator();
        System.out.println(“MENU\n----\nBREAKFAST”);
        printMenu(pancakeIterator);
        System.out.println(“\nLUNCH”);
        printMenu(dinerIterator);
        System.out.println(“\nDINNER”);
        printMenu(cafeIterator);
    }

    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());
        }
    }
}
```

The Café menu is passed into the Waitress in the constructor with the other menus, and we stash it in an instance variable.

We're using the Café's menu for our dinner menu. All we have to do to print it is create the iterator, and pass it to printMenu(). That's it!

Nothing changes here

Breakfast, lunch AND dinner

Let's update our test drive to make sure this all works.

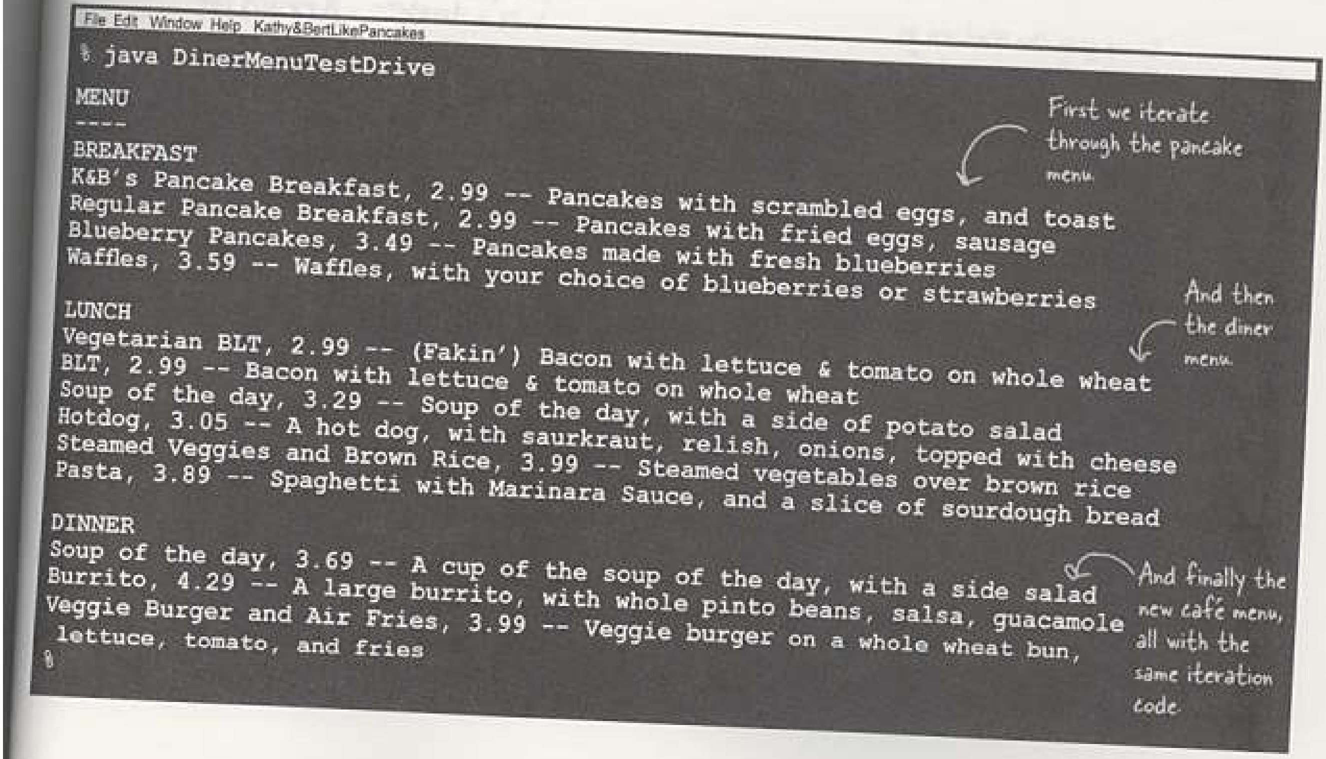
```
public class MenuTestDrive {
    public static void main(String args[]) {
        PancakeHouseMenu pancakeHouseMenu = new PancakeHouseMenu();
        DinerMenu dinerMenu = new DinerMenu();
        CafeMenu cafeMenu = new CafeMenu();

        Waitress waitress = new Waitress(pancakeHouseMenu, dinerMenu, cafeMenu);

        waitress.printMenu();
}
```

Create a CafeMenu...

... and pass it to the waitress.

Now, when we print we should see all three menus.

Here's the test run; check out the new dinner menu from the Café!



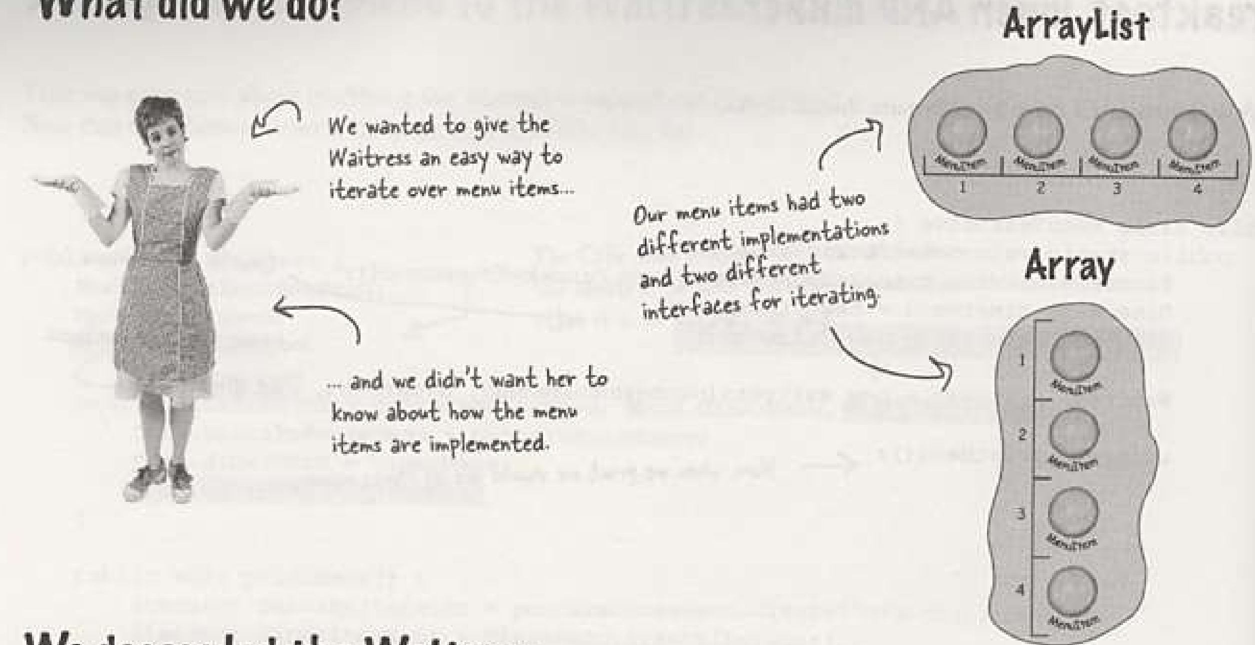
```
% java DinerMenuTestDrive
MENU
----
BREAKFAST
K&B's Pancake Breakfast, 2.99 -- Pancakes with scrambled eggs, and toast
Regular Pancake Breakfast, 2.99 -- Pancakes with fried eggs, sausage
Blueberry Pancakes, 3.49 -- Pancakes made with fresh blueberries
Waffles, 3.59 -- Waffles, with your choice of 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

DINNER
Soup of the day, 3.69 -- A cup of the soup of the day, with a side salad
Burrito, 4.29 -- A large burrito, with whole pinto beans, salsa, guacamole
Veggie Burger and Air Fries, 3.99 -- Veggie burger on a whole wheat bun,
 lettuce, tomato, and fries
%
```

First we iterate through the pancake menu.

And then the diner menu.

And finally the new café menu, all with the same iteration code.

What did we do?

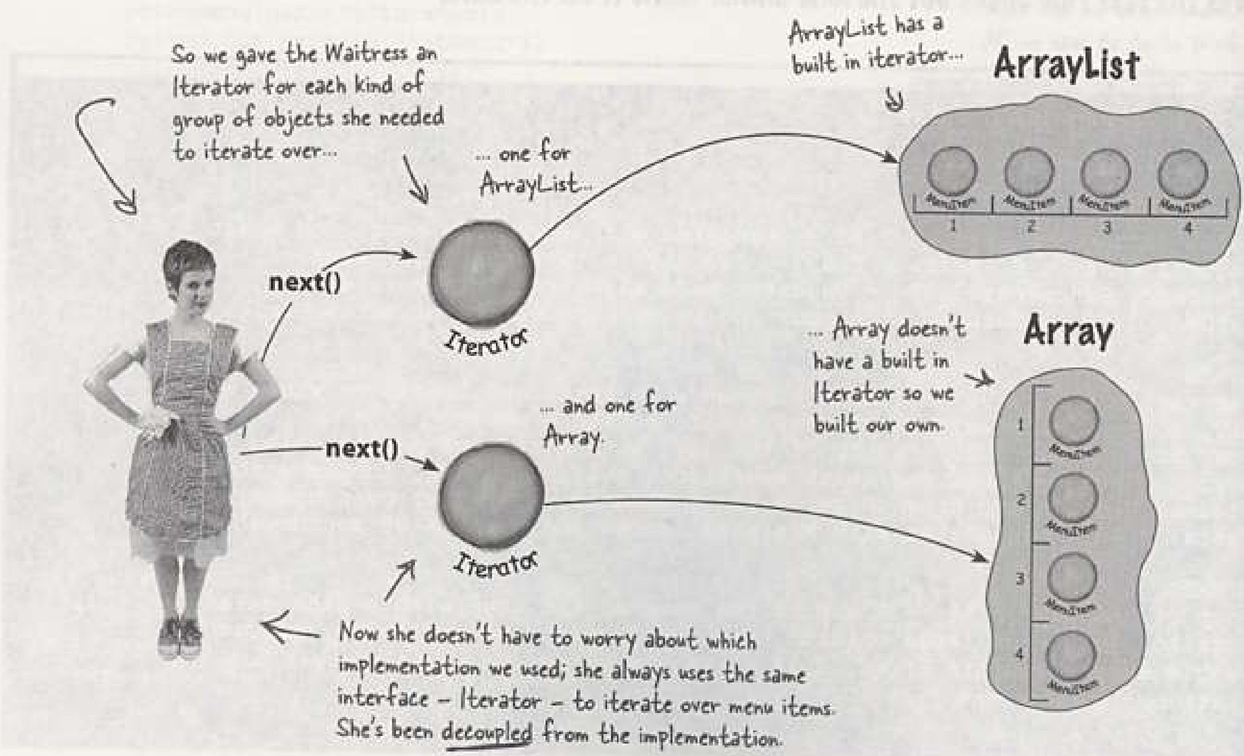


We wanted to give the Waitress an easy way to iterate over menu items...

... 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.

We decoupled the Waitress....



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...

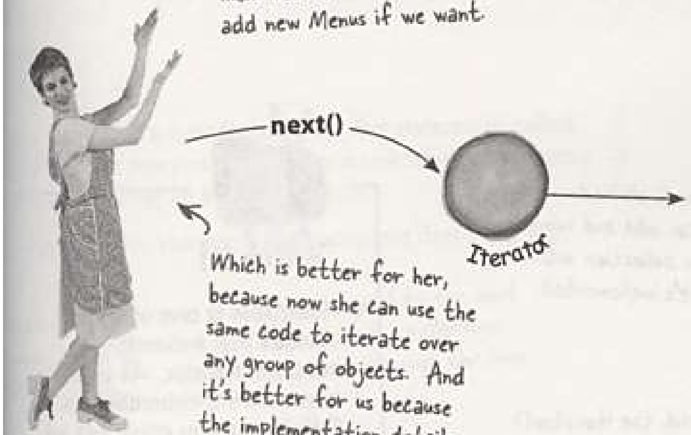
... and one for Array.

... Array doesn't have a built in Iterator so we built our own.

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.

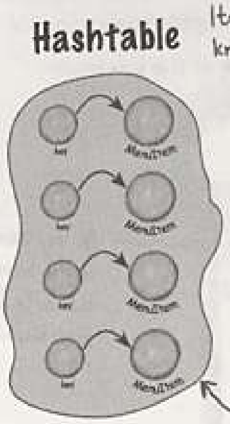
. and we made the Waitress more extensible

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 Hashtable values was easy; when you call values.iterator() you get an Iterator.

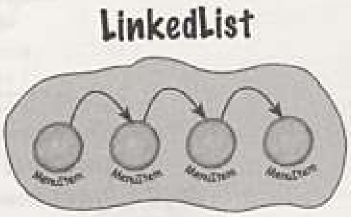
But there's more!

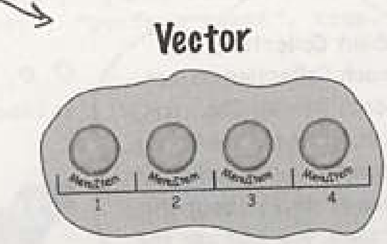
Java gives you a lot of "collection" classes that allow you to store and retrieve groups of objects. For example, Vector and LinkedList.

Most have different interfaces.

But almost all of them support a way to obtain an Iterator.

And if they don't support Iterator, that's ok, because now you know how to build your own.



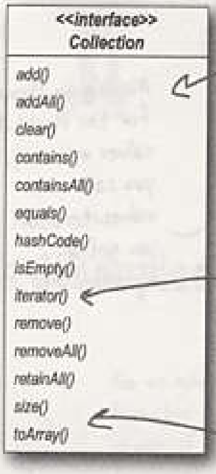


...and more!

Iterators and Collections

We've been using a couple of classes that are part of the Java Collections Framework. This "framework" is just a set of classes and interfaces, including ArrayList, which we've been using, and many others like Vector, LinkedList, Stack, and PriorityQueue. Each of these classes implements the java.util.Collection interface, which contains a bunch of useful methods for manipulating groups of objects.

Let's take a quick look at the interface:



As you can see, there's all kinds of good stuff here. You can add and remove elements from your collection without even knowing how it's implemented.

Here's our old friend, the iterator() method. With this method, you can get an Iterator for any class that implements the Collection interface.

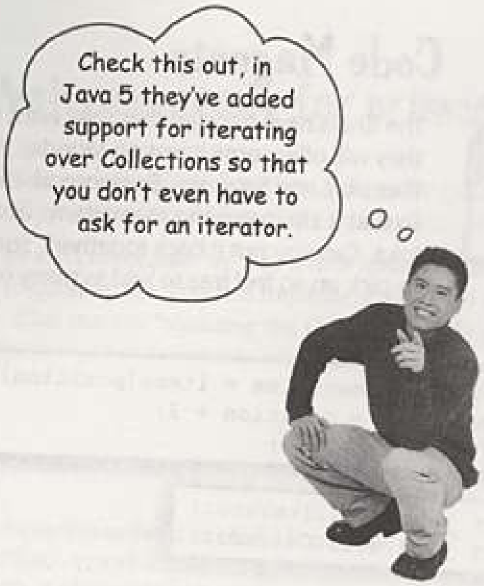
Other handy methods include size(), to get the number of elements, and toArray() to turn your collection into an array.

Watch it!

Hashtable is one of a few classes that *indirectly* supports Iterator. As you saw when we implemented the CafeMenu, you could get an Iterator from it, but only by first retrieving its Collection called values. If you think about it, this makes sense: the Hashtable holds two sets of objects: keys and values. If we want to iterate over its values, we first need to retrieve them from the Hashtable, and then obtain the iterator.

The nice thing about Collections and Iterator is that each Collection object knows how to create its own Iterator. Calling iterator() on an ArrayList returns a concrete Iterator made for ArrayLists, but you never need to see or worry about the concrete class it uses; you just use the Iterator interface.

Iterators and Collections in Java 5


Check this out, in Java 5 they've added support for iterating over Collections so that you don't even have to ask for an iterator.

Java 5 includes a new form of the **for** statement, called **for/in**, that lets you iterate over a collection or an array without creating an iterator explicitly.

To use **for/in**, you use a **for** statement that looks like:

Iterates over each object in the collection.

obj is assigned to the next element in the collection each time through the loop.

```
for (Object obj: collection) {
        ...
}
```

Here's how you iterate over an ArrayList using **for/in**:

Load up an ArrayList of MenuItems.

```
ArrayList items = new ArrayList();
items.add(new MenuItem("Pancakes", "delicious pancakes", true, 1.59);
items.add(new MenuItem("Waffles", "yummy waffles", true, 1.99);
items.add(new MenuItem("Toast", "perfect toast", true, 0.59);

for (MenuItem item: items) {
    System.out.println("Breakfast item: " + item);
}
```

Iterate over the list and print each item.

Watch it!

You need to use Java 5's new generics feature to ensure for/in type safety. Make sure you read up on the details before using generics and for/in.

Code Magnets

The Chefs have decided that they want to be able to alternate their lunch menu items; in other words, they will offer some items on Monday, Wednesday, Friday and Sunday, and other items on Tuesday, Thursday, and Saturday. Someone already wrote the code for a new "Alternating" DinerMenu Iterator so that it alternates the menu items, but they scrambled it up and put it on the fridge in the Diner as a joke. Can you put it back together? Some of the curly braces fell on the floor and they were too small to pick up, so feel free to add as many of those as you need.

```
MenuItem menuItem = items[position];
position = position + 2;
return menuItem;
```

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

```
public Object next() {
```

```
{
```

```
public AlternatingDinerMenuIterator(MenuItem[] items)
```

```
this.items = items;
Calendar rightNow = Calendar.getInstance();
position = rightNow.DAY_OF_WEEK % 2;
```

```
public void remove() {
```

```
implements Iterator
```

```
MenuItem[] items;
int position;
```

```
}
```

```
public class AlternatingDinerMenuIterator
```

```
public boolean hasNext() {
```

```
throw new UnsupportedOperationException(
    "Alternating Diner Menu Iterator does not support remove()");
```

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

```
}
```

Is the Waitress ready for prime time?

The Waitress has come a long way, but you've gotta admit those three calls to printMenu() are looking kind of ugly.

Let's be real, every time we add a new menu we are going to have to open up the Waitress implementation and add more code. *Can you say "violating the Open Closed Principle?"*

Three createIterator() calls.

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

    System.out.println(“MENU\n----\nBREAKFAST”);
    printMenu(pancakeIterator);

    System.out.println(“\nLUNCH”);
    printMenu(dinerIterator);

    System.out.println(“\nDINNER”);
    printMenu(cafeIterator);
}
```

Three calls to printMenu.

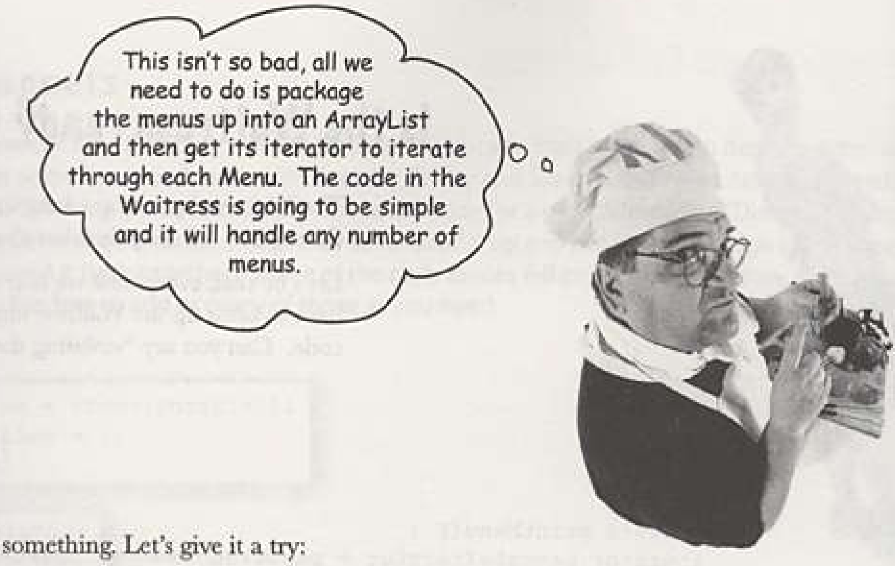
Everytime we add or remove a menu we're going to have to open this code up for changes.

so, re-thinking of a design will make bring better than before

It's not the Waitress' fault. We have done a great job of decoupling the menu implementation and extracting the iteration into an iterator. But we still are handling the menus with separate, independent objects – we need a way to manage them together.

BRAIN POWER

The Waitress still needs to make three calls to printMenu(), one for each menu. Can you think of a way to combine the menus so that only one call needs to be made? Or perhaps so that one Iterator is passed to the Waitress to iterate over all the menus?


This isn't so bad, all we need to do is package the menus up into an ArrayList and then get its iterator to iterate through each Menu. The code in the Waitress is going to be simple and it will handle any number of menus.

Sounds like the chef is on to something. Let's give it a try:

```
public class Waitress {
    ArrayList menus;

    public Waitress(ArrayList menus) {
        this.menus = menus;
    }

    public void printMenu() {
        Iterator menuIterator = menus.iterator();
        while(menuIterator.hasNext()) {
            Menu menu = (Menu)menuIterator.next();
            printMenu(menu.createIterator());
        }
    }

    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());
        }
    }
}
```

Now we just take an ArrayList of menus.

And we iterate through the menus, passing each menu's iterator to the overloaded printMenu() method.

No code changes here.

This looks pretty good, although we've lost the names of the menus, but we could add the names to each menu.