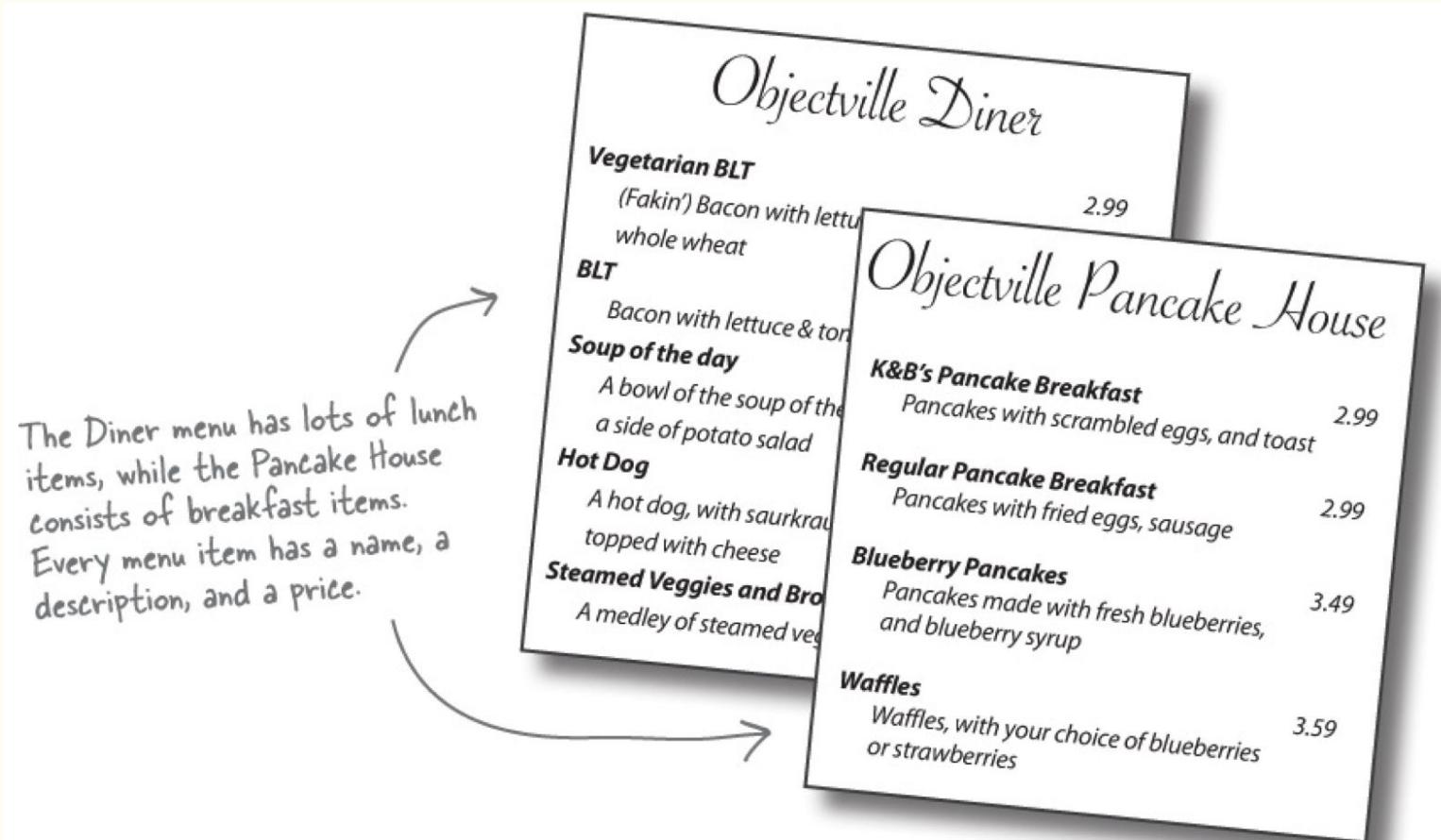


# Iterator and Composite Pattern



# Scenario: Merging of two restaurants



# MenuItem Class

```
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.

# Pancake House Menu - ArrayList

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

```
public class PancakeHouseMenu {  
    ArrayList<MenuItem> menuItems;  
  
    public PancakeHouseMenu() {  
        menuItems = new ArrayList<MenuItem>();  
  
        addItem("K&B's Pancake Breakfast",  
            "Pancakes with scrambled eggs, and toast",  
            true,  
            2.99);  
  
        addItem("Regular Pancake Breakfast",  
            "Pancakes with fried eggs, sausage",  
            false,  
            2.99);  
  
        addItem("Blueberry Pancakes",  
            "Pancakes made with fresh blueberries",  
            true,  
            3.49);  
  
        addItem("Waffles",  
            "Waffles, with your choice of blueberries or strawberries",  
            true,  
            3.59);  
    }  
  
    public void addItem(String name, String description,  
        boolean vegetarian, double price)  
    {  
        MenuItem menuItem = new MenuItem(name, description, vegetarian, price);  
        menuItems.add(menuItem);  
    }  
  
    public ArrayList<MenuItem> getMenuItems() {  
        return menuItems;  
    }  
  
    // other menu methods here  
}
```

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

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

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

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

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

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

# DinerMenu

## - Array

```
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 saukraut, 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.out.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  
}
```

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.

`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.

# Now, Java enabled Waitress

Java-Enabled Waitress: code-name "Alice"

`printMenu()`  
- prints every item on the menu

`printBreakfastMenu()`  
- prints just breakfast items

`printLunchMenu()`  
- prints just lunch items

`printVegetarianMenu()`  
- prints all vegetarian menu items

`isItemVegetarian(name)`  
- given the name of an item, returns true  
if the item is vegetarian, otherwise,  
returns false

# printMenu()

- getMenuItems() from two menus
  - Returns different data type -> ArrayList and Array
- Looping over the returned data -> more than two menus... what??

```
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.

# Merging into one interface

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();

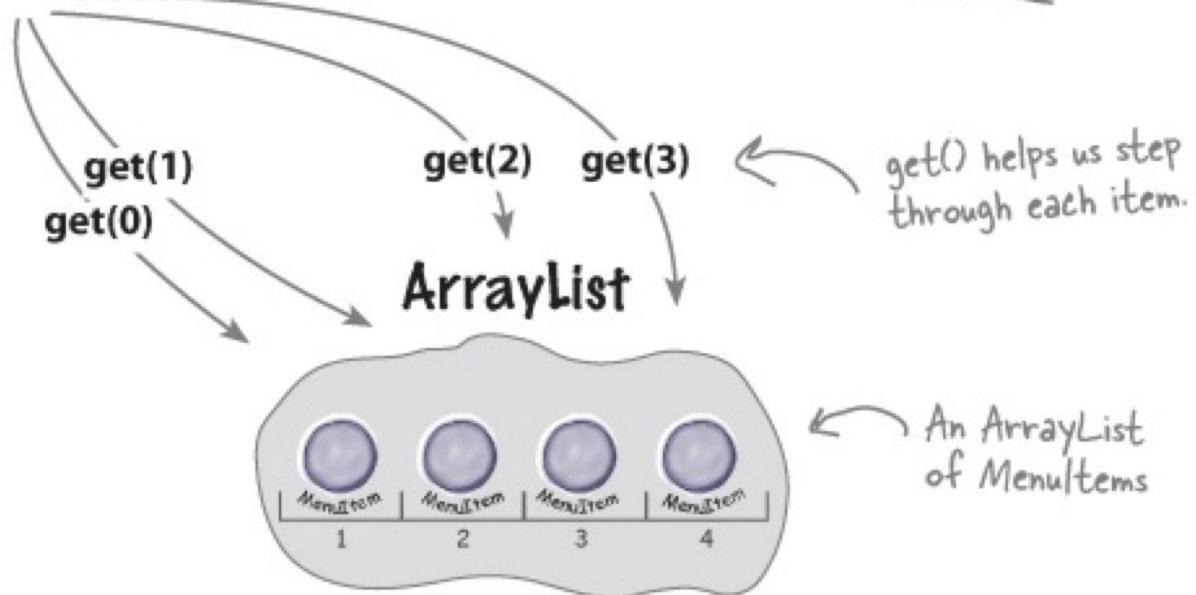
for (MenuItem menuItem : breakfastItems) {
    System.out.print(menuItem.getName());
    System.out.println("\t\t" + menuItem.getPrice());
    System.out.println("\t" + menuItem.getDescription());
}

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

# Iteration of ArrayList

- size(), get() on ArrayList

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

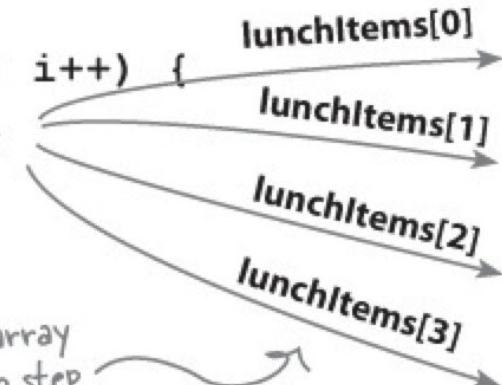


# Iteration of Array

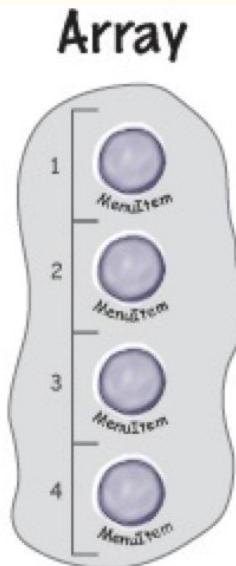
- length field + subscription

```
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.



# Encapsulation of ArrayList using iterator

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

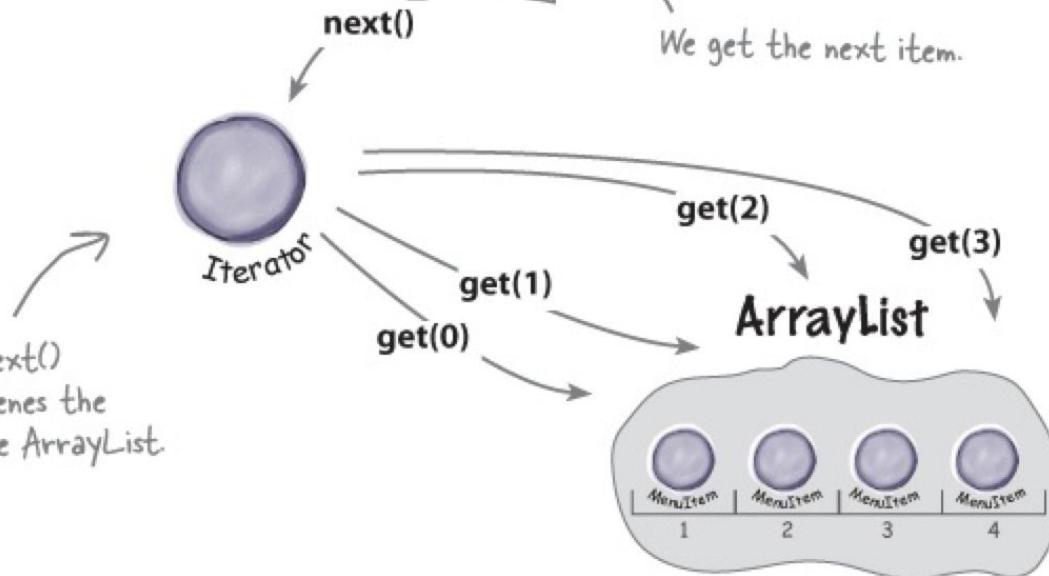
```
while (iterator.hasNext()) {  
    MenuItem menuItem = iterator.next();  
}
```

We ask the breakfastMenu for an iterator of its MenuItem.

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.



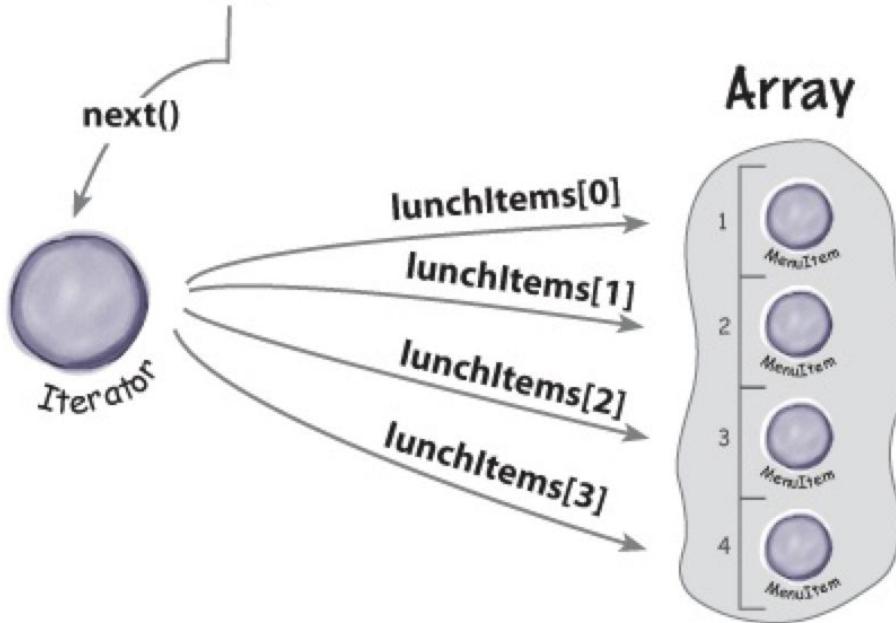
# Encapsulation of Array using iterator

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

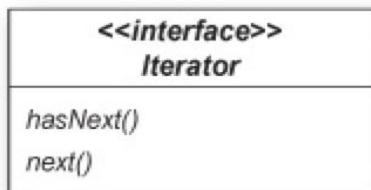
```
while (iterator.hasNext()) {  
    MenuItem menuItem = iterator.next();  
}
```

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.



# Design Pattern of Iterator Pattern



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

The `next()` method returns the next

```
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.

# Adding an Iterator to DinerMenu

```
public class DinerMenuItemIterator implements Iterator {  
    MenuItem[] items;  
    int position = 0;  
  
    public DinerMenuItemIterator(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.

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.

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.

# Modification to DinerMenu

```
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 DinerMenuItemIterator(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 `DinerMenuItemIterator` 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 `DinerMenuItemIterator` is implemented. It just needs to use the iterators to step through the items in the menu.

# New waitress

```
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 dinnerIterator = dinnerMenu.createIterator();  
  
        System.out.println("MENU\n----\nBREAKFAST");  
        printMenu(pancakeIterator);  
        System.out.println("\nLUNCH");  
        printMenu(dinnerIterator);  
    }  
  
    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.

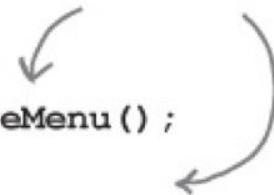
The overloaded printMenu() method uses the Iterator to step through the menu items and print them.

Note that we're down to one loop. Use the item to get name, price, and description and print them.

# Test

```
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.



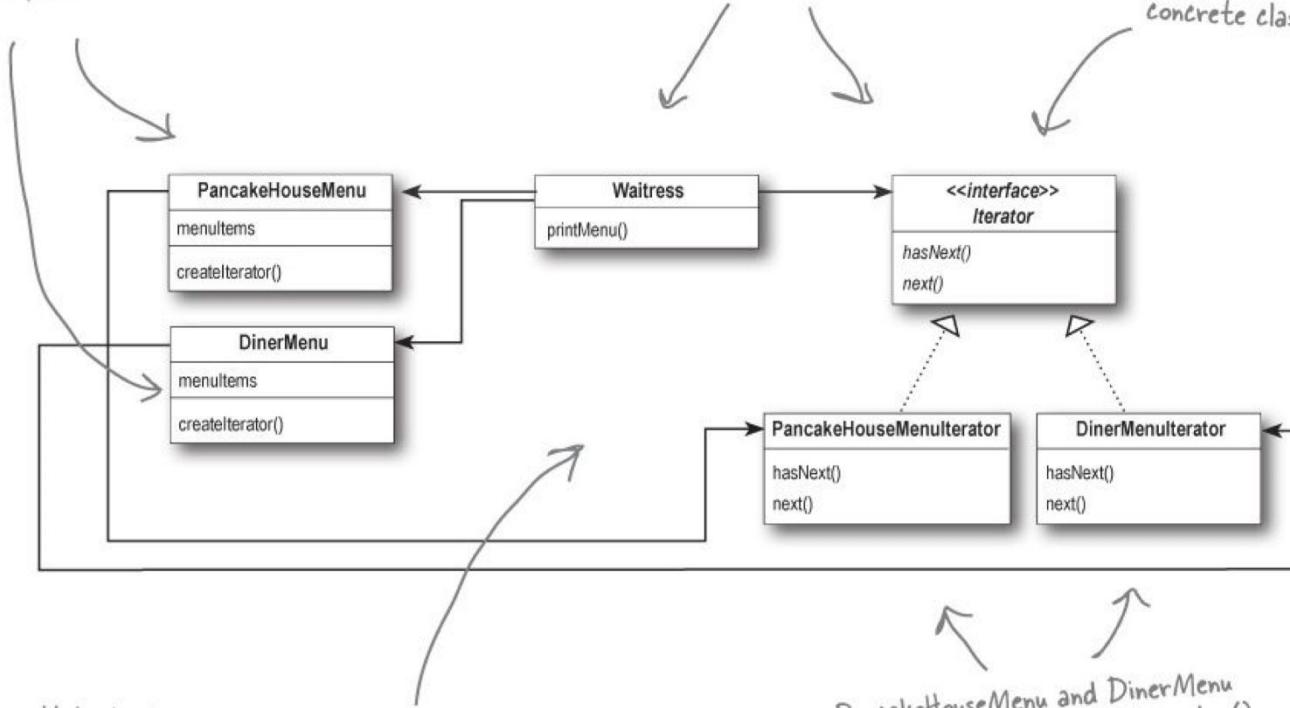
Then we create a  
Waitress and pass  
her the menus.

Then we print them.

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

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

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



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

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

# Java Iterator interface

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

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.

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

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

```
import java.util.Iterator;  
  
public class DinerMenuItemIterator implements Iterator {  
    MenuItem[] list;  
    int position = 0;  
  
    public DinerMenuItemIterator(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.

# New waitress

```
import java.util.Iterator;  
  
public class DinerMenuItemIterator implements Iterator {  
    MenuItem[] list;  
    int position = 0;  
  
    public DinerMenuItemIterator(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.

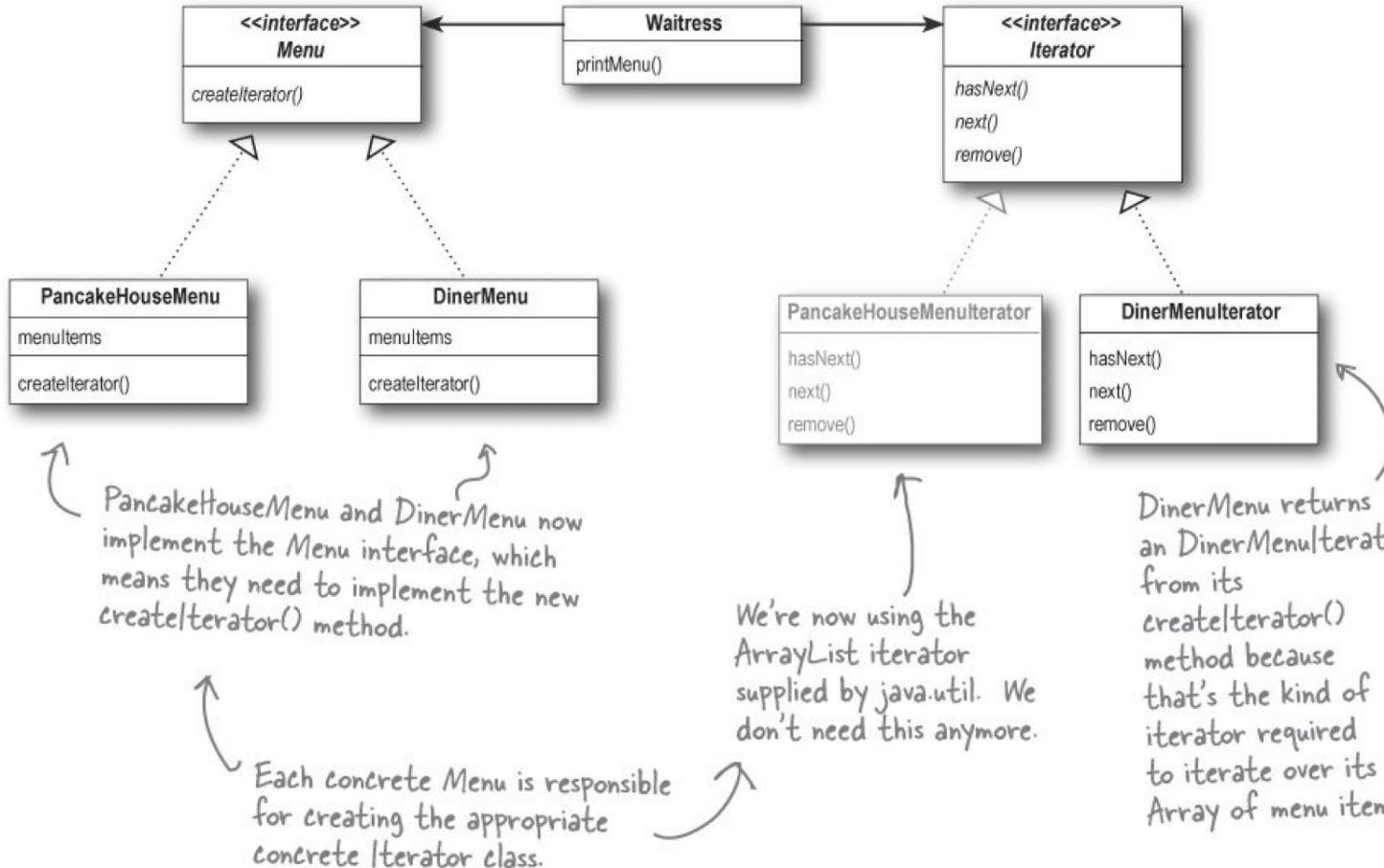
None of our current implementation changes...

...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.

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

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

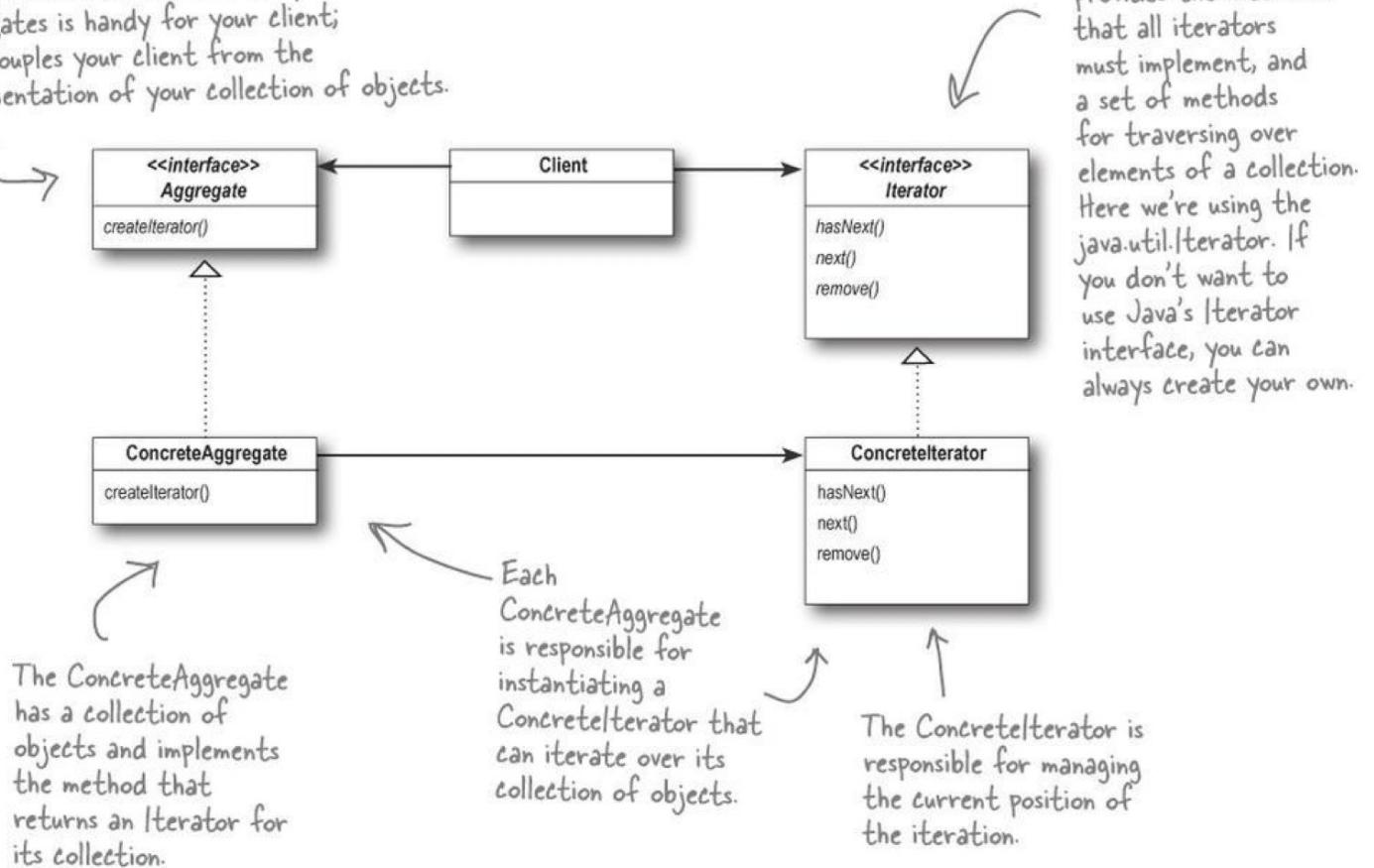
We've decoupled Waitress from the  
implementation of the menus, so now  
we can use an Iterator to iterate  
over any list of menu items without  
having to know about how the list of  
items is implemented.



# Definition of Iterator Design Pattern

- Provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation

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



# Café Menu Scenario

- interface
- HashMap support Iterator in different way

```
public class CafeMenu {  
    HashMap<String, MenuItem> menuItems = new HashMap<String, MenuItem>();  
  
    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 Map<String, MenuItem> getItems() {  
        return menuItems;  
    }  
}
```

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

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

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

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.

# Correction to CafeMenu

```
public class CafeMenu implements Menu {
    HashMap<String, MenuItem> menuItems = new HashMap<String, MenuItem>();

    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 Map<String, MenuItem> getItems()
    {
        return menuItems;
    }

    public Iterator<MenuItem> 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 HashMap because it's a common data structure for storing value.

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 HashMap, just for the values.

# New Waitress for Cafeteria

```
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<MenuItem> pancakeIterator = pancakeHouseMenu.createIterator();  
        Iterator<MenuItem> dinerIterator = dinerMenu.createIterator();  
        Iterator<MenuItem> 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 = 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.

# Test for CafeMenu

```
public class MenuTestDrive {  
    public static void main(String args[]) {  
        PancakeHouseMenu pancakeHouseMenu = new PancakeHouseMenu(); Create a CafeMenu...  
        DinerMenu dinerMenu = new DinerMenu();  
        CafeMenu cafeMenu = new CafeMenu();  
  
        Waitress waitress = new Waitress(pancakeHouseMenu, dinerMenu, cafeMenu); ... and pass it to the waitress.  
  
        waitress.printMenu(); ← Now, when we print we should see all three menus.  
    }  
}
```

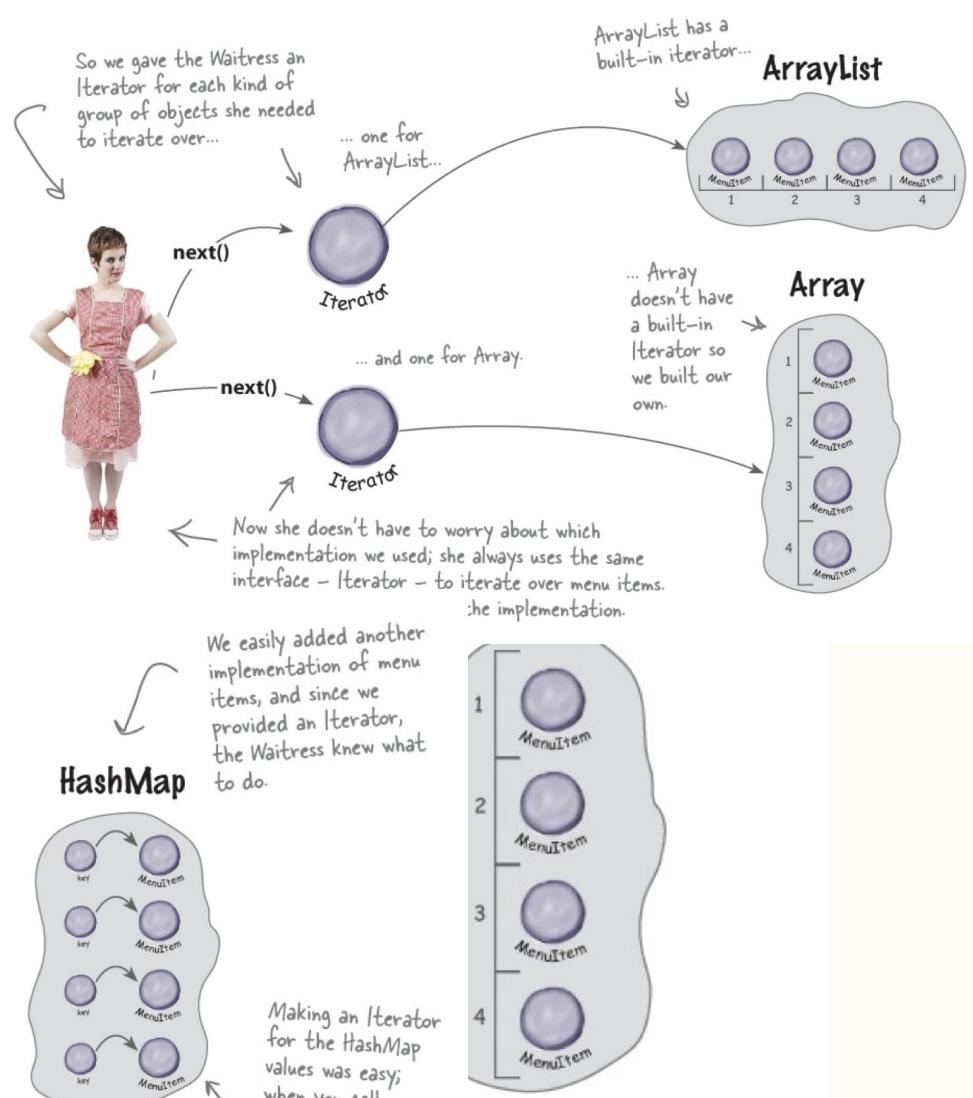
# Decoupling of the waitress



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.

next()  
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.

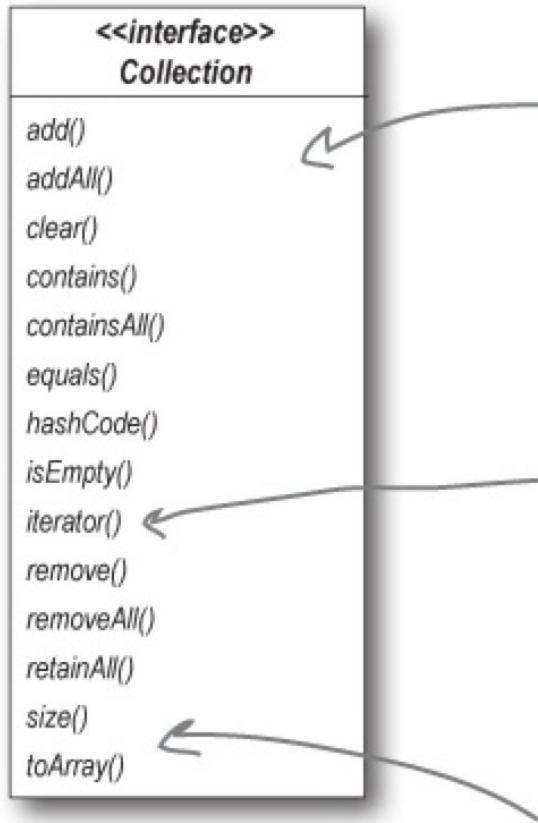
Our diff in



# Iterators and Collections

- Java collections Framework :-  
PriorityQueue
  - Implementation of java.util.Collection

Hashtable ={(key, value)}  
- HoshMap = keys + values  
- Iteration over values



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.

# Consideration over the waitress

New Addition of a new menu  
-> modification of the  
printMenu()

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



Three createIterator() calls.

Three calls to  
printMenu.

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

# One call for every menu??

```
public class Waitress {  
    ArrayList<Menu> menus;  
  
    public Waitress(ArrayList<Menu> menus) {  
        this.menus = menus;  
    }  
  
    public void printMenu() {
```

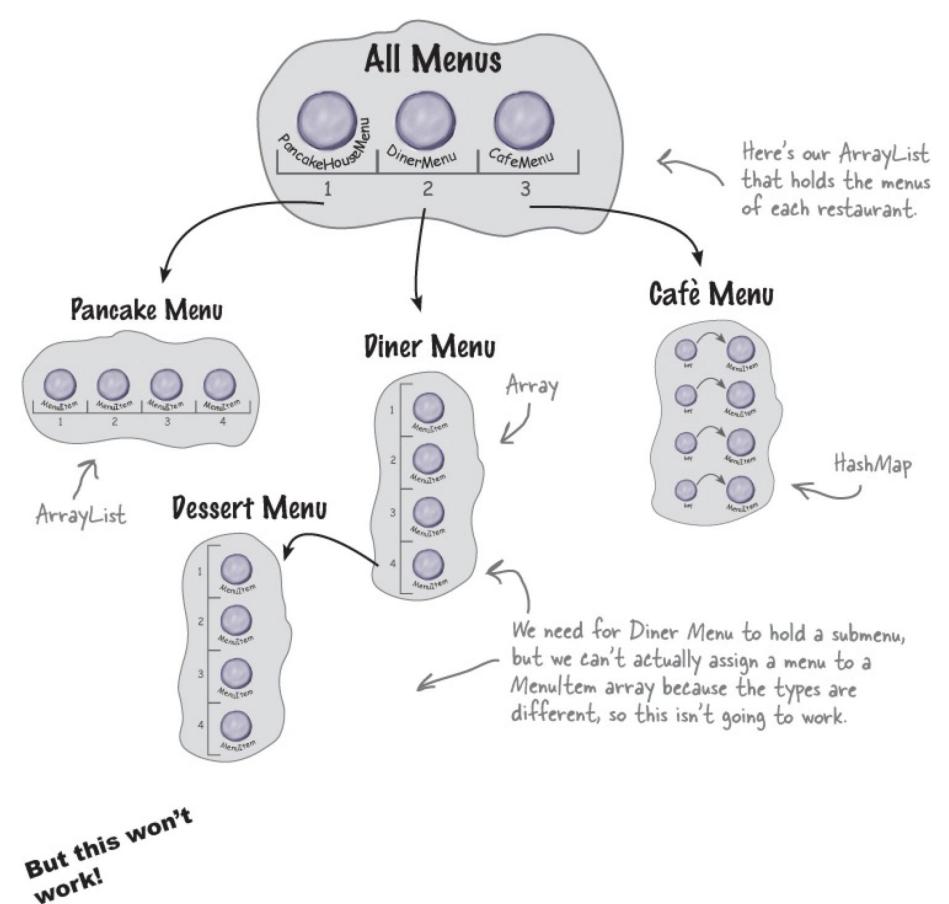
Now we just take an  
ArrayList of menus.

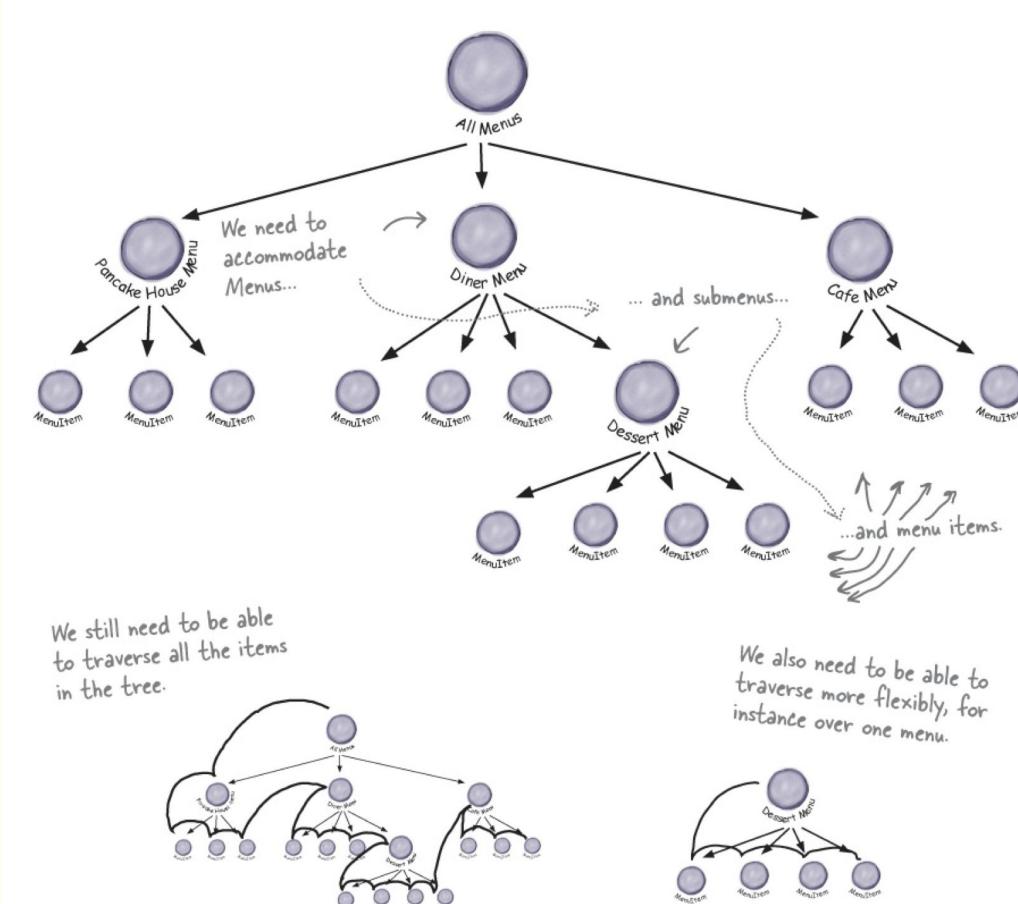
```
        Iterator<Menu> menuIterator = menus.iterator();  
        while(menuIterator.hasNext()) {  
            Menu menu = menuIterator.next();  
            printMenu(menu.createIterator());  
        }  
    }
```

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

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

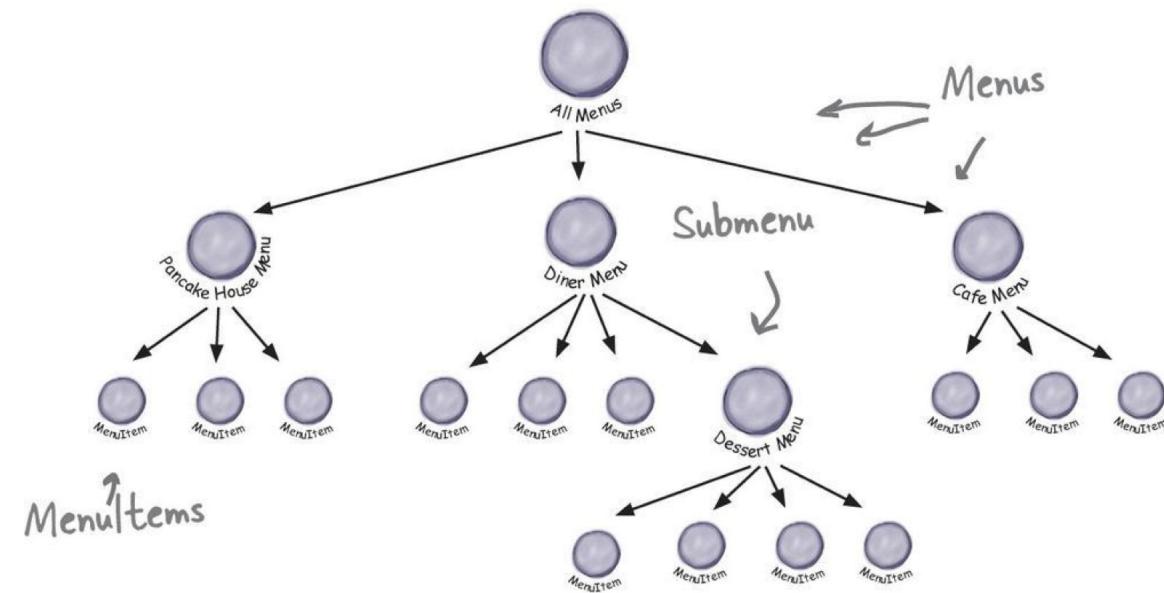
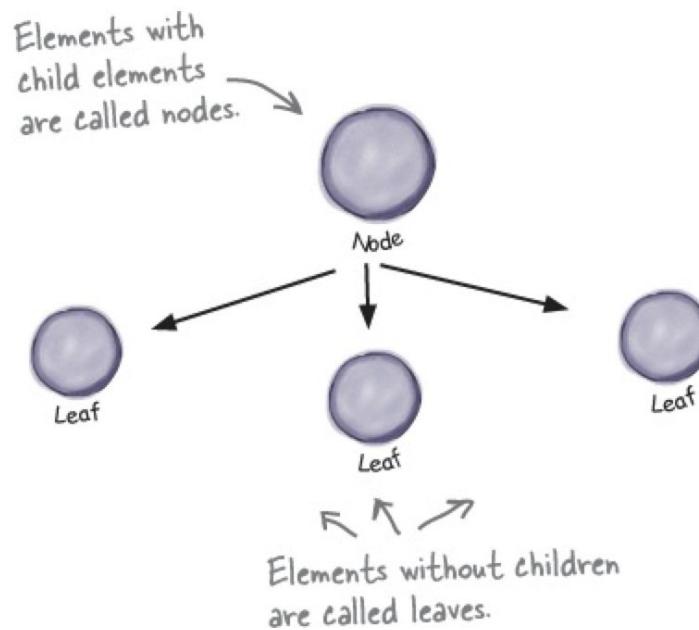
No code  
changes here.





# Definition of Composite Pattern

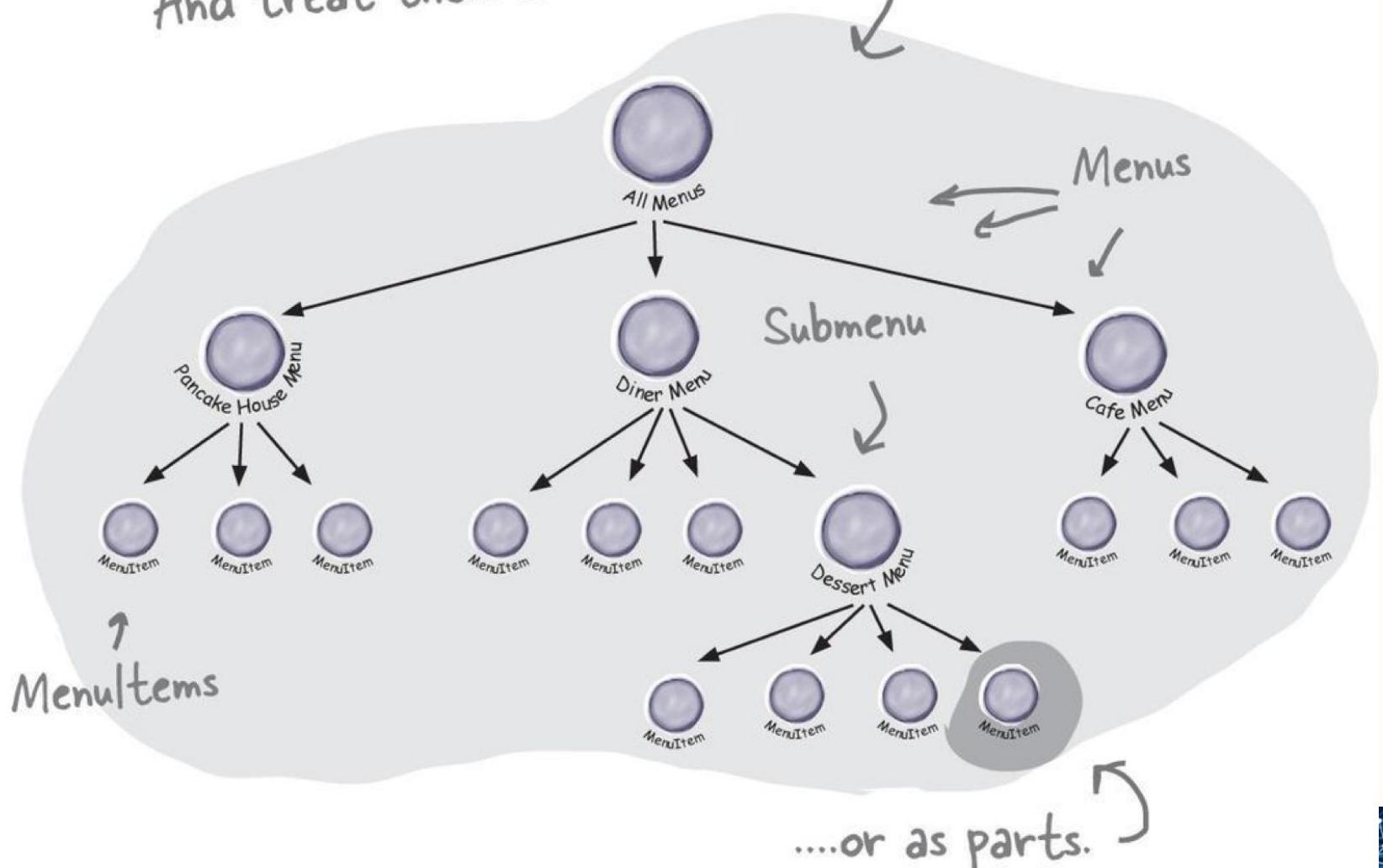
- Allows you to compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly



Homogeneous or Heterogeneous Tree??

# Part-Whole Structure

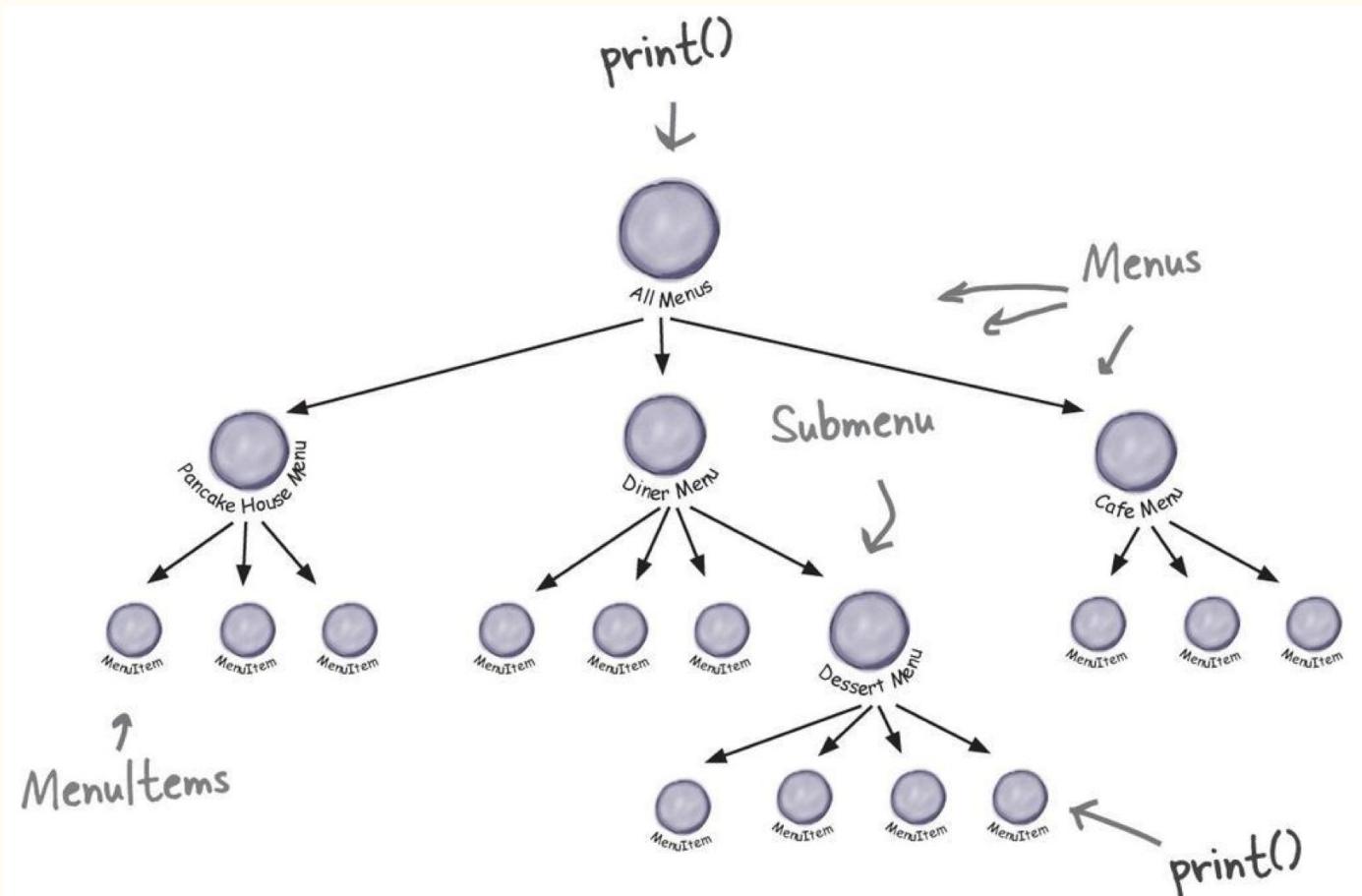
And treat them as a whole...



Within a structure,  
different kinds of nodes ->  
Subtree -> as part



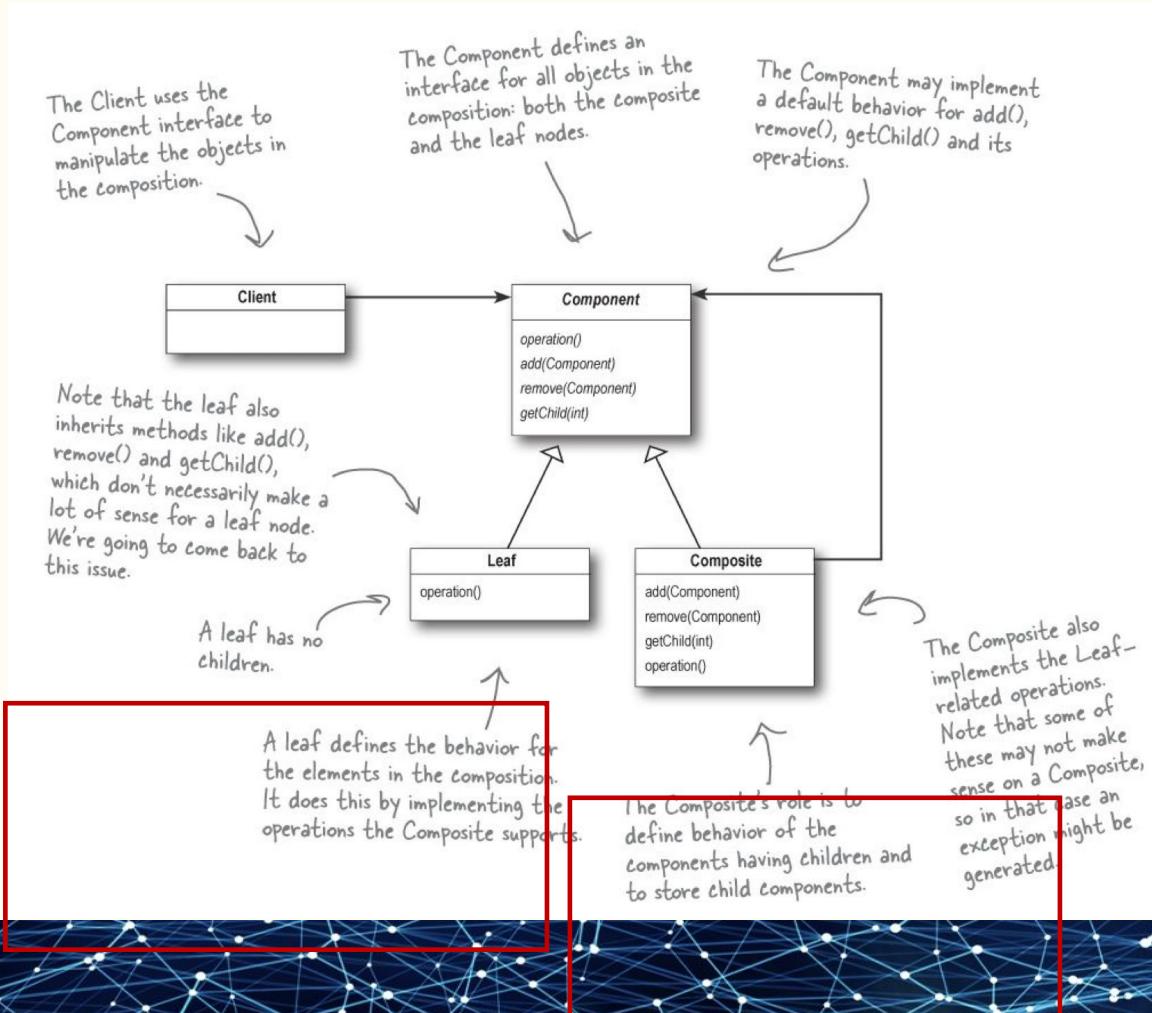
Within a collection,  
another level of collections  
as item



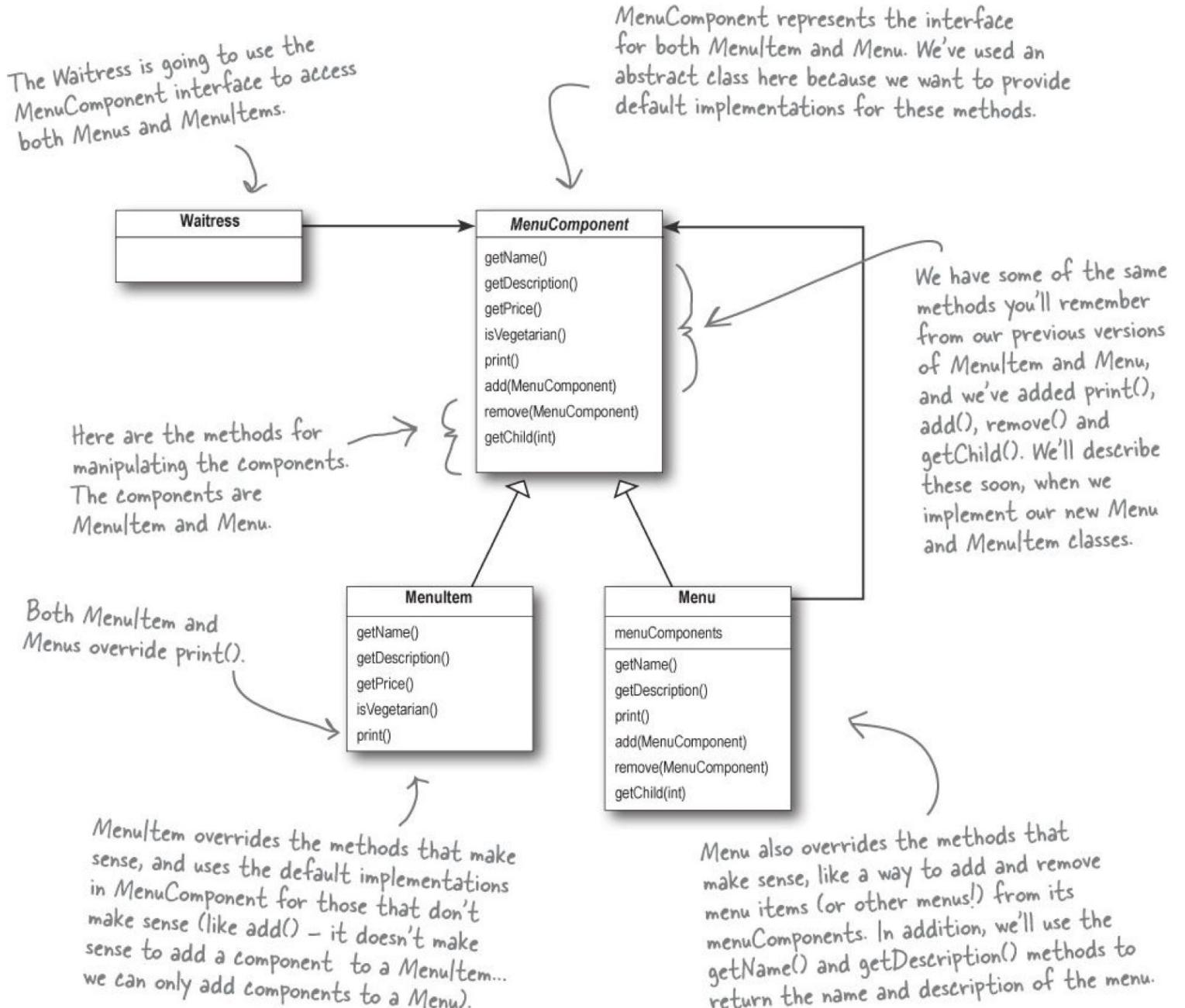
# Definition of Composition Pattern

The Composite Pattern allows us to build structures of objects in the form of trees that contain both compositions of objects and individual objects as nodes.

Using a composite structure, we can apply the same operations over both composites and individual objects. In other words, in most cases we can ignore the differences between compositions of objects and individual objects.



# Application to



MenuComponent provides default implementations for every method.

```
public abstract class MenuComponent {  
  
    public void add(MenuComponent menuComponent) {  
        throw new UnsupportedOperationException();  
    }  
    public void remove(MenuComponent menuComponent) {  
        throw new UnsupportedOperationException();  
    }  
    public MenuComponent getChild(int i) {  
        throw new UnsupportedOperationException();  
    }  
  
    public String getName() {  
        throw new UnsupportedOperationException();  
    }  
    public String getDescription() {  
        throw new UnsupportedOperationException();  
    }  
    public double getPrice() {  
        throw new UnsupportedOperationException();  
    }  
    public boolean isVegetarian() {  
        throw new UnsupportedOperationException();  
    }  
  
    public void print() {  
        throw new UnsupportedOperationException();  
    }  
}
```

We've grouped together the "Composite" methods – that is, methods to add, remove and get MenuComponents.

Here are the "operation" methods; these are used by the MenuItem's. It turns out we can also use a couple of them in Menu too, as you'll see in a couple of pages when we show the Menu code.

print() is an "operation" method that both our Menus and MenuItem's will implement, but we provide a default operation here.

```
public class MenuItem extends MenuComponent {  
    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;  
    }  
  
    public void print()  
    {  
        System.out.print(" " + getName());  
        if (isVegetarian()) {  
            System.out.print("(v)");  
        }  
        System.out.println(", " + getPrice());  
        System.out.println(" -- " + getDescription());  
    }  
}
```

First we need to extend  
the MenuComponent  
interface.

The constructor just takes the  
name, description, etc. and  
keeps a reference to them all.  
This is pretty much like our  
old menu item implementation.

Here's our getter methods  
- just like our previous  
implementation.

This is different from the previous implementation.  
Here we're overriding the print() method in the  
MenuComponent class. For MenuItem this method  
prints the complete menu entry: name, description,  
price and whether or not it's veggie.

```

Menu is also a MenuComponent,
just like MenuItem. ↴
public class Menu extends MenuComponent {
    ArrayList<MenuComponent> menuComponents = new ArrayList<MenuComponent>();
    String name;
    String description;

    public Menu(String name, String description) {
        this.name = name;
        this.description = description;
    }

    public void add(MenuComponent menuComponent) {
        menuComponents.add(menuComponent);
    }

    public void remove(MenuComponent menuComponent) {
        menuComponents.remove(menuComponent);
    }

    public MenuComponent getChild(int i) {
        return menuComponents.get(i);
    }

    public String getName() {
        return name;
    }

    public String getDescription() {
        return description;
    }

    public void print() {
        System.out.print("\n" + getName());
        System.out.println(", " + getDescription());
        System.out.println("-----");
    }
}

Menu can have any number of children
of type MenuComponent. We'll use an
internal ArrayList to hold these. ↴
This is different than our old
implementation: we're going to give each
Menu a name and a description. Before,
we just relied on having different classes
for each menu. ↴
Here's how you add MenuItem's or
other Menus to a Menu. Because
both MenuItem's and Menus are
MenuComponents, we just need one
method to do both. ↴
You can also remove a MenuComponent
or get a MenuComponent. ↴
Here are the getter methods for getting the name
and description. ↴
Notice, we aren't overriding getPrice() or
isVegetarian() because those methods don't make
sense for a Menu (although you could argue that
isVegetarian() might make sense). If someone tries
to call those methods on a Menu, they'll get an
UnsupportedOperationException. ↴
To print the Menu, we print the
Menu's name and description. ↴

```

# Waitress

```
public class Waitress {  
    MenuComponent allMenus;  
  
    public Waitress(MenuComponent allMenus) {  
        this.allMenus = allMenus;  
    }  
  
    public void printMenu() {  
        allMenus.print();  
    }  
}
```



Yup! The Waitress code really is this simple. Now we just hand her the top-level menu component, the one that contains all the other menus. We've called that allMenus.



All she has to do to print the entire menu hierarchy – all the menus, and all the menu items – is call print() on the top level menu.

We're gonna have one happy Waitress.

# Test

```
public class MenuTestDrive {
    public static void main(String args[]) {
        MenuComponent pancakeHouseMenu =
            new Menu("PANCAKE HOUSE MENU", "Breakfast");
        MenuComponent dinerMenu =
            new Menu("DINER MENU", "Lunch");
        MenuComponent cafeMenu =
            new Menu("CAFE MENU", "Dinner");
        MenuComponent dessertMenu =
            new Menu("DESSERT MENU", "Dessert of course!");

        MenuComponent allMenus = new Menu("ALL MENUS", "All menus combined");

        allMenus.add(pancakeHouseMenu); ← Let's first create
        allMenus.add(dinerMenu); ← all the menu objects.
        allMenus.add(cafeMenu);

        // add menu items here ← We also need a top-
        dinerMenu.add(new MenuItem( ← level menu that we'll
            "Pasta",
            "Spaghetti with Marinara Sauce, and a slice of sourdough bread",
            true,
            3.89)); ← name allMenus.

        dinerMenu.add(dessertMenu); ← Now we need to add all
        dessertMenu.add(new MenuItem( ← the menu items. Here's one
            "Apple Pie",
            "Apple pie with a flaky crust, topped with vanilla icecream",
            true,
            1.59)); ← example; for the rest, look at
            ← the complete source code.

        // add more menu items here ← And we're also adding a menu to a
        waitress waitress = new Waitress(allMenus); ← menu. All dinerMenu cares about is that
        waitress.printMenu(); ← everything it holds; whether it's a menu
        } ← item or a menu, is a MenuComponent.

    }
}
```

← Add some apple pie to the dessert menu...

Once we've constructed our entire menu hierarchy, we hand the whole thing to the Waitress, and as you've seen, it's as easy as apple pie for her to print it out.

# Iterator with a Composite



We've added a *createIterator()* method to the *MenuComponent*. This means that each *Menu* and *MenuItem* will need to implement this method. It also means that calling *createIterator()* on a composite should apply to all children of the composite.

```
public class Menu extends MenuComponent {  
    Iterator<MenuComponent> iterator = null;  
    // other code here doesn't change  
  
    public Iterator<MenuComponent> createIterator() {  
        if (iterator == null) {  
            iterator = new CompositeIterator(menuComponents.iterator());  
        }  
        return iterator;  
    }  
}  
  
public class MenuItem extends MenuComponent {  
    // other code here doesn't change  
  
    public Iterator<MenuComponent> createIterator() {  
        return new NullIterator();  
    }  
}
```

Here we're using a new iterator called CompositeIterator. It knows how to iterate over any composite. We pass it the current composite's iterator.

Now for the MenuItem...

Whoa! What's this NullIterator?  
You'll see in two pages.

# Need to know tree traversal

```
import java.util.*;  
  
public class CompositeIterator implements Iterator {  
    Stack<Iterator<MenuComponent>> stack = new Stack<Iterator<MenuComponent>>();  
  
    public CompositeIterator(Iterator iterator) {  
        stack.push(iterator);  
    }  
  
    public Object next() {  
        if (hasNext()) {  
            Iterator<MenuComponent> iterator = stack.peek();  
            MenuComponent component = iterator.next();  
  
            stack.push(component.createIterator());  
  
            return component;  
        } else {  
            return null;  
        }  
    }  
  
    public boolean hasNext() {  
        if (stack.empty()) {  
            return false;  
        } else {  
            Iterator<MenuComponent> iterator = stack.peek();  
            if (!iterator.hasNext()) {  
                stack.pop();  
                return hasNext();  
            } else {  
                return true;  
            }  
        }  
    }  
}
```

Like all iterators, we're implementing the `java.util.Iterator` interface.

The iterator of the top-level composite we're going to iterate over is passed in. We throw that in a stack data structure.

Okay, when the client wants to get the next element we first make sure there is one by calling `hasNext()`...

If there is a next element, we get the current iterator off the stack and get its next element.

We then throw that component's iterator on the stack. If the component is a `Menu`, it will iterate over all its items. If the component is a `MenuItem`, we get the `NullIterator`, and no iteration happens. Then we return the component.

To see if there is a next element, we check to see if the stack is empty; if so, there isn't.

Otherwise, we get the iterator off the top of the stack and see if it has a next element. If it doesn't we pop it off the stack and call `hasNext()` recursively.

Otherwise there is a next element and we return true.

We're not supporting remove, so we don't implement it and leave it up to the default behavior in `java.util.Iterator`.

# Null iterator

```
import java.util.Iterator;

public class NullIterator implements <MenuComponent> {

    public Object next() {
        return null;
    }

    public boolean hasNext() {
        return false;
    }

    public void remove() {
        throw new UnsupportedOperationException();
    }
}
```

This is the laziest Iterator you've ever seen. At every step of the way it punts.

When next() is called, we return null.

Most importantly when hasNext() is called we always return false.

And the NullIterator wouldn't think of supporting remove. We don't need to implement this; we could leave it off and let the default java.util.Iterator remove handle it.

```

public class Waitress {
    MenuComponent allMenus;

    public Waitress(MenuComponent allMenus) {
        this.allMenus = allMenus;
    }

    public void printMenu() {
        allMenus.print();
    }

    public void printVegetarianMenu() {
        Iterator<MenuComponent> iterator = allMenus.createIterator();

        System.out.println("\nVEGETARIAN MENU\n----");
        while (iterator.hasNext()) {
            MenuComponent menuComponent = iterator.next();
            try {
                if (menuComponent.isVegetarian()) {
                    menuComponent.print();
                }
            } catch (UnsupportedOperationException e) {}
        }
    }
}

The printVegetarianMenu() method takes the allMenus's composite and gets its iterator. That will be our CompositIterator.

Iterate through every element of the composite.

Call each element's isVegetarian() method and if true, we call its print() method.

print() is only called on MenuItem, never composites. Can you see why?

We implemented isVegetarian() on the Menus to always throw an exception. If that happens we catch the exception, but continue with our iteration.

```