Lecture 10: The Iterator Pattern

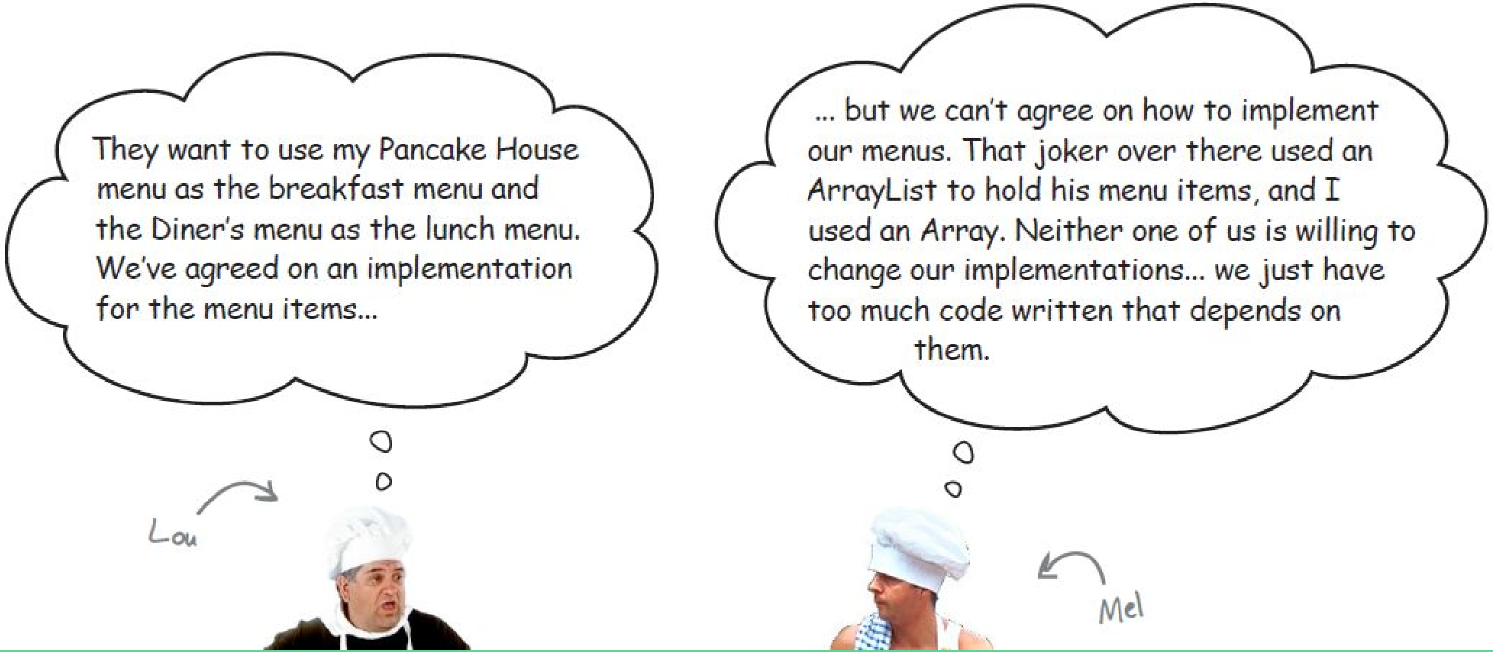
SE313, Software Design and Architecture
Damla Oguz

Chapter 9: The Iterator Pattern

- There are lots of ways to stuff objects into a collection.
- Put them into an Array, a Stack, a List, a Hashmap, 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.
- In this lecture, we are going to see how we can allow our clients to iterate through our objects without ever getting a peek at how we store our objects.
- And we're also going to learn a design principle about object responsibility.

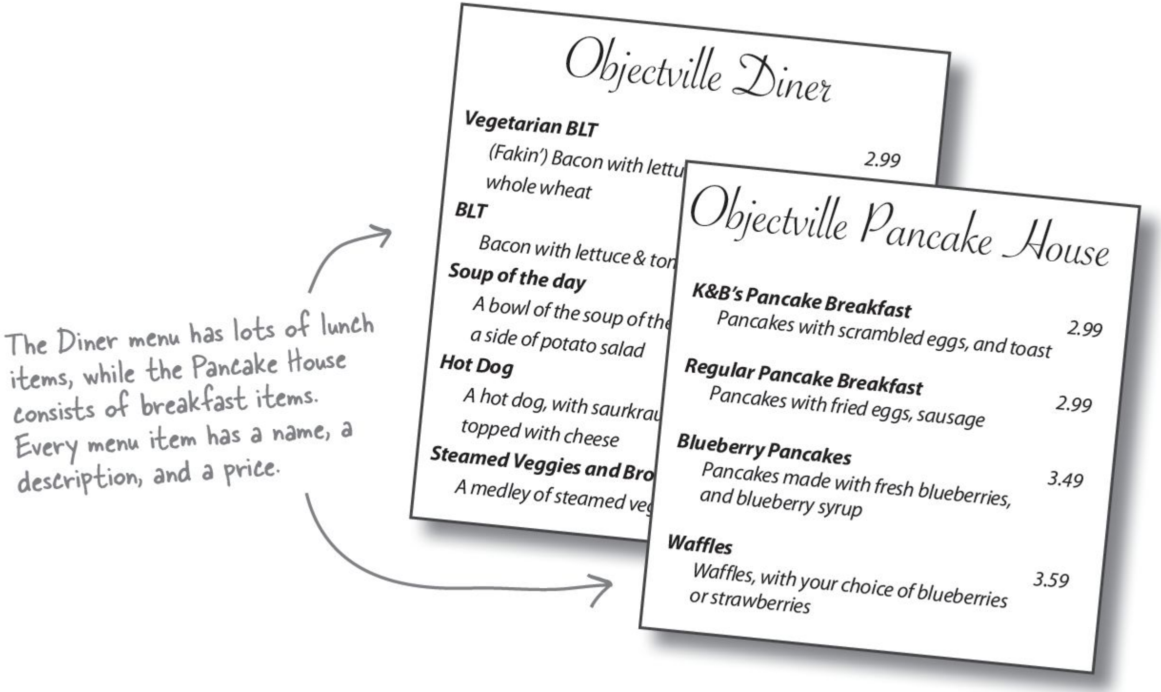
Breaking News: Objectville Diner and Objectville Pancake House Merge

- 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.



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 2.99
whole wheat

BLT
Bacon with lettuce & ton

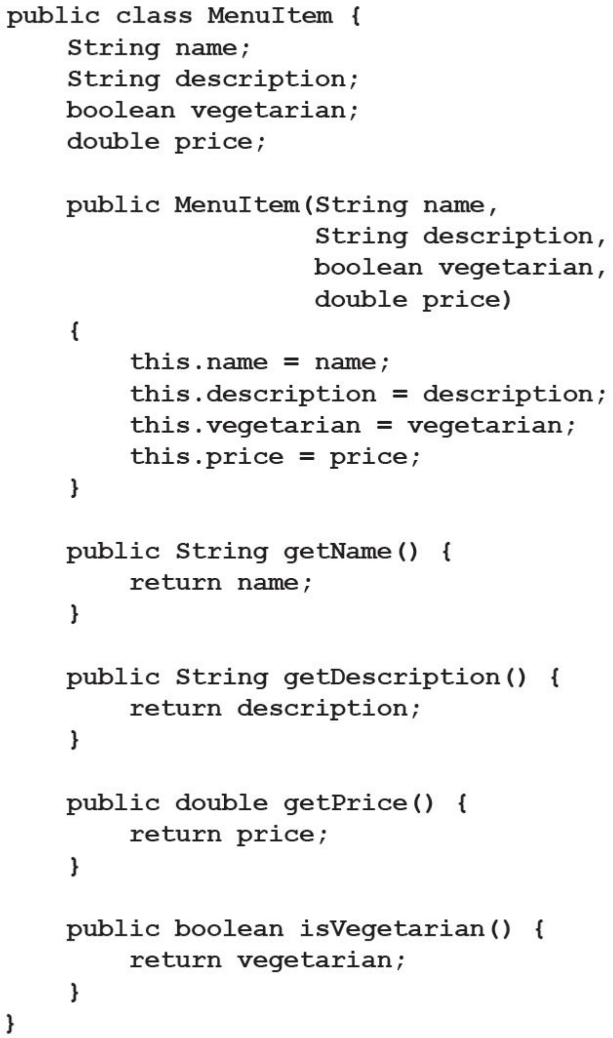
Soup of the day
A bowl of the soup of the
a side of potato salad

Hot Dog
A hot dog, with saurkrau
topped with cheese

Steamed Veggies and Bro
A medley of steamed veg

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;
    }
}
```

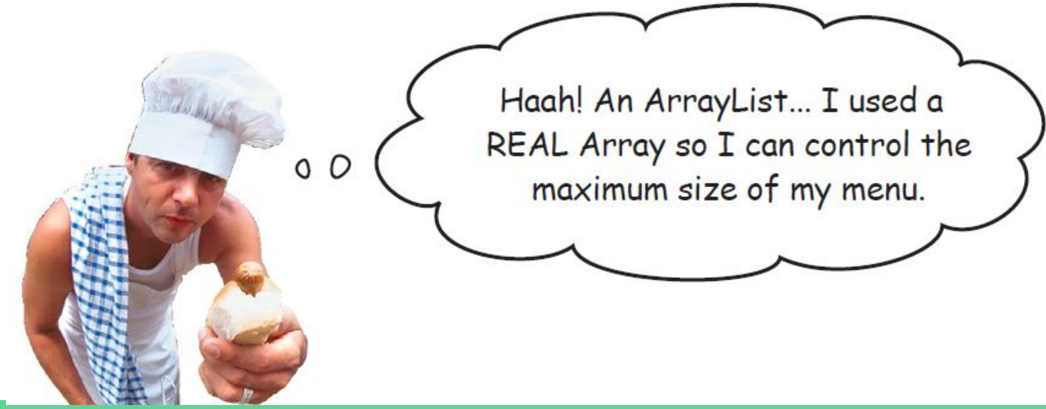
A MenuItem consists of a name, a description, a flag to indicate if the item is vegetarian, and a price. You pass all these values into the constructor to initialize the MenuItem.

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

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.

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


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

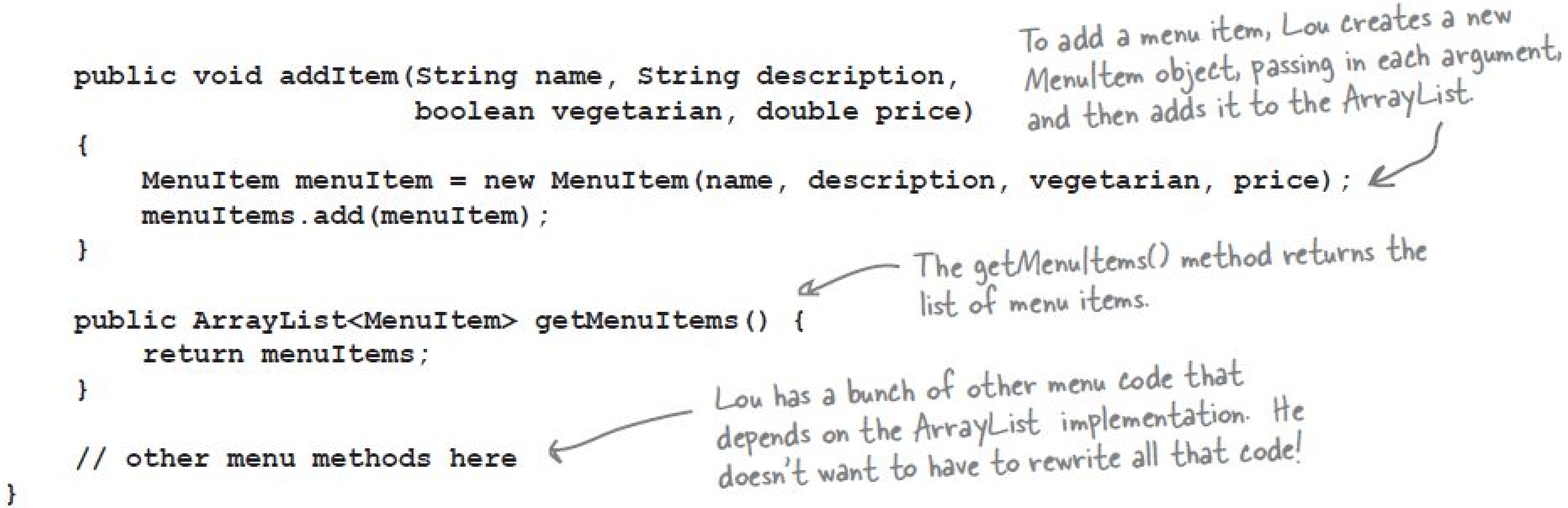
Here's Lou's implementation of the Pancake House menu.

```
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);
    }
```

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.

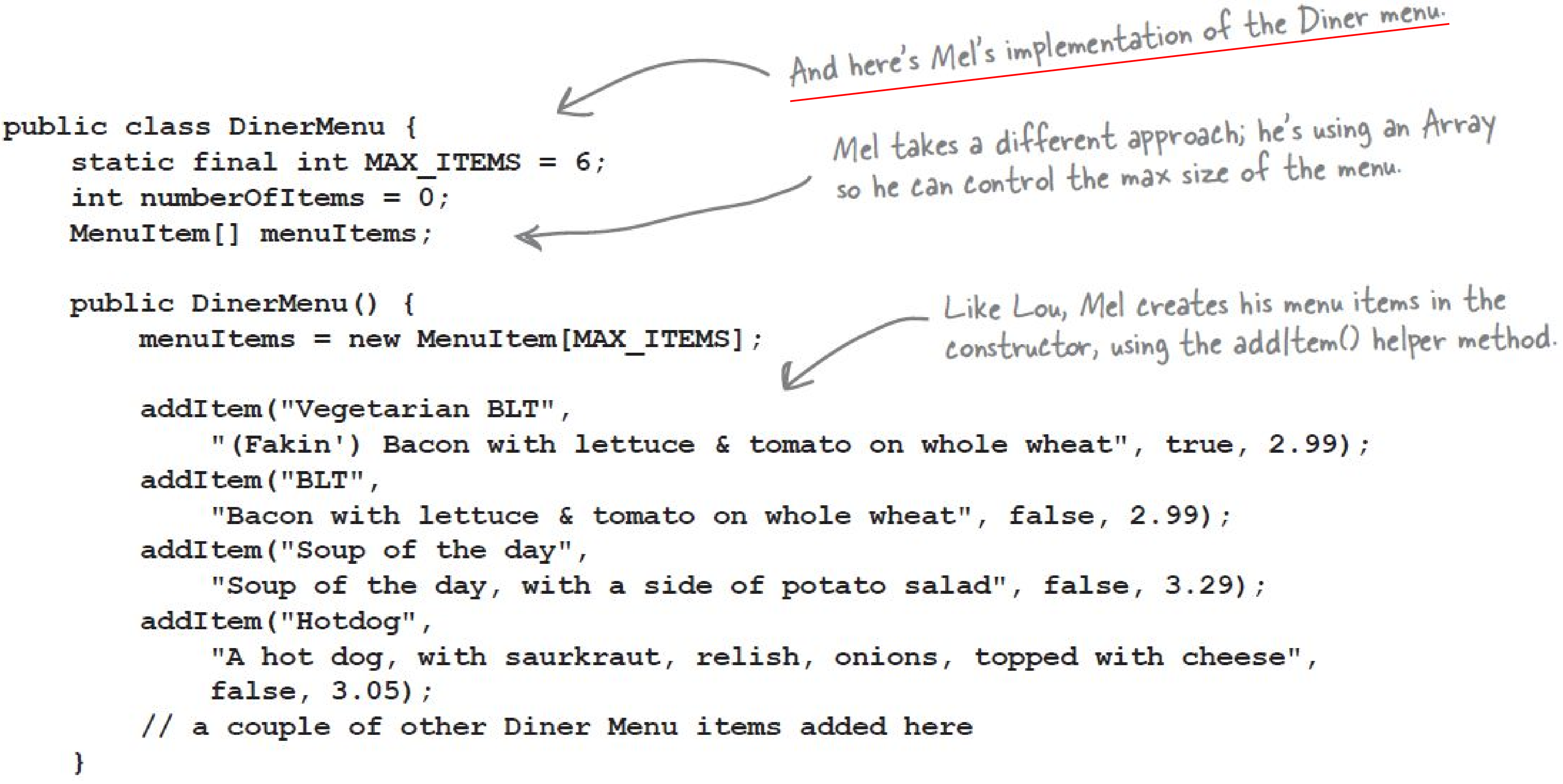


```
    public void addItem(String name, String description,
                        boolean vegetarian, double price)
    {
        MenuItem menuItem = new MenuItem(name, description, vegetarian, price);
        menuItems.add(menuItem);
    }

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

    // other menu methods here
}
```

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.

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.

```
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
    }
```

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).

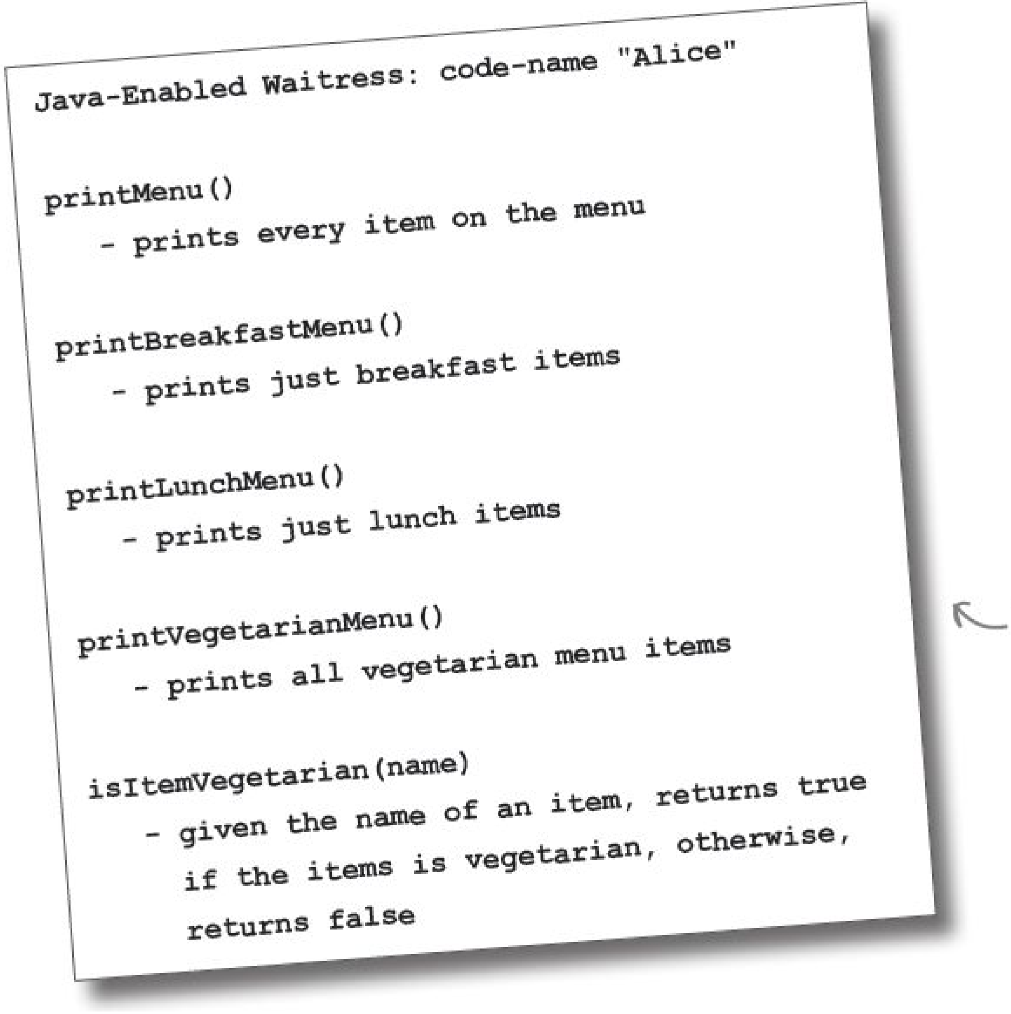
getMenuItems() returns the array of menu items.

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

// 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.

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.
- Let’s check out the spec, and then step through what it might take to implement her…



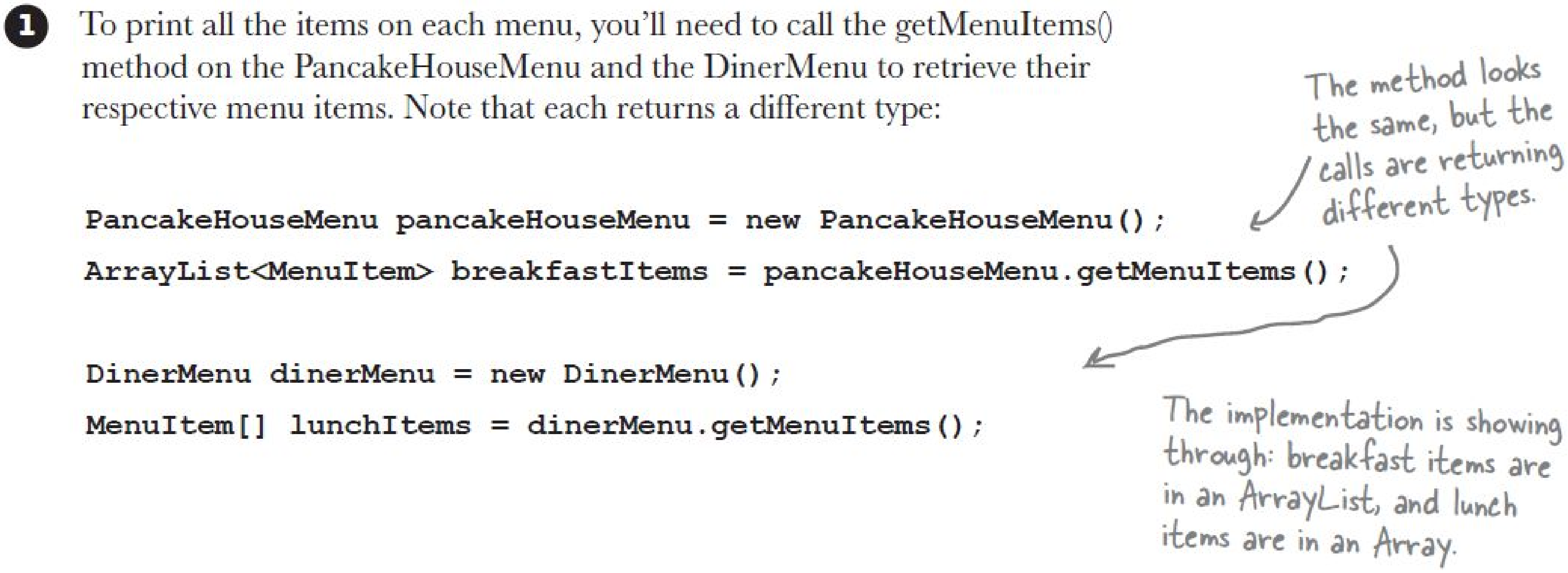
```
Java-Enabled Waitress: code-name "Alice"

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

The Waitress is getting Java-enabled.

What’s the problem with having two different menu representations? (cont.)

- 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 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.

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 = 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.

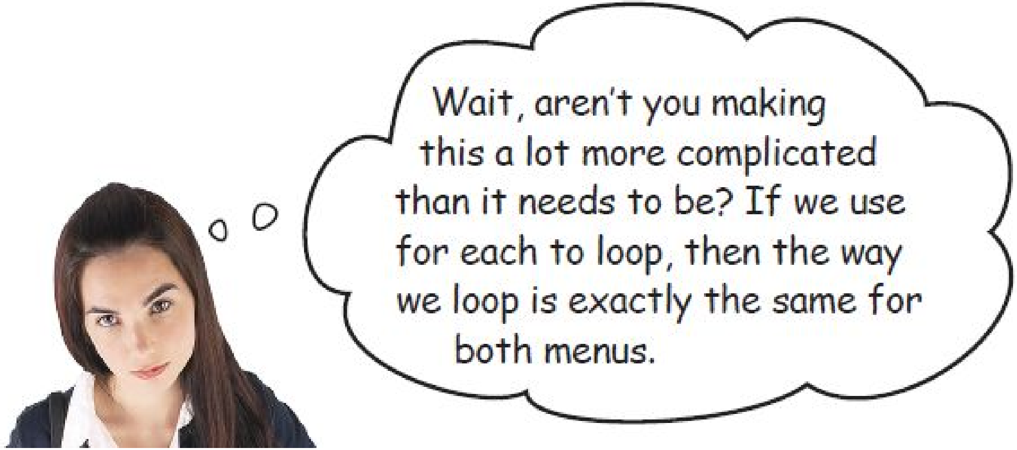
Based on our implementation of printMenu(), the followings apply

- We are coding to the PancakeHouseMenu and DinerMenu concrete implementations, not to an interface.
- 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.
- The Waitress needs to know how each menu represents its internal collection of menu items; this violates encapsulation.
- 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.

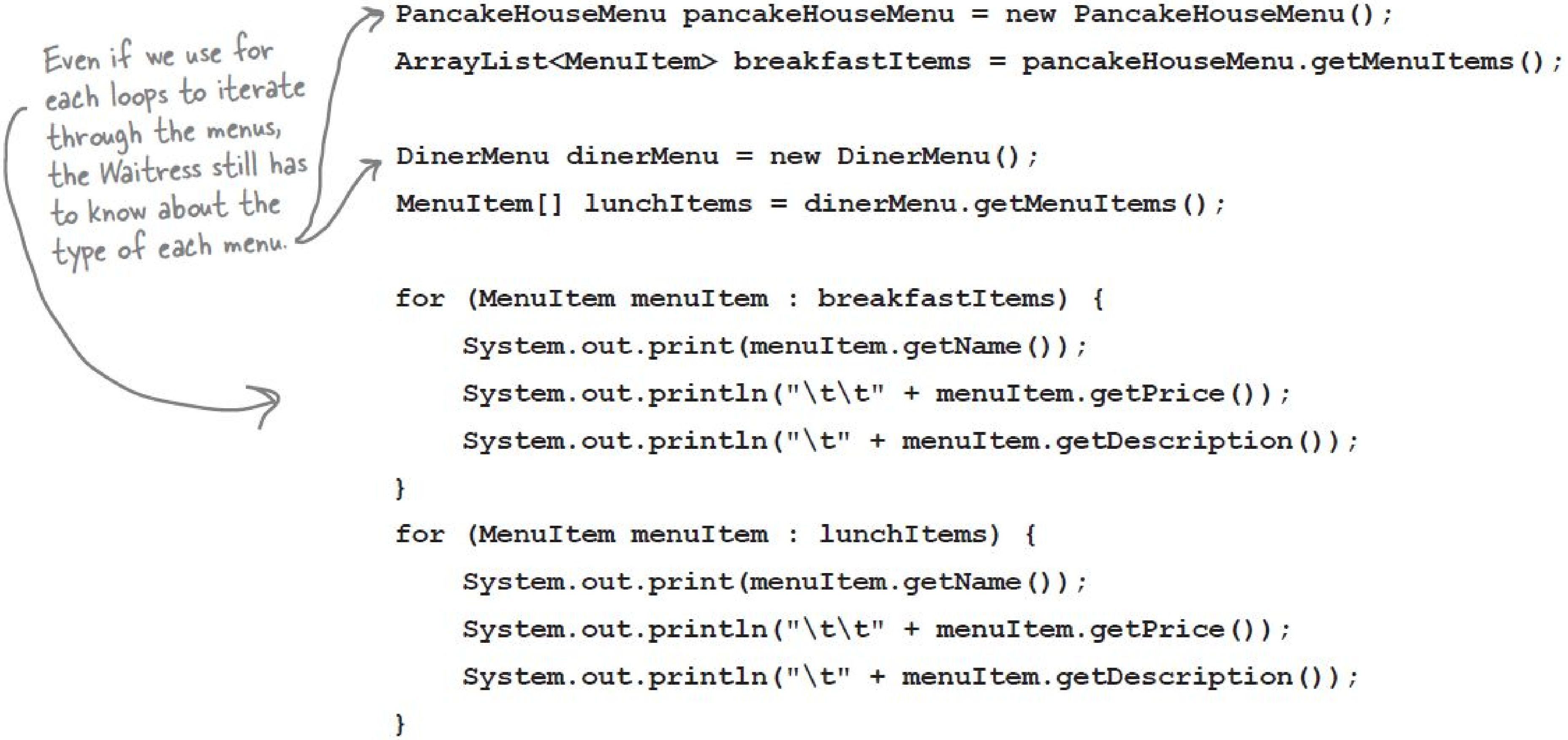
What now?

- Mel and Lou don’t want to change their implementations because it would mean rewriting a lot of code that is in each respective menu class.
- 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?

A question


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.

- Yes, using for each would allow us to hide the complexity of the different kinds of iteration.
- But that doesn't solve the real problem here: that we've got two different implementations of the menus, and the Waitress has to know how each kind of menu is implemented.
- That's not really the Waitress's job.
- We want her to focus on being a waitress, and not have to think about the type of the menus at all.



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

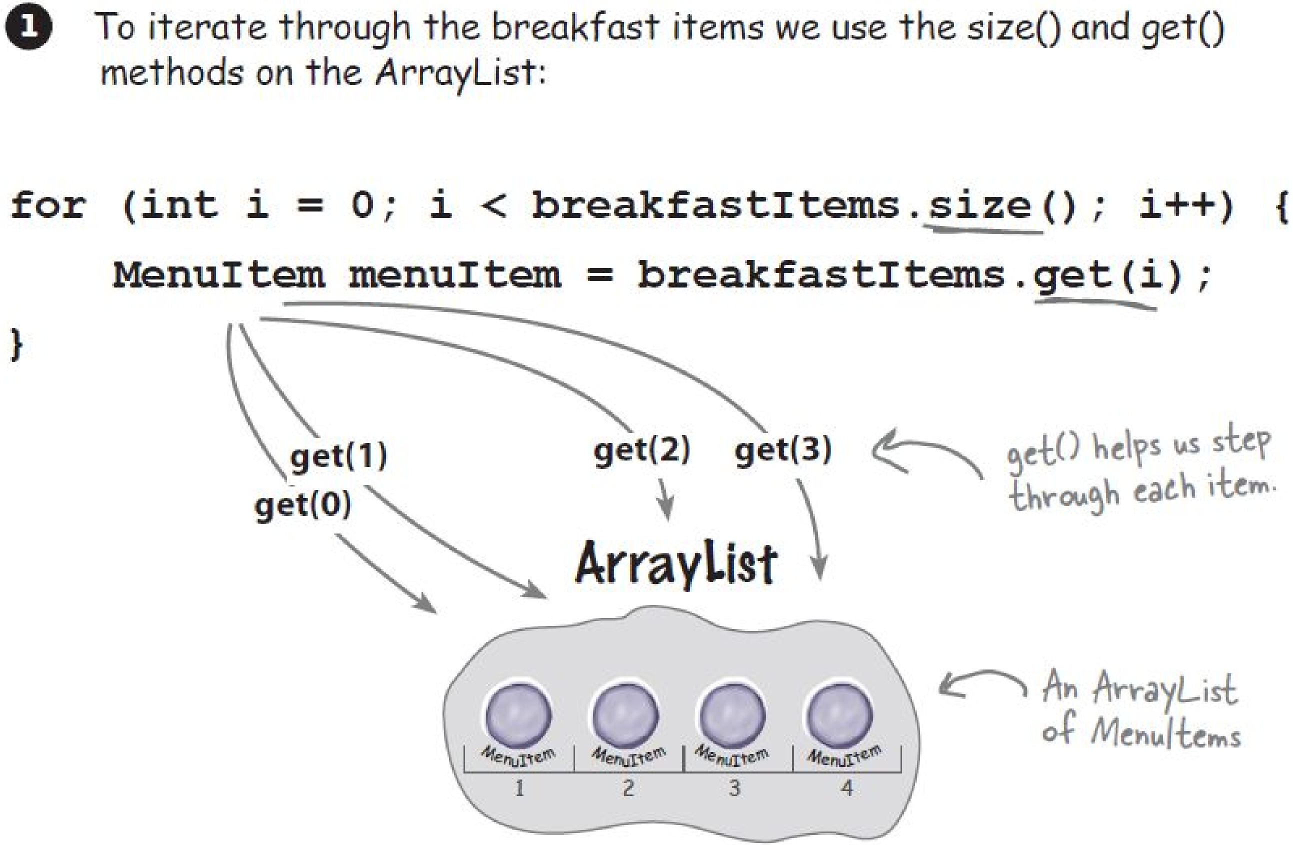
```
PancakeHouseMenu pancakeHouseMenu = new PancakeHouseMenu();
ArrayList<MenuItem> breakfastItems = pancakeHouseMenu.getMenuItems();

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());
}
```

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

Can we encapsulate the iteration?

- Remember the design principle which is about the encapsulation of parts that 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…

Can we encapsulate the iteration? (cont.)



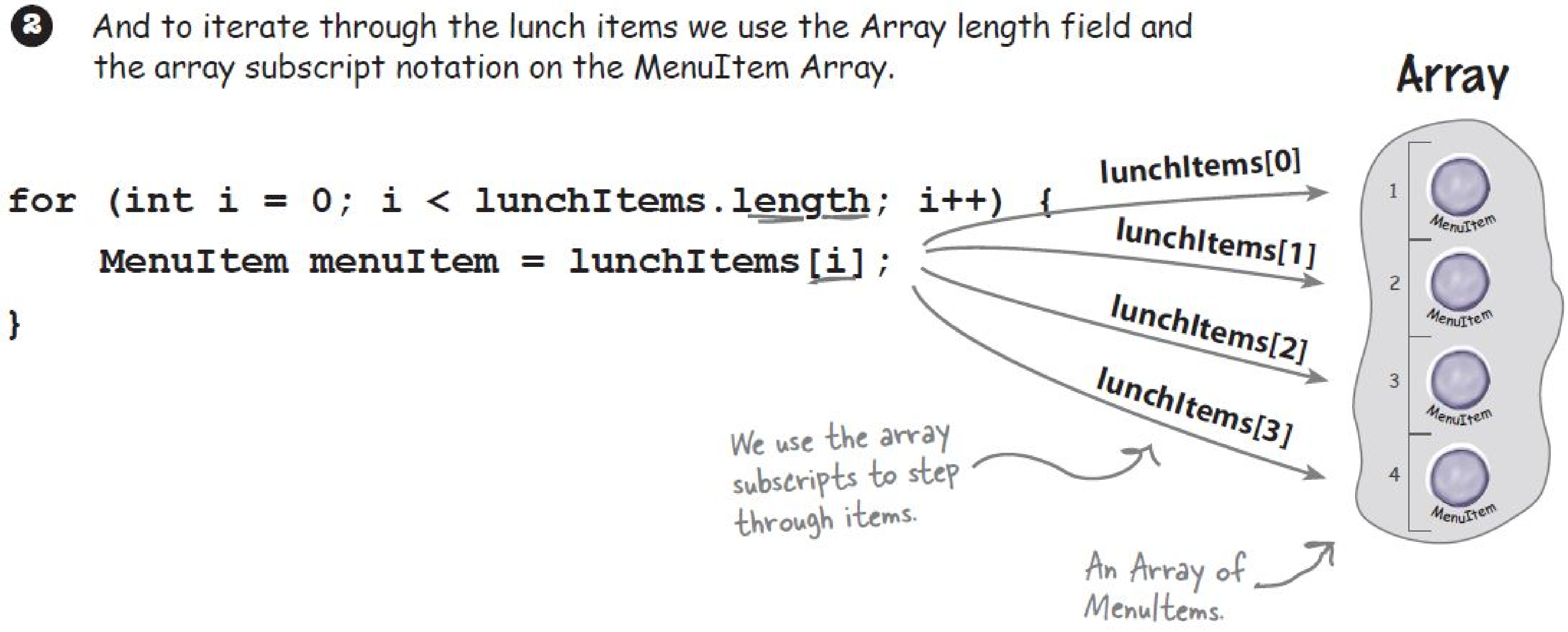
1 To iterate through the breakfast items we use the size() and get() methods on the ArrayList:

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

get() helps us step through each item.

An ArrayList of MenuItems

Can we encapsulate the iteration? (cont.)



❷ 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.

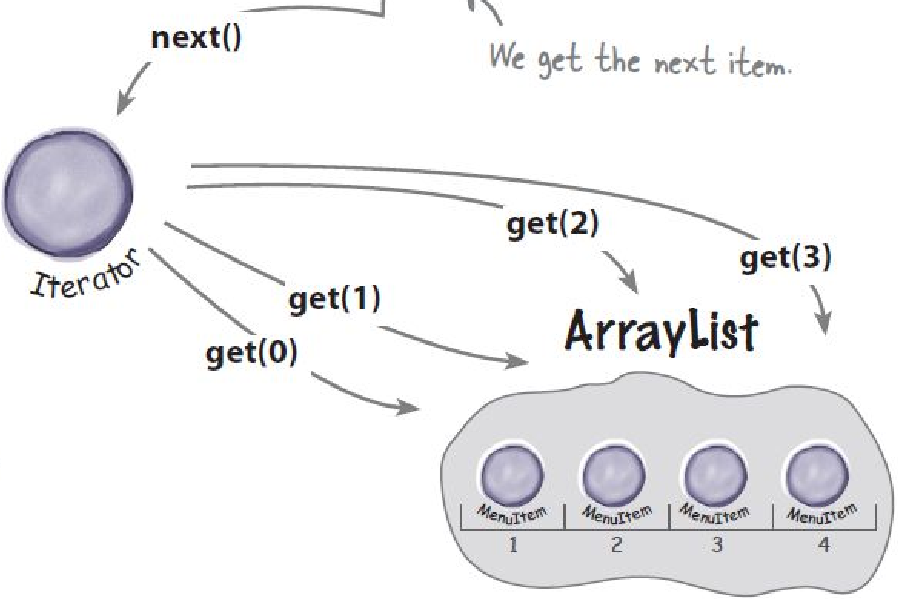
3. Now what if we create an object, let's call it an Iterator, that encapsulates the way we iterate through a collection of objects? Let's try this on the ArrayList

We ask the breakfastMenu for an iterator of its MenuItems.

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

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

And while there are more items left...

We get the next item.

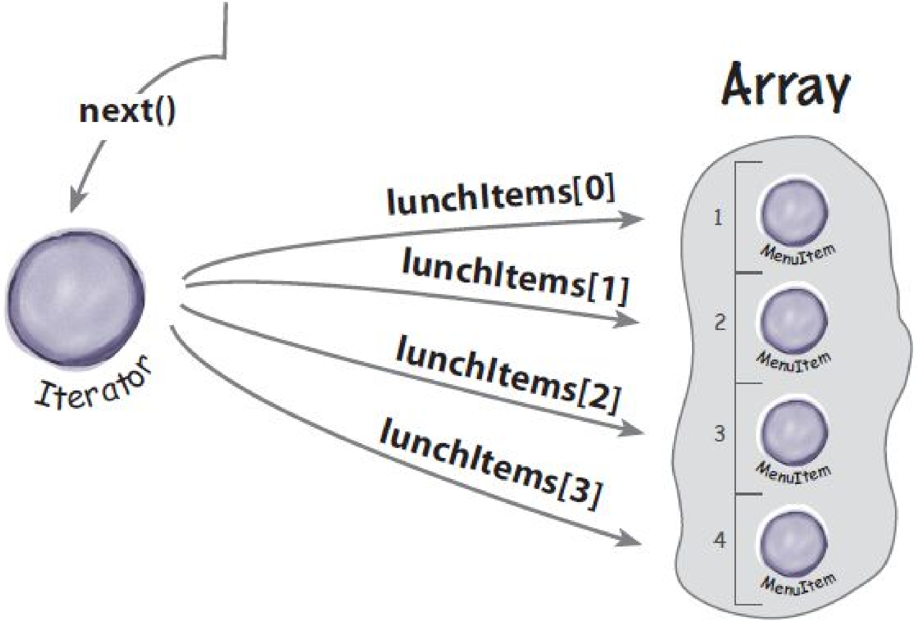


The client just calls hasNext() and next(); behind the scenes the iterator calls get() on the ArrayList.

4 Let's try that on the Array too:

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

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

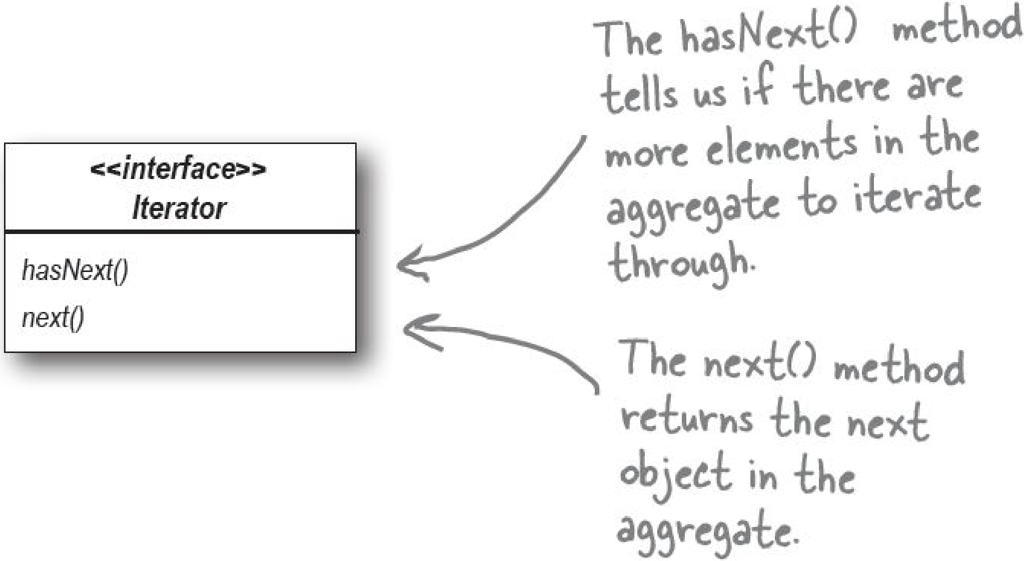
Wow, this code is exactly the same as the breakfastMenu code.



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

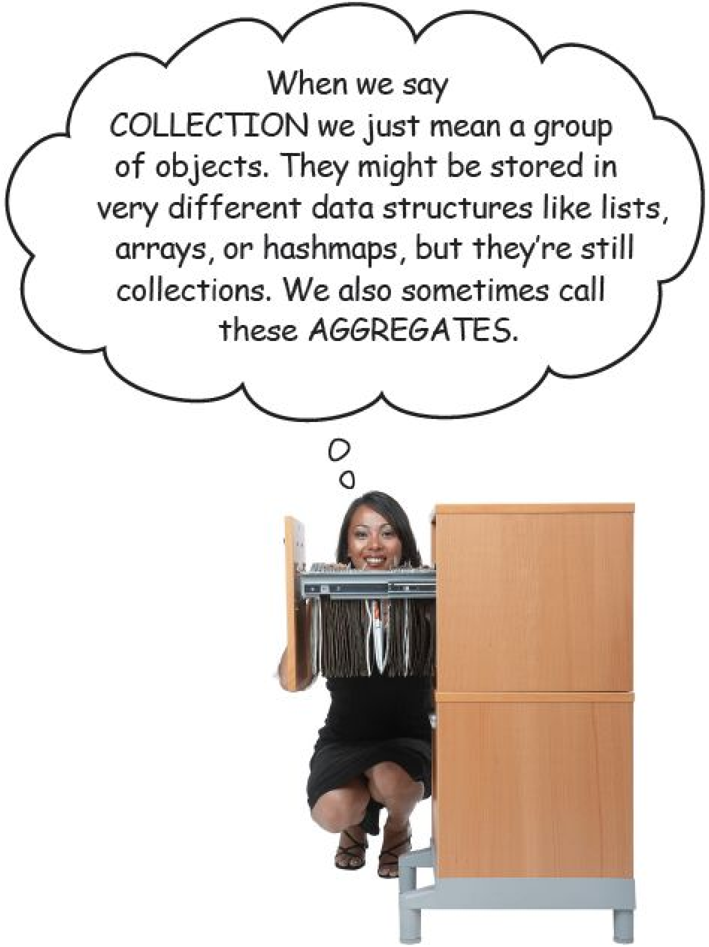
Meet the Iterator Pattern

- Well, it looks like our plan of encapsulating iteration just might actually work; and as you've probably already guessed, it is 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:


<<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.

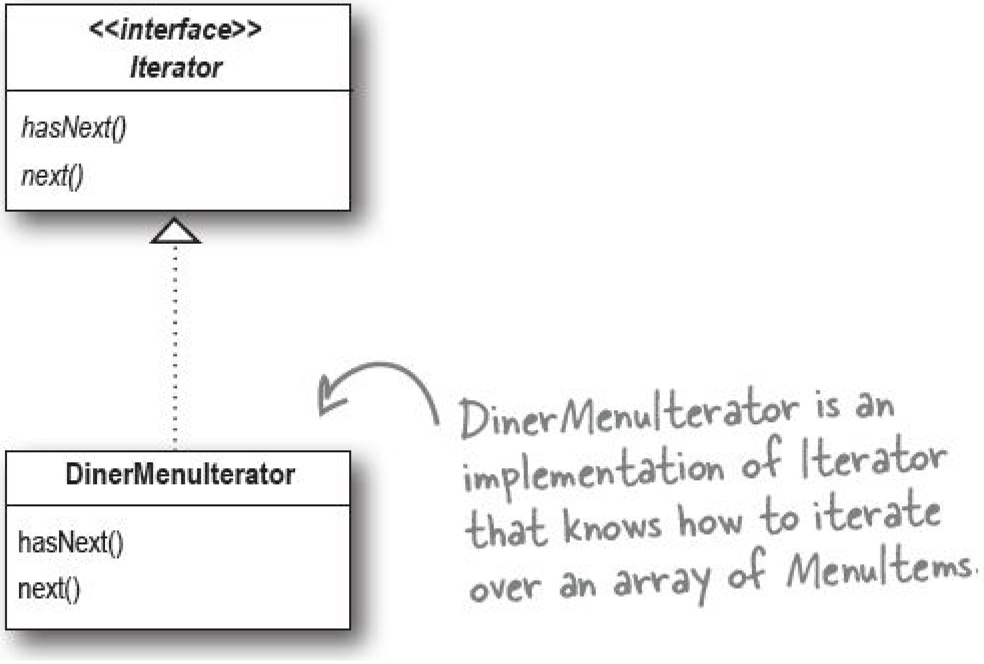
Meet the Iterator Pattern (cont.)

- Now, once we have this interface, we can implement Iterators for any kind of collection of objects: arrays, lists, hashmaps,...pick your favorite collection of objects.


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.

Meet the Iterator Pattern (cont.)

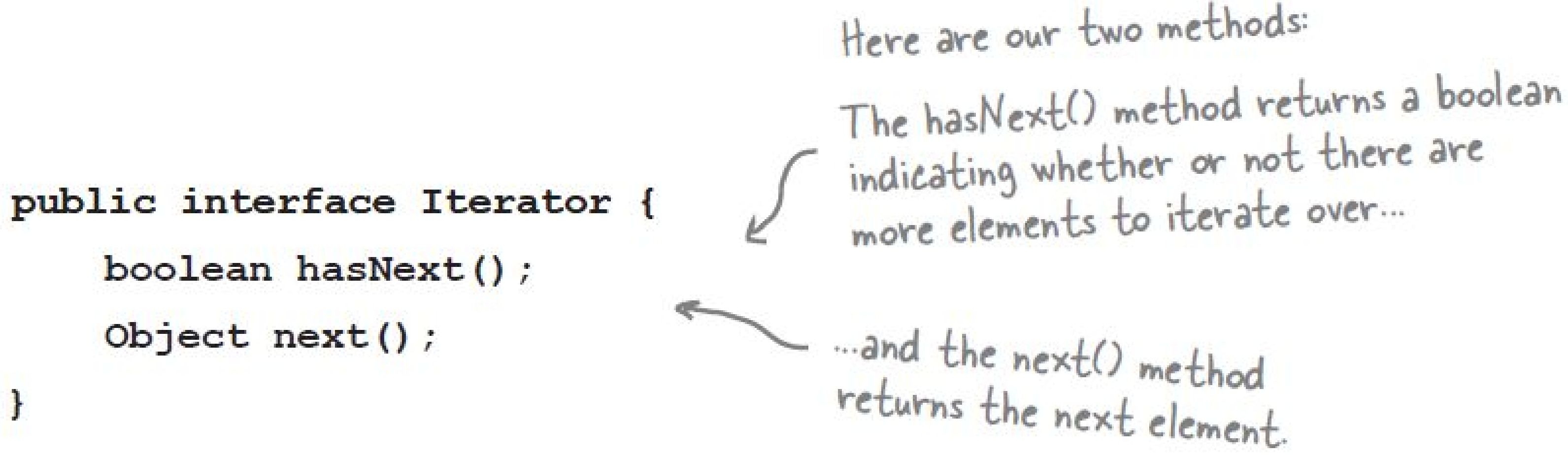
- Let’s say we wanted to implement the Iterator for the Array used in the DinerMenu. It would look like this:



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

Adding an Iterator to DinerMenu

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



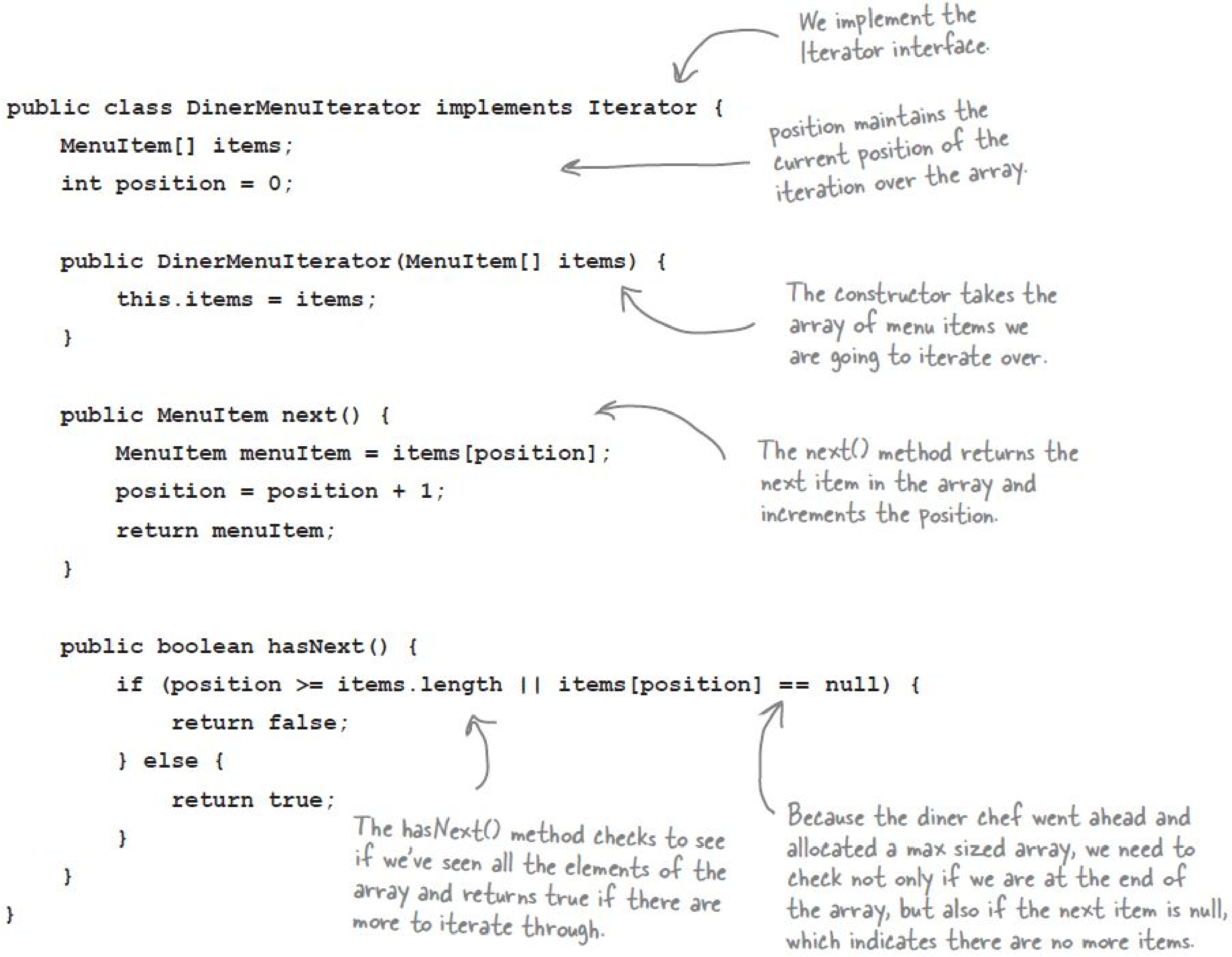
```
public interface Iterator {
    boolean hasNext();
    Object next();
}
```

Here are 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.

- And 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 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.

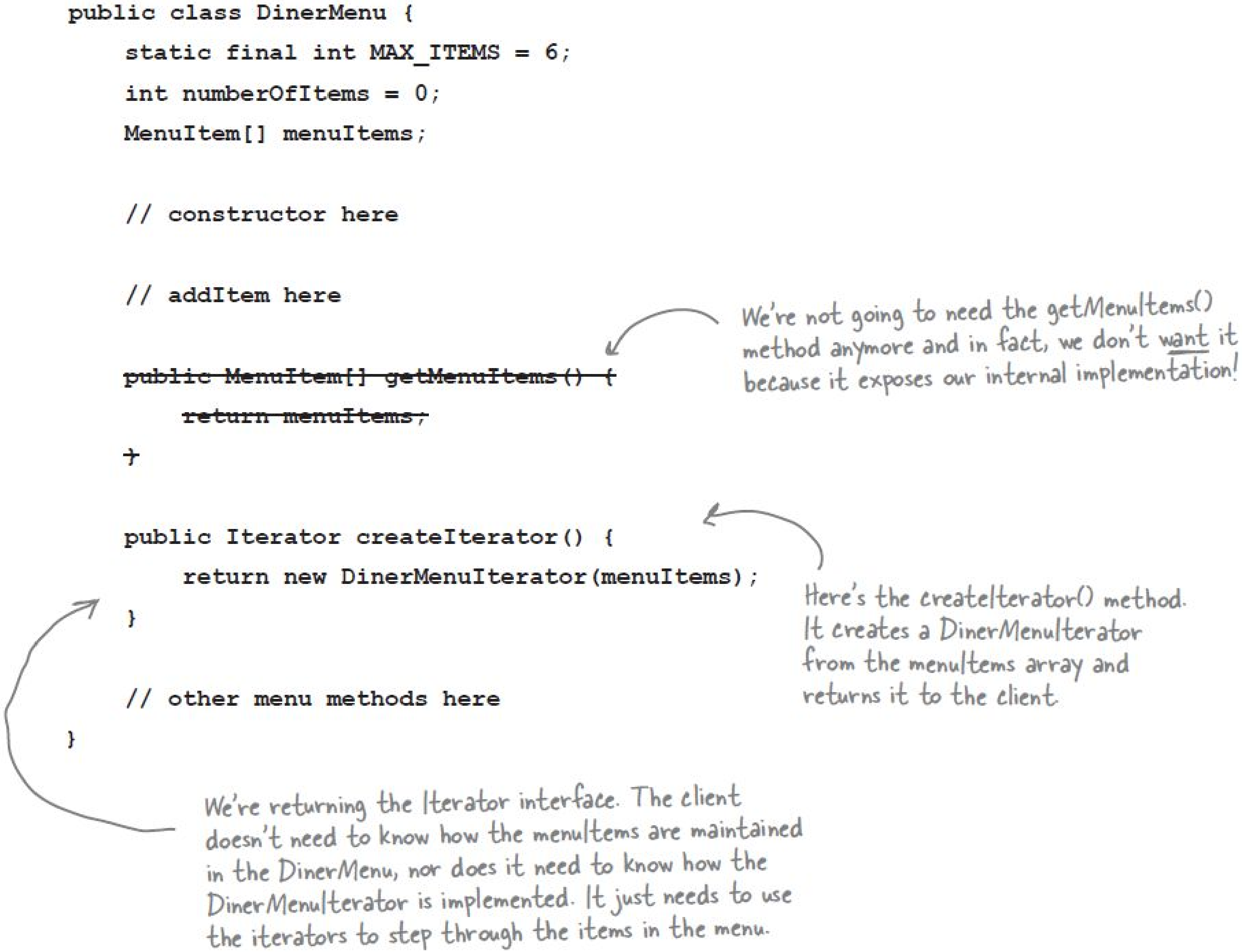
The next() method returns the next item in the array and increments the position.

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.

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
}
```

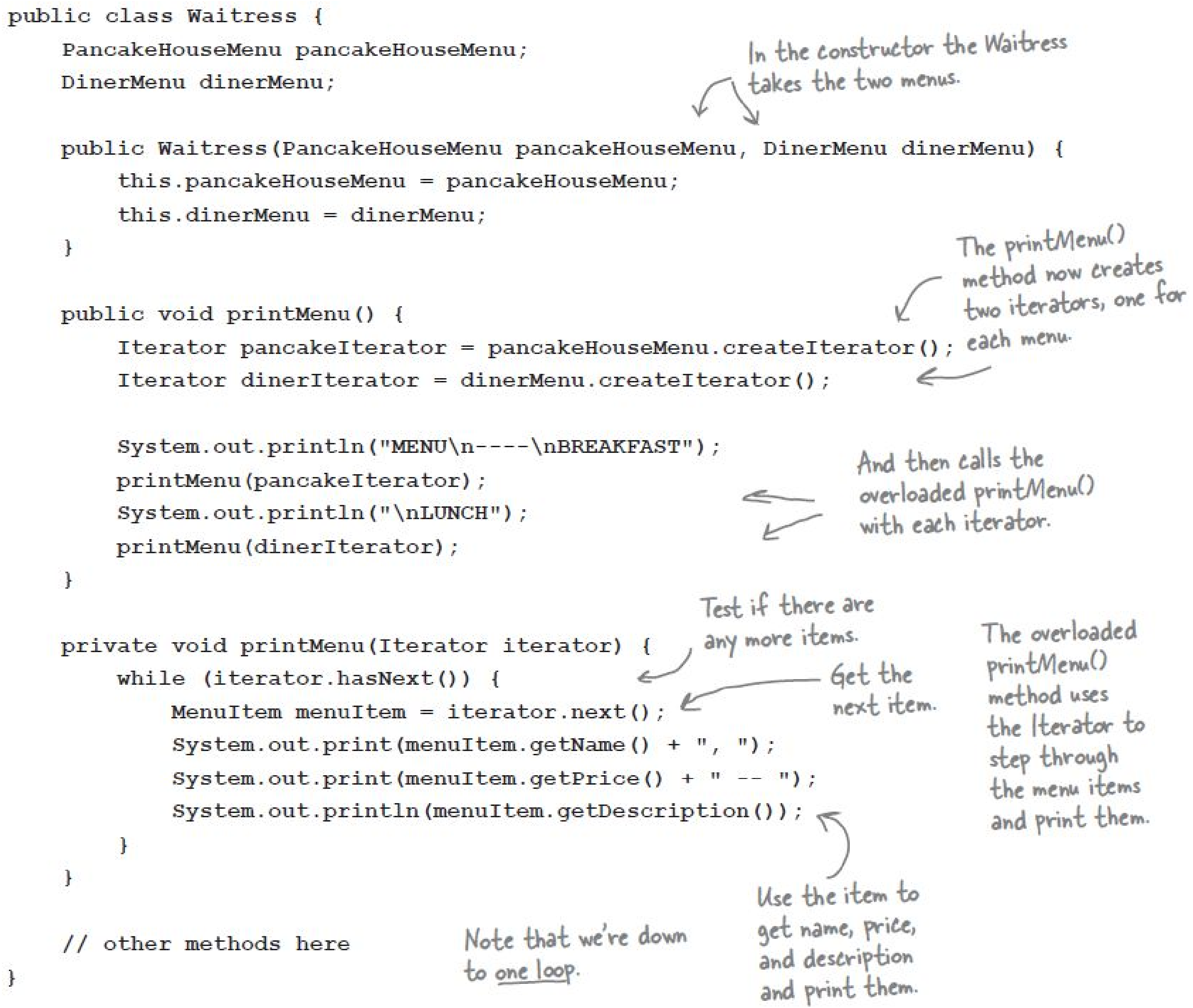
We're not going to need the getMenuItems() method anymore and in fact, we don't want it because it exposes our internal implementation!

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.

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 createIterator() method on each menu to retrieve the Iterator and pass it to the new method.



```
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.

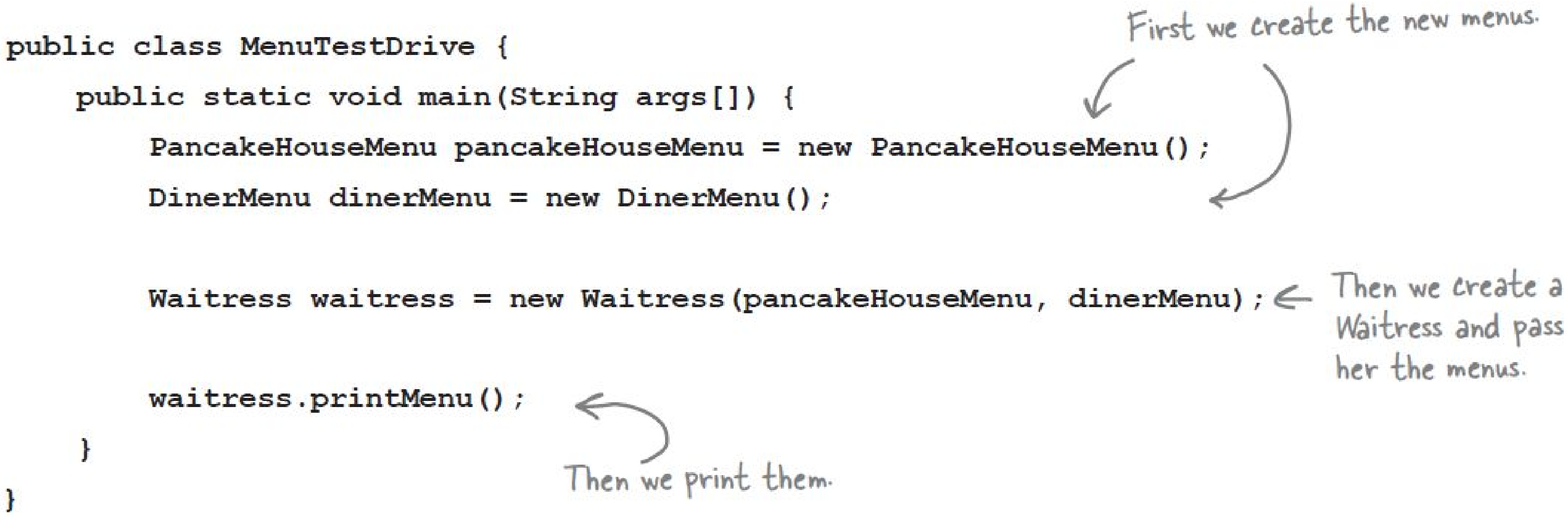
The overloaded printMenu() method uses the Iterator to step through the menu items and print them.

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

Note that we're down to one loop.

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…



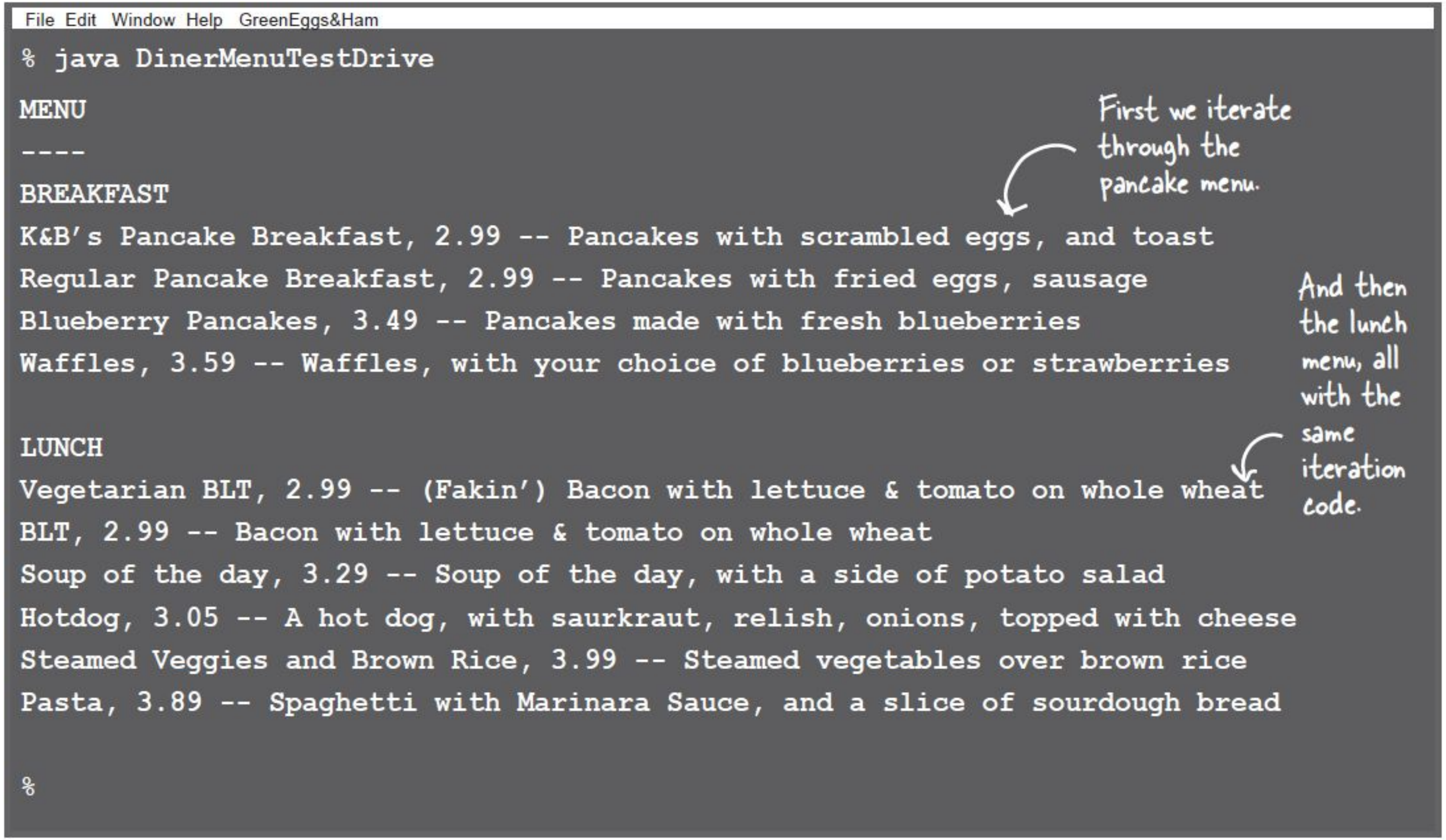
```
public class MenuTestDrive {
    public static void main(String args[]) {
        PancakeHouseMenu pancakeHouseMenu = new PancakeHouseMenu();
        DinerMenu dinerMenu = new DinerMenu();

        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.

Testing our code (cont.)



```
% 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?

- Objectville cooks 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 createIterator() 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…

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

What have we done so far? (cont.)

Hard to Maintain Waitress Implementation

- The Menus are not well encapsulated; we can see the Diner is using an ArrayList and the Pancake House an Array.
- We need two loops to iterate through the MenuItems.

New Waitress Powered by Iterator

- The Menu implementations are now encapsulated. The Waitress has no idea how the Menus hold their collection of menu items.
- All we need is a loop that polymorphically handles any collection of items as long as it implements Iterator.

What have we done so far? (cont.)

Hard to Maintain Waitress Implementation

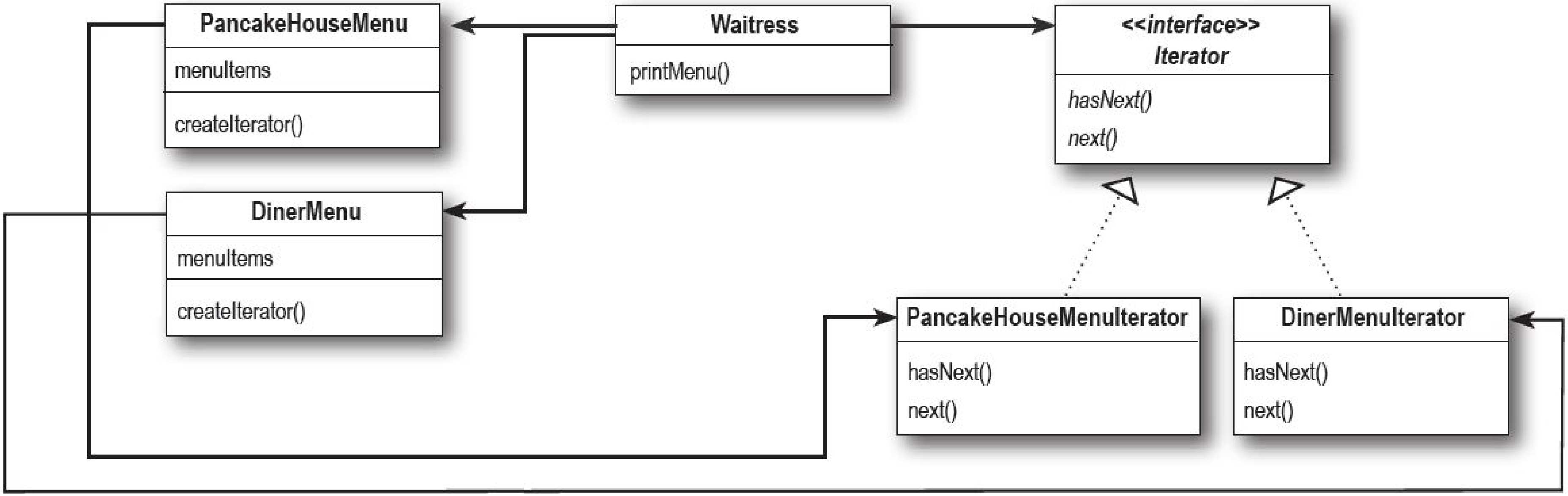
- The Waitress is bound to concrete classes (MenuItem[] and ArrayList).
- The Waitress is bound to two different concrete Menu classes, despite their interfaces being almost identical.

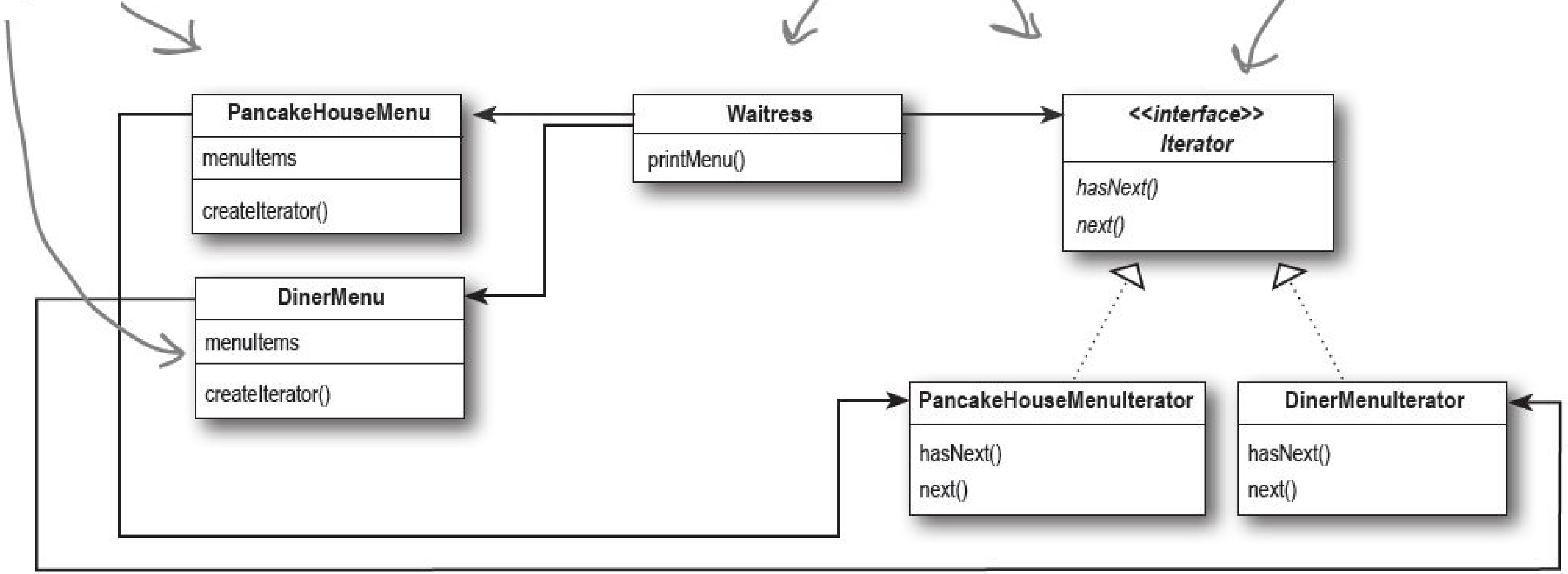
New Waitress Powered by Iterator

- The Waitress now uses an interface (Iterator).
- 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.

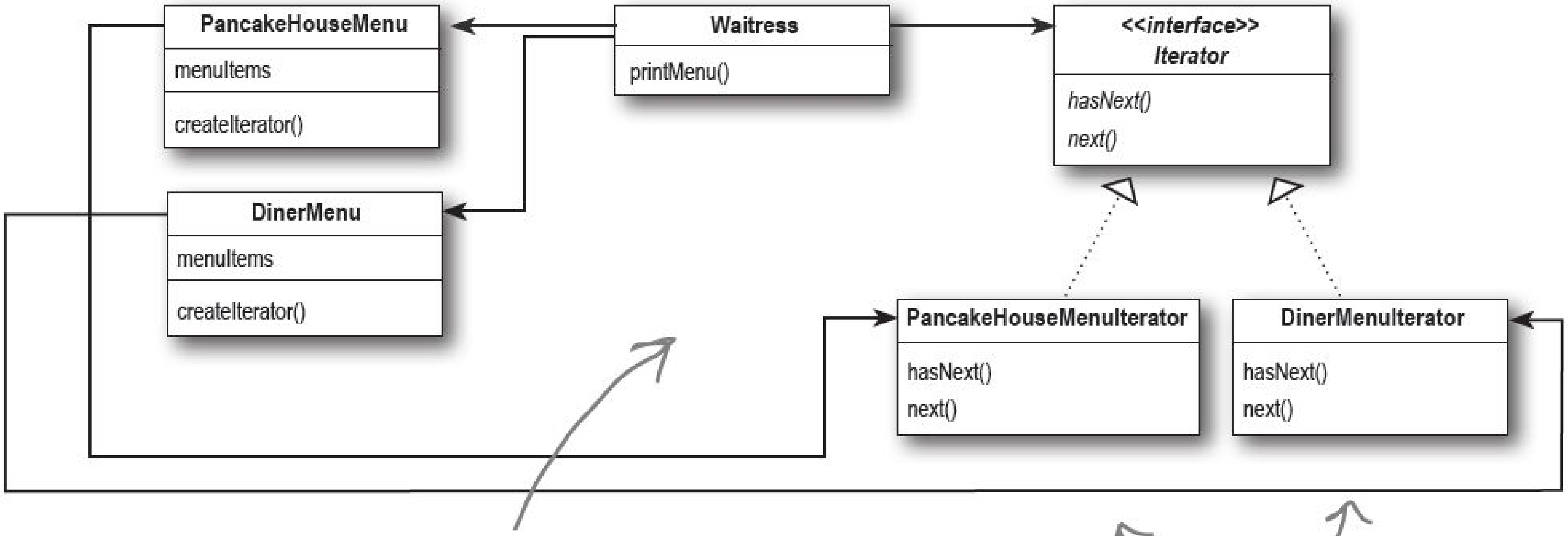




These two menus implement the same exact set of methods, but they aren't implementing the same interface. We're going to fix this and free the Waitress from any dependencies on concrete Menus.

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 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.

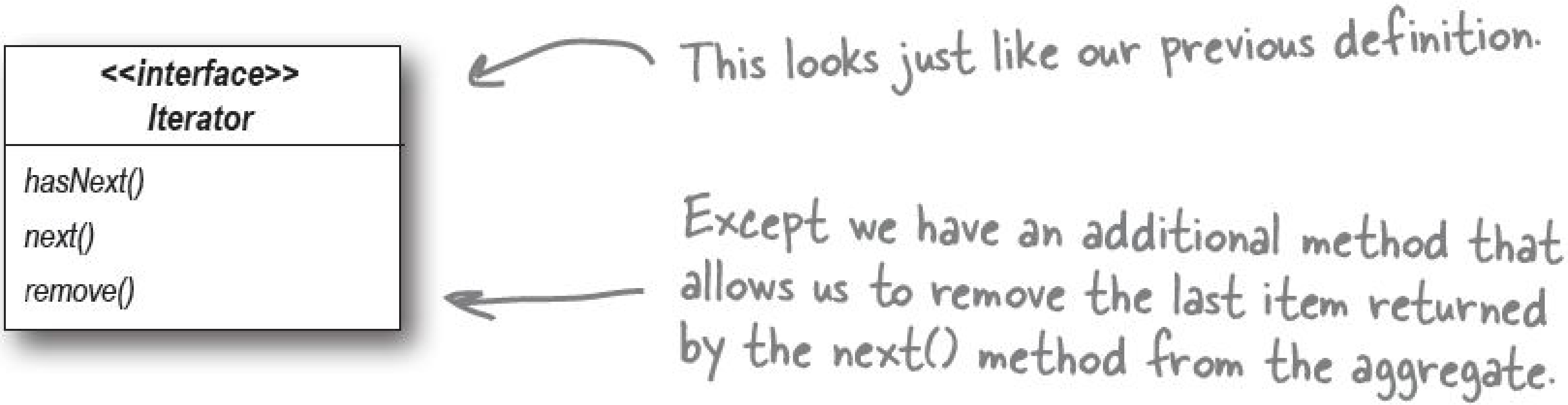
PancakeHouseMenu and DinerMenu implement the new createIterator() method; they are responsible for creating the iterator for their respective menu items' implementations.

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.

Making some improvements…(cont.)

- 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.

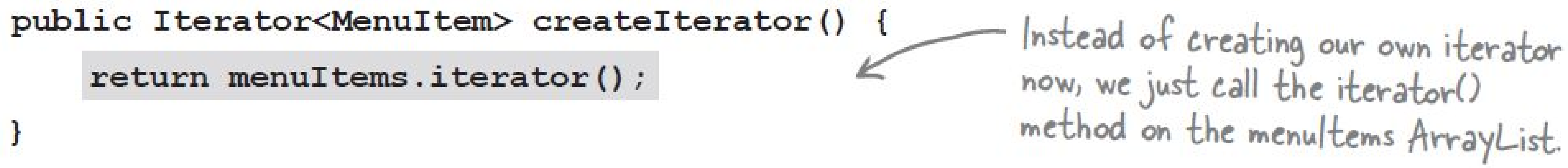
- This is going to be a piece of cake: we just need to change the interface that both PancakeHouseMenuIterator and DinerMenuIterator extend, right?
- Almost… actually, it’s even easier than that. Not only does java.util have its own Iterator interface, but ArrayList has an iterator() method that returns an iterator.

Making some improvements...(cont.)

- 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).

Cleaning things up with java.util.Iterator

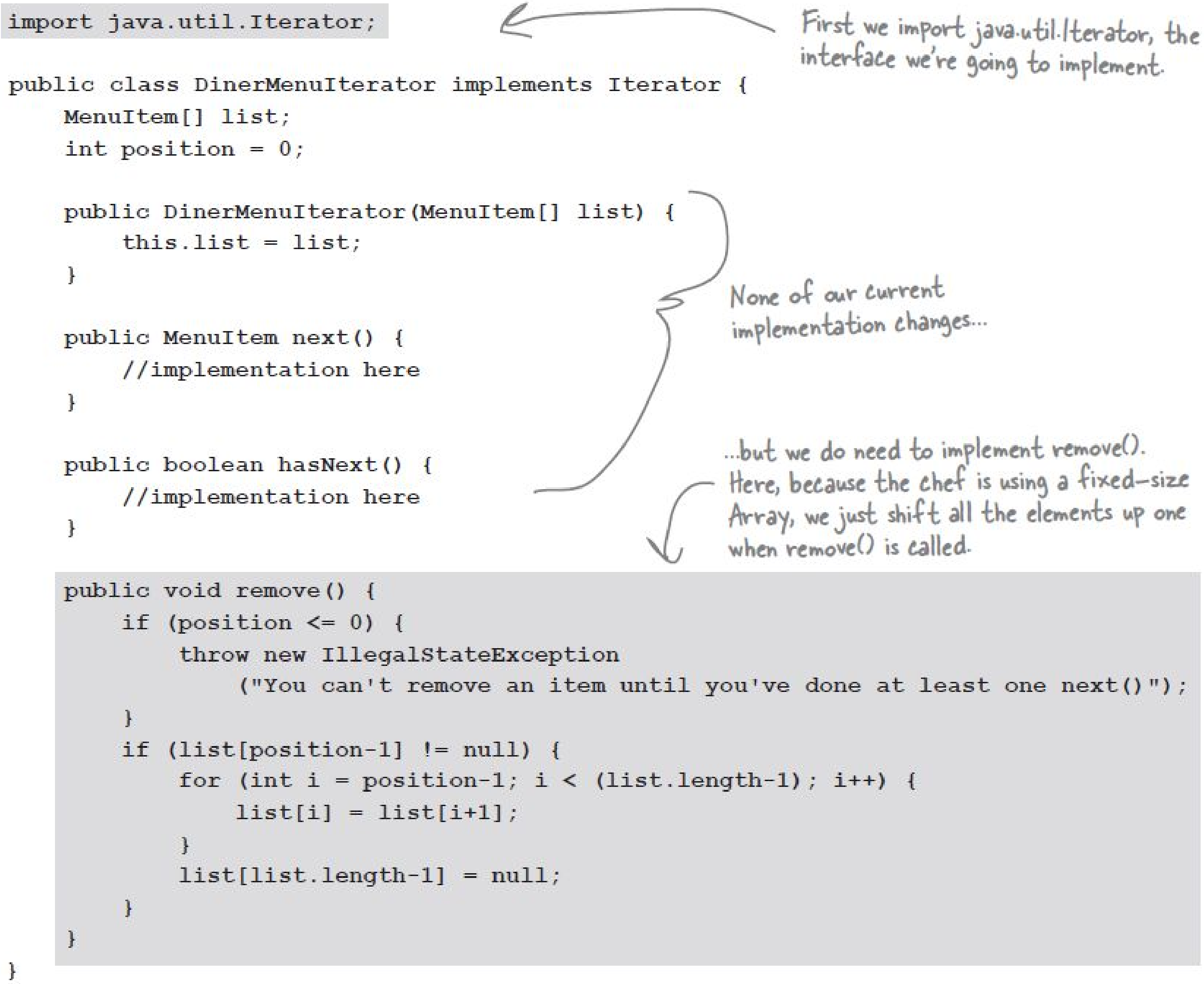
- Let’s start with the PancakeHouseMenu. Changing it over to java.util.Iterator is going to be easy.
- We just delete the PancakeHouseMenuIterator class, add an import java.util.Iterator to the top of PancakeHouseMenu and change one line of the PancakeHouseMenu:



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

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

- 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 MenuItem 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;
        }
    }
}
```

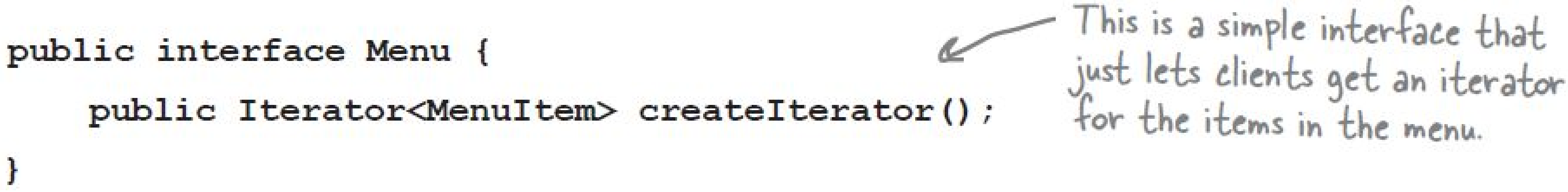
First we import java.util.Iterator, the interface we're going to implement.

None of our current implementation changes...

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

We are almost there...

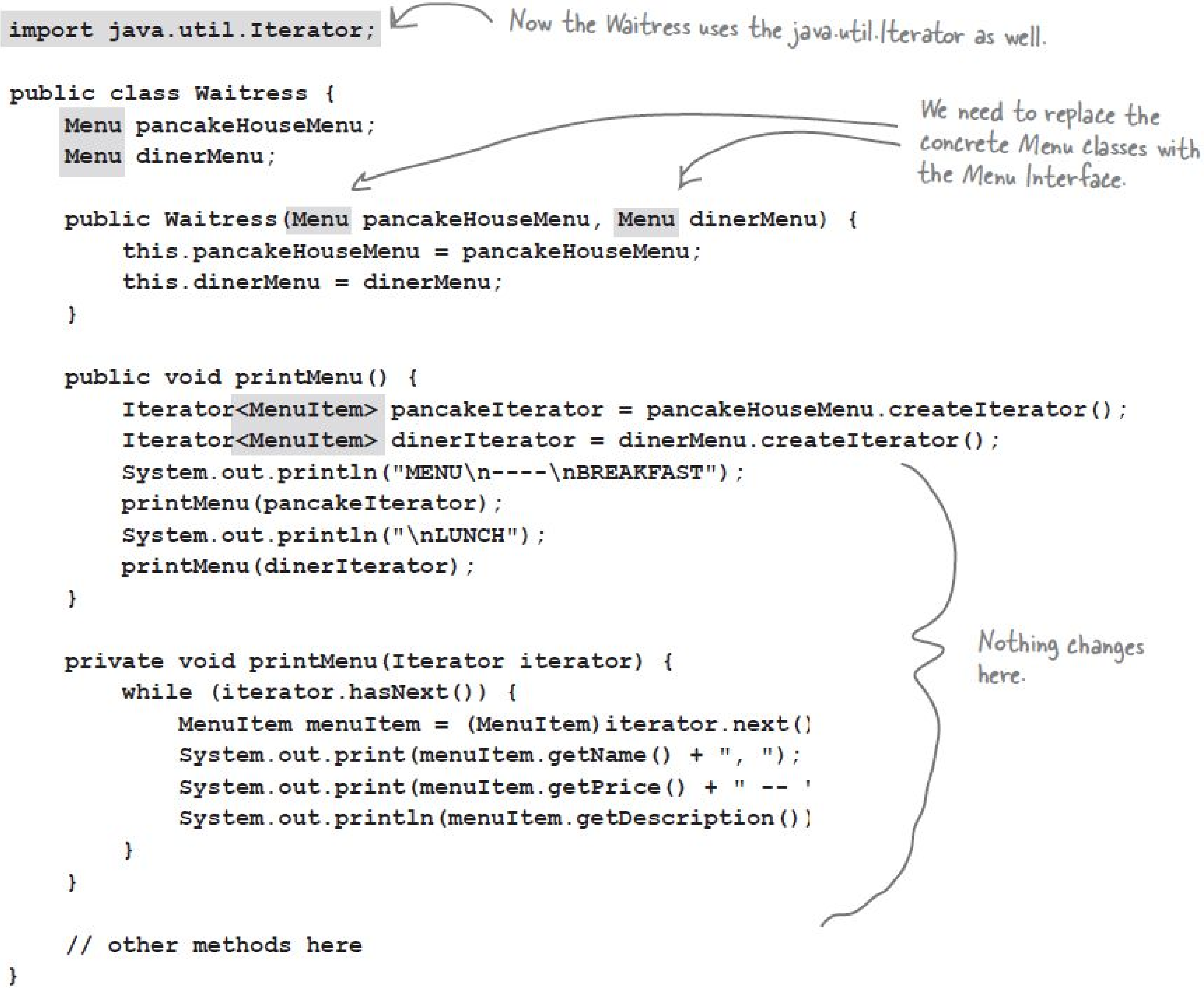
- We just need to give the Menus a common interface and rework the Waitress a little.
- The Menu interface is quite simple: we might want to add a few more methods to it eventually, like addItem(), but for now we will let the chefs control their menus by keeping that method out of the public interface:



```
public interface Menu {
    public Iterator<MenuItem> 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<MenuItem> pancakeIterator = pancakeHouseMenu.createIterator();
        Iterator<MenuItem> 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.

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.

What does this get us? (cont.)

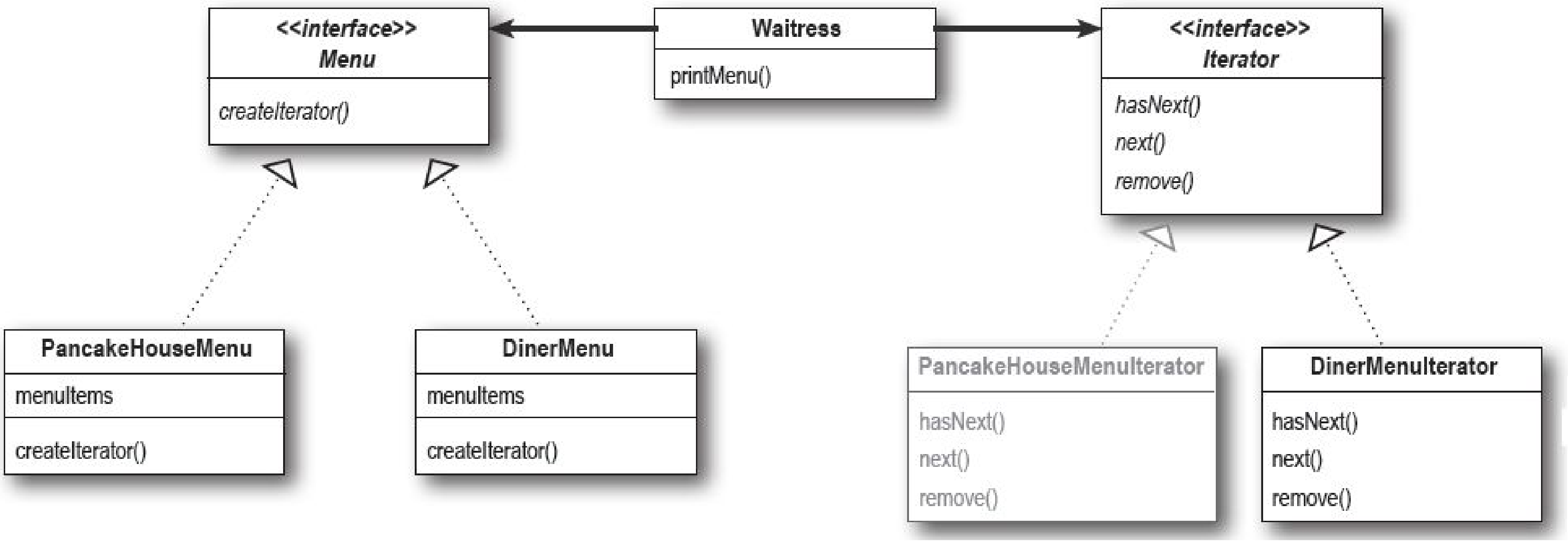
- The PancakeHouseMenu and DinerMenu classes implement an interface, Menu.
- Waitress can refer to each menu object using the interface rather than the concrete class. So, we’re reducing the dependency between the Waitress and the concrete classes by “programming to an interface, not an implementation.”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



<<interface>>
Menu
createIterator()

Waitress
printMenu()

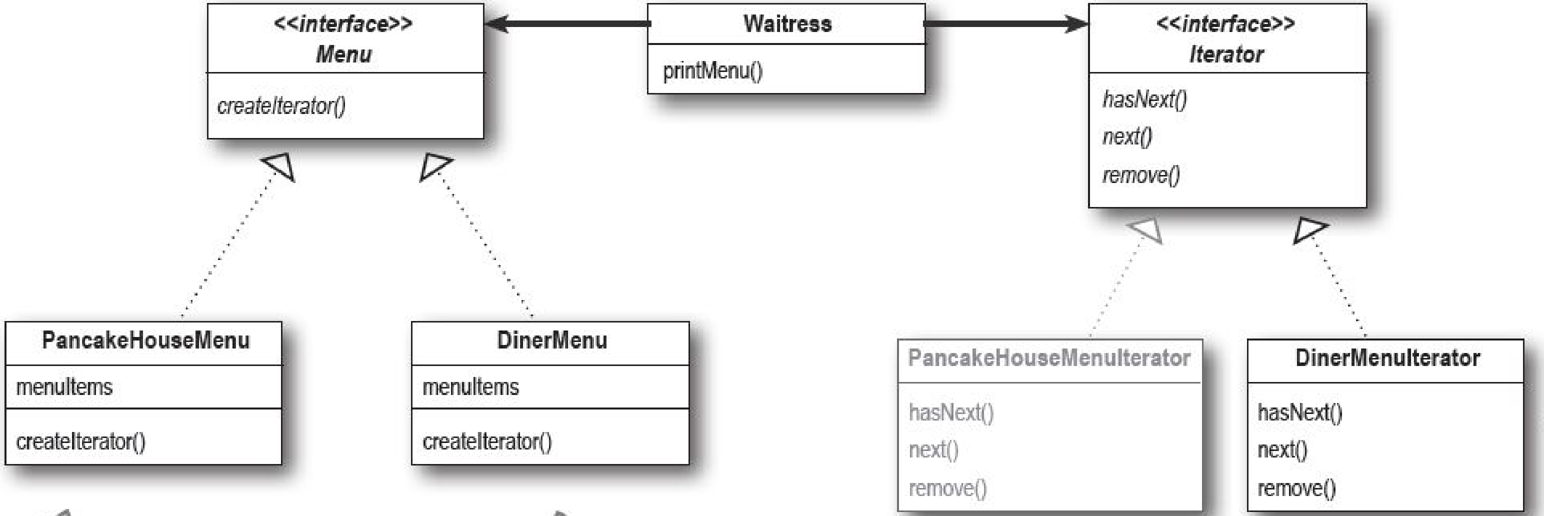
<<interface>>
Iterator
hasNext()
next()
remove()

PancakeHouseMenu
menuItems
createIterator()

DinerMenu
menuItems
createIterator()

PancakeHouseMenuIterator
hasNext()
next()
remove()

DinerMenuIterator
hasNext()
next()
remove()



<<interface>>
Menu

createIterator()

Waitress

printMenu()

<<interface>>
Iterator

hasNext()
next()
remove()

PancakeHouseMenu

menuItems

createIterator()

DinerMenu

menuItems

createIterator()

PancakeHouseMenuIterator

hasNext()
next()
remove()

DinerMenuIterator

hasNext()
next()
remove()

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

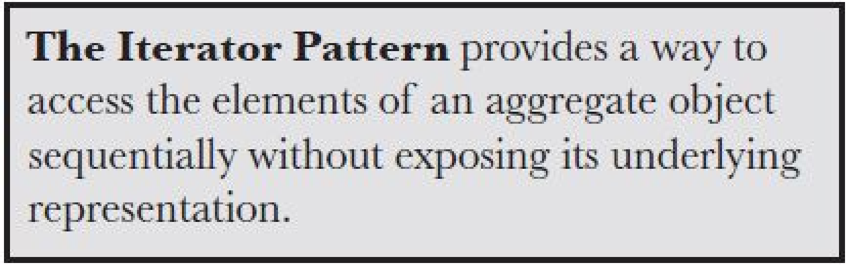
We're now using the ArrayList iterator supplied by java.util. We don't need this anymore.

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:



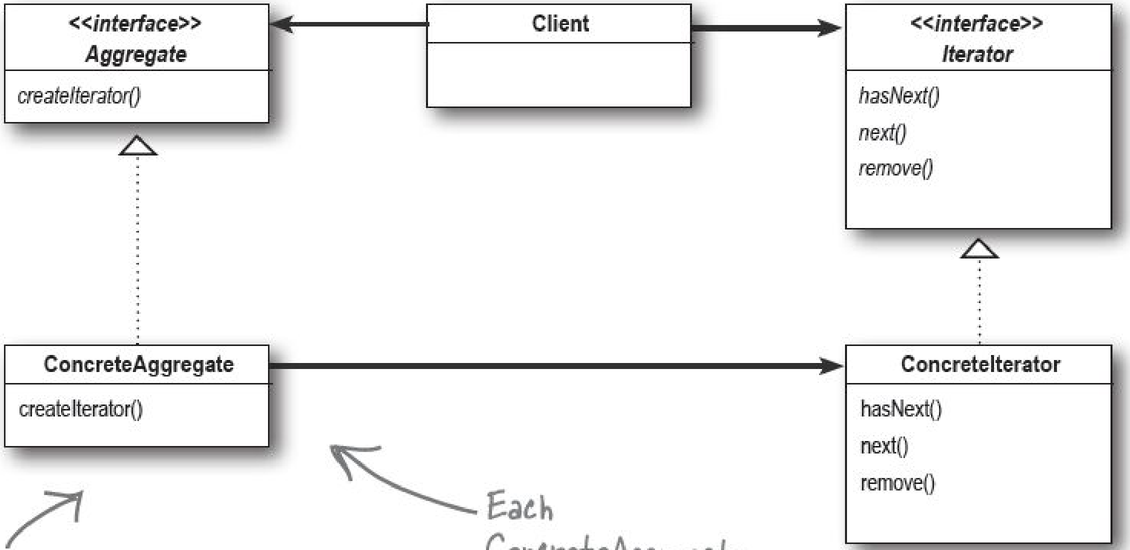
The Iterator Pattern provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

Iterator Pattern defined (cont.)

- 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.
- Let’s check out the class diagram to put all the pieces in context…

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.

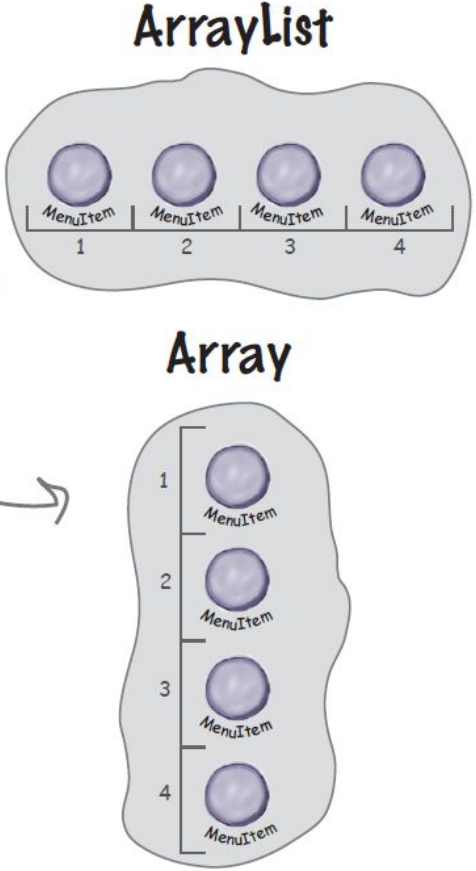
The ConcreteIterator is responsible for managing the current position of the iteration.

What did we do?

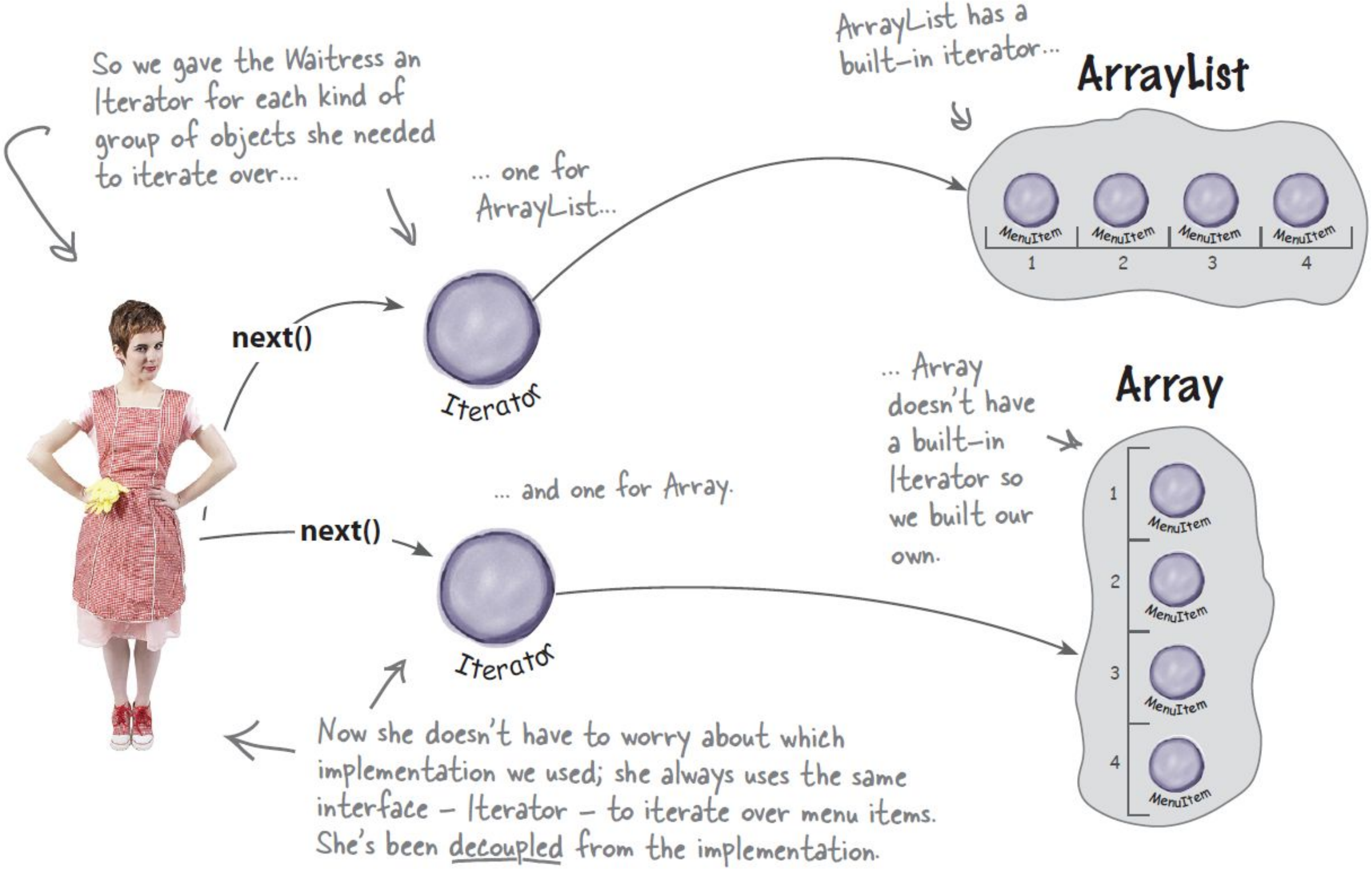
We wanted to give the Waitress an easy way to iterate over menu items...

... 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...

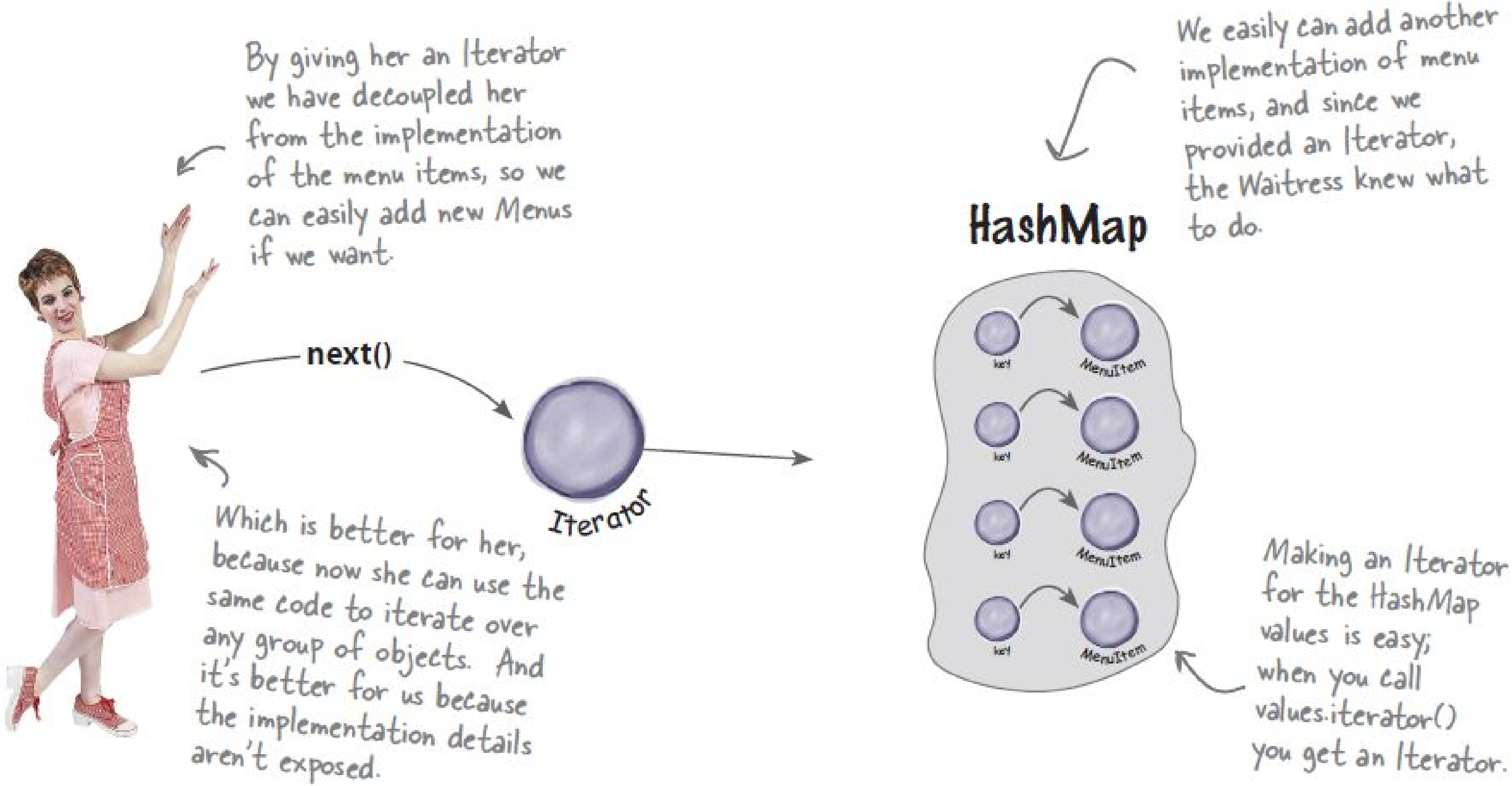
ArrayList has a built-in iterator...

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

... and one for Array.

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 can add another implementation of menu items, and since we provided an Iterator, the Waitress knew what to do.

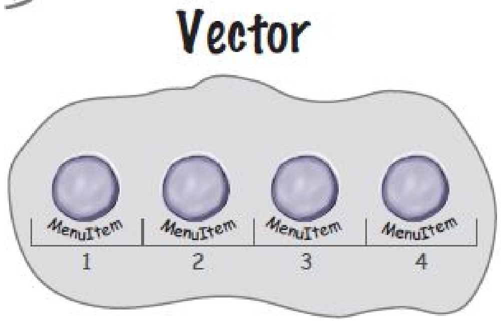
Making an Iterator for the HashMap values is easy; when you call values.iterator() you get an Iterator.

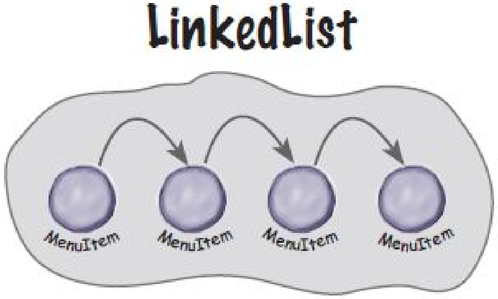
There's more!

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 more!

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

Single Responsibility

- What if we allowed our aggregates to implement their internal collections and related operations AND the iteration methods?
- 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.

Single Responsibility (cont.)

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

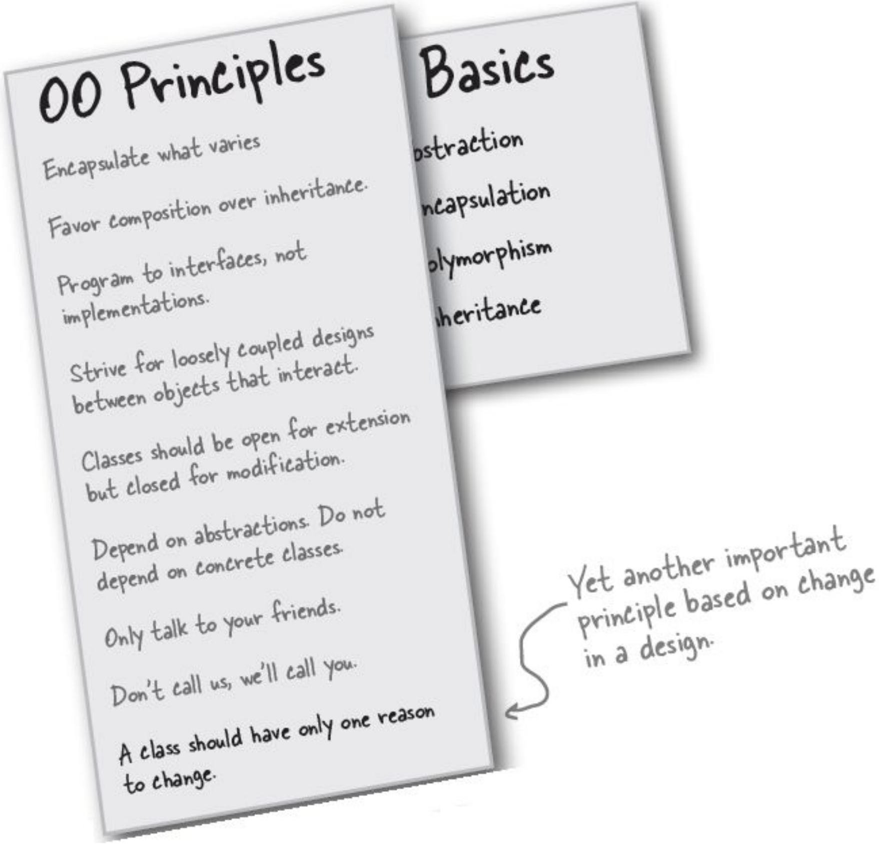
Single Responsibility (cont.)

- **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.

Some Bullet Points

- An Iterator allows access to an aggregate’s elements without exposing its internal structure.
- An Iterator takes the job of iterating over an aggregate and encapsulates it in another object.
- 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.
- We should strive to assign only one responsibility to each class.

Tools for your Design Toolbox



OO Principles

Encapsulate what varies

Favor composition over inheritance.

Program to interfaces, not implementations.

Strive for loosely coupled designs between objects that interact.

Classes should be open for extension but closed for modification.

Depend on abstractions. Do not depend on concrete classes.

Only talk to your friends.

Don't call us, we'll call you.

A class should have only one reason to change.

Basics

bstraction

ncapsulation

olymorphism

heritance

Yet another important principle based on change in a design.

References

Material in this lecture is taken from Freeman, E., Robson, E., Bates, B., & Sierra, K., *Head First Design Patterns: A Brain-Friendly Guide*, O'Reilly Media, Inc., 2014.