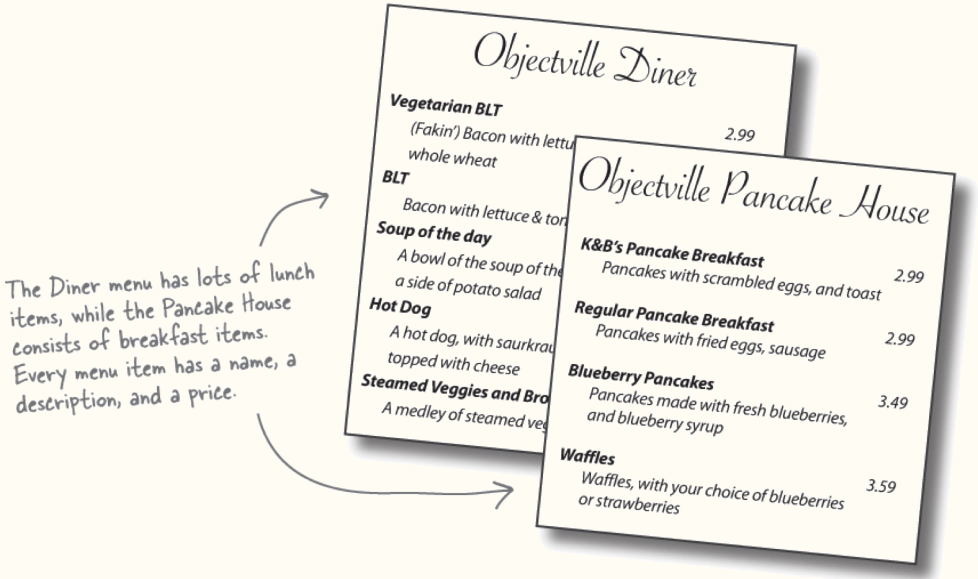
Iterator and Composite Pattern

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

DinerMenu - 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.

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.

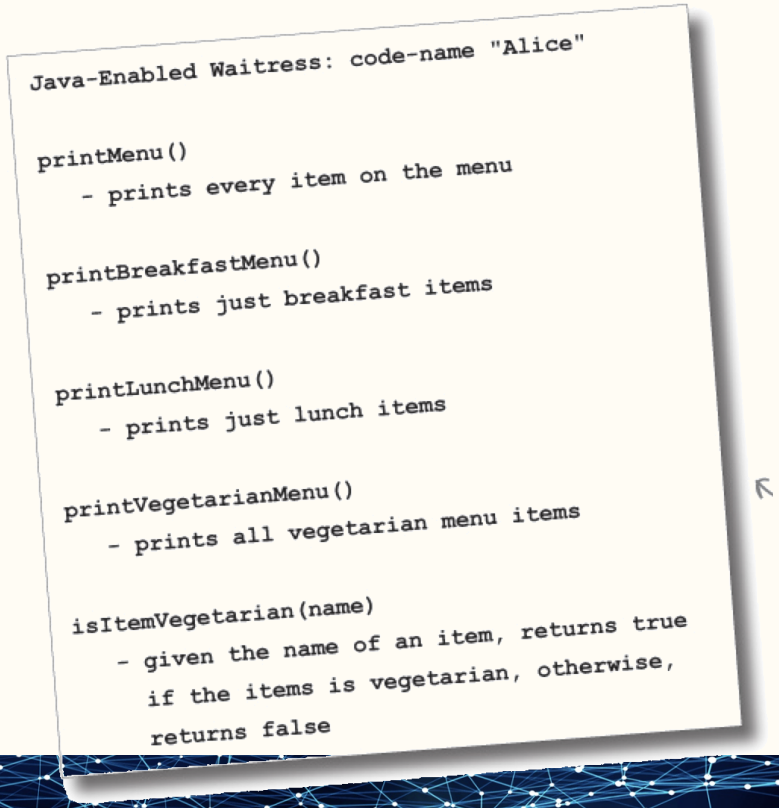
Mel specifically wants to keep his menu under a certain size (presumably so he doesn't have to remember too many recipes).

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.

```
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
```

Now, Java enabled Waitress



printMenu()

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

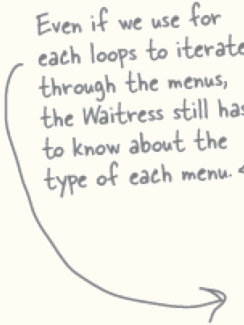
```
for (int i = 0; i < breakfastItems.size(); i++) {
    MenuItem menuItem = breakfastItems.get(i);
    System.out.print(menuItem.getName() + " ");
    System.out.println(menuItem.getPrice() + " ");
    System.out.println(menuItem.getDescription());
}
for (int i = 0; i < lunchItems.length; i++) {
 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

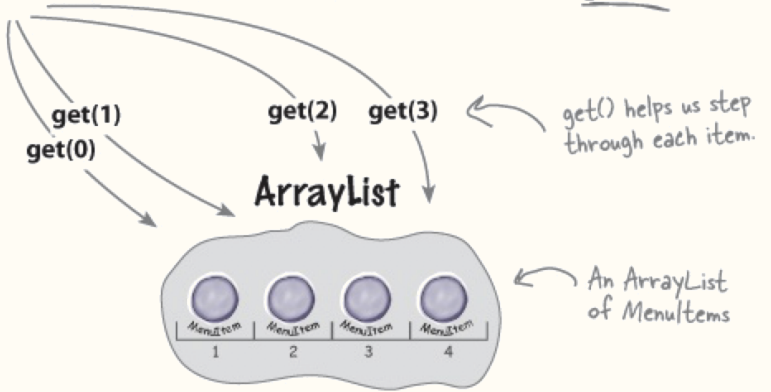



```
PancakeHouseMenu pancakeHouseMenu = new PancakeHouseMenu();
ArrayList<MenuItem> breakfastItems = pancakeHouseMenu.getMenuItems();

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

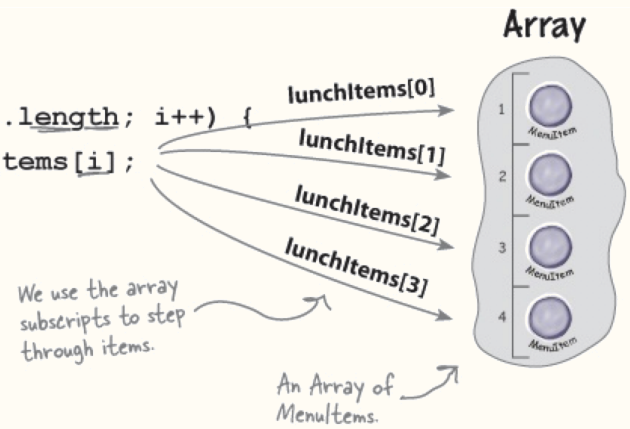
```
for (int i = 0; i < breakfastItems.size(); i++) {
    MenuItem menuItem = breakfastItems.get(i);
}
```



Iteration of Array

- length field + subscription

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



Encapsulation of ArrayList using iterator

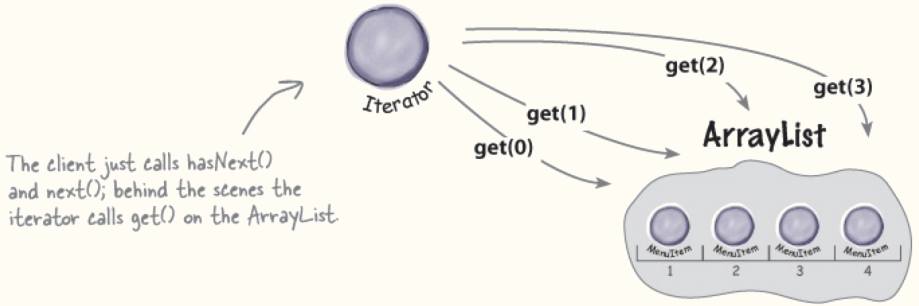
We ask the breakfastMenu for an iterator of its MenuItems.

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

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

And while there are more items left...

next()

We get the next item.



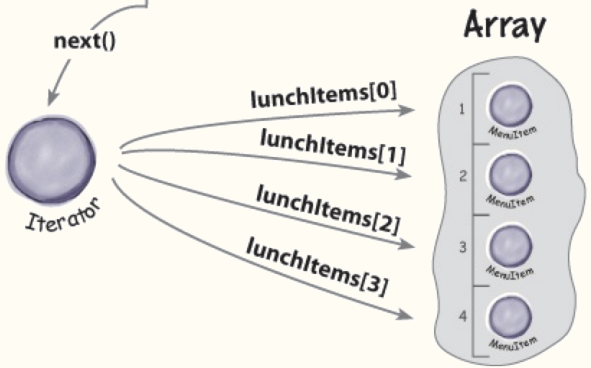
The client just calls hasNext() and next(); behind the scenes the iterator calls get() on the ArrayList.

Encapsulation of Array using iterator

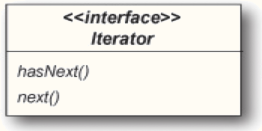
```
Iterator iterator = lunchMenu.createIterator();

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 object in the aggregate.

Here are our two methods:

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

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

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

Adding an Iterator to DinerMenu

```
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.

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 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.

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 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.

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.

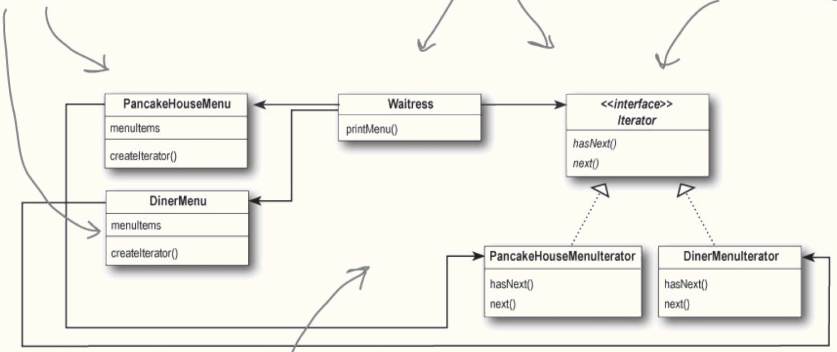
Then we create a Waitress and pass her the menus.

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.

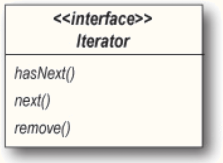
We're now using a common Iterator interface and we've implemented two concrete classes.



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 interation.

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

Java Iterator interface



This looks just like our previous definition.

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

```
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 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.

New waitress

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

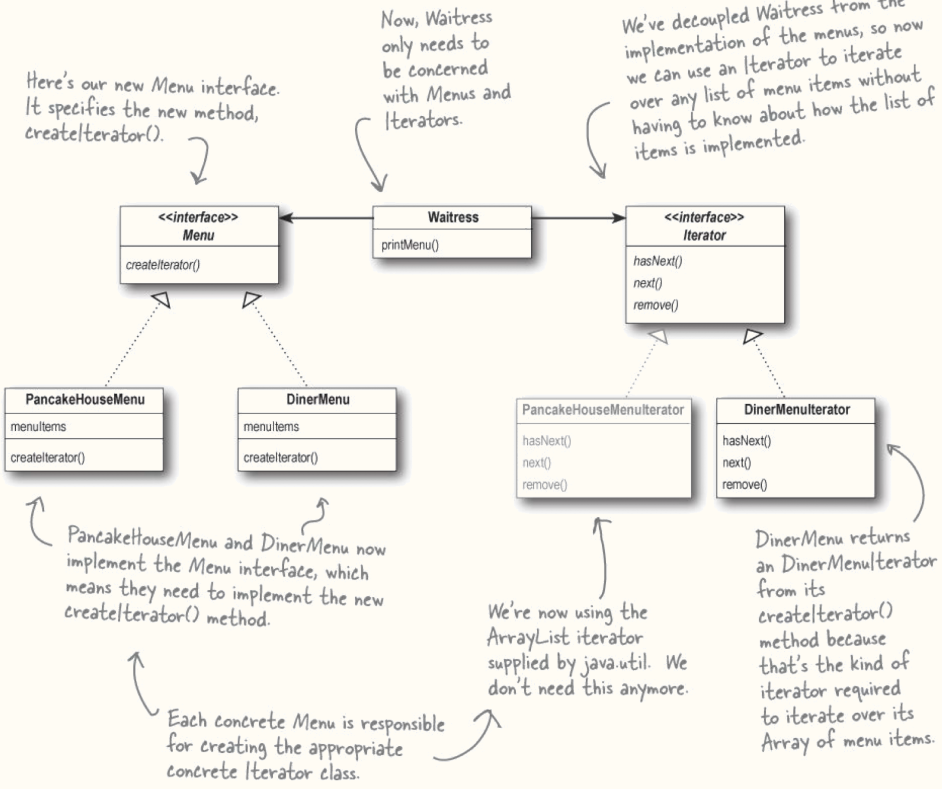
First we import java.util.Iterator, the interface we're going to implement.

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

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

    public MenuItem next() {
        //implementation here
    }

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

None of our current implementation changes...

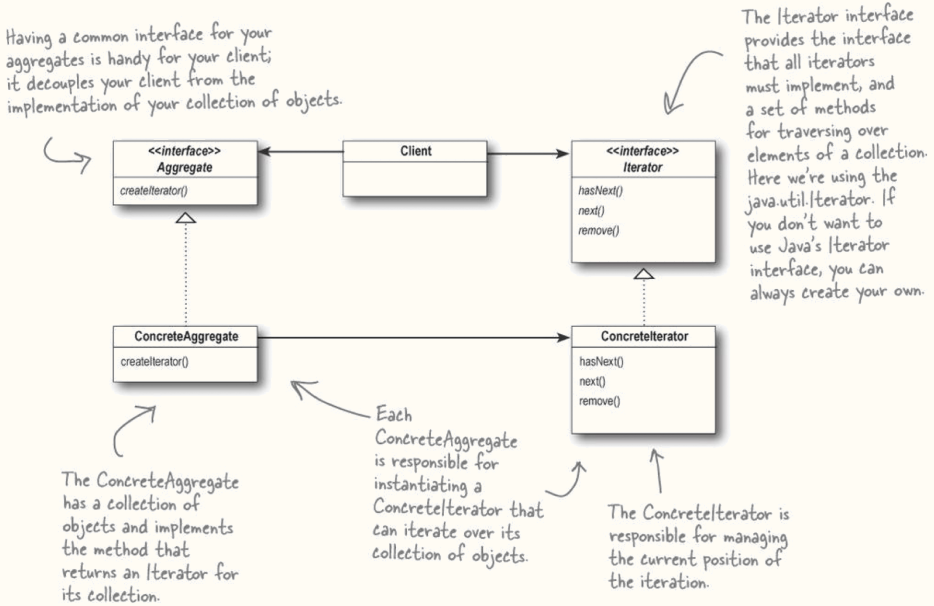
...but we do need to implement remove(). Here, because the chef is using a fixed-size Array, we just shift all the elements up one when remove() is called.

```
    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;
        }
    }
}
```


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()
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.

Definition of Iterator Design Pattern

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



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 cafe 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.

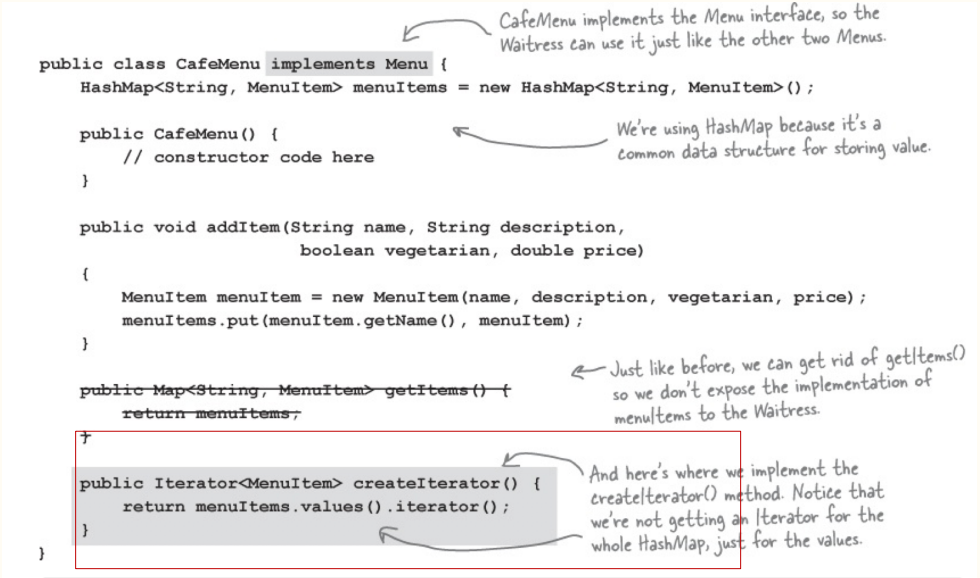
Here's where we create a new MenuItem and add it to the menuItems hashtable.

The key is the item name.

The value is the menuItem object.

We're not going to need this anymore.

Correction to CafeMenu



New Waitress for C

```
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

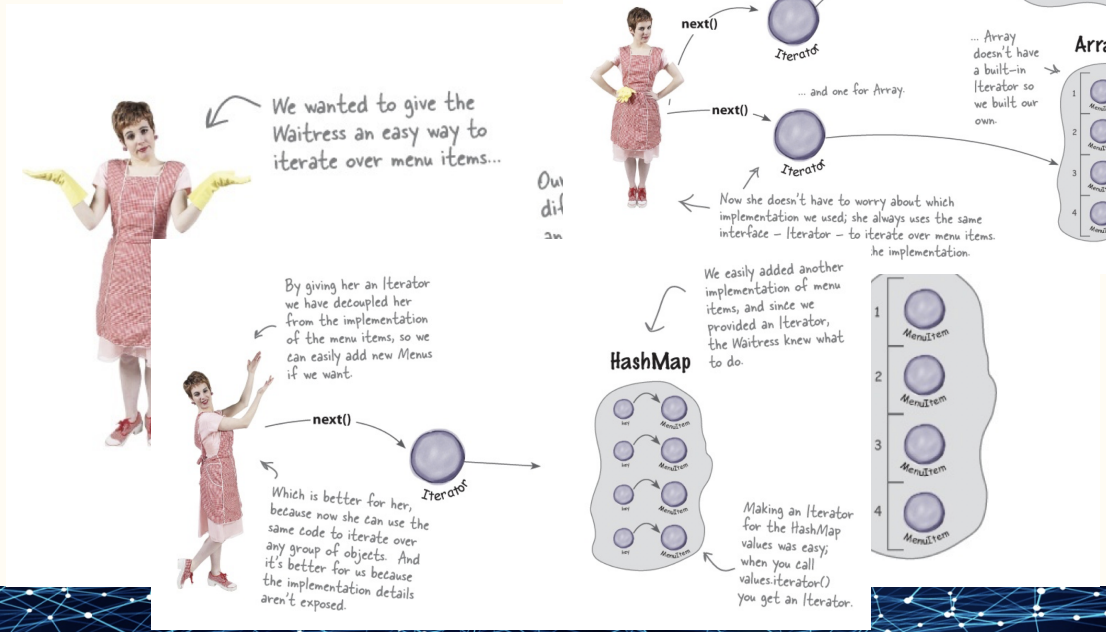
```
public class MenuTestDrive {
    public static void main(String args[]) {
        PancakeHouseMenu pancakeHouseMenu = new PancakeHouseMenu();
        DinerMenu dinerMenu = new DinerMenu();
        CafeMenu cafeMenu = new CafeMenu();

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

        waitress.printMenu();
    }
}
```

Create a CafeMenu...

... and pass it to the waitress.

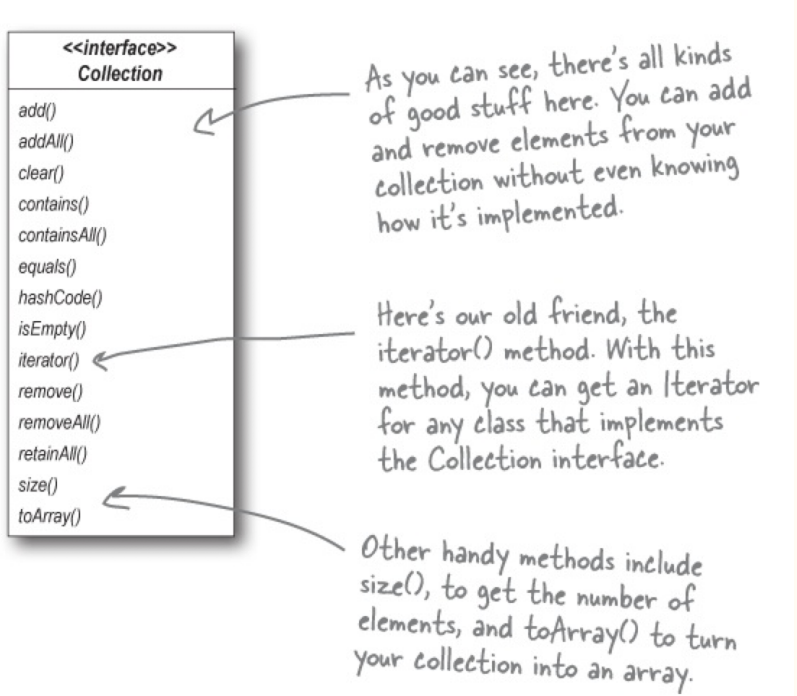
Now, when we print we should see all three menus.

Decoupling of the wa



Iterators and Collections

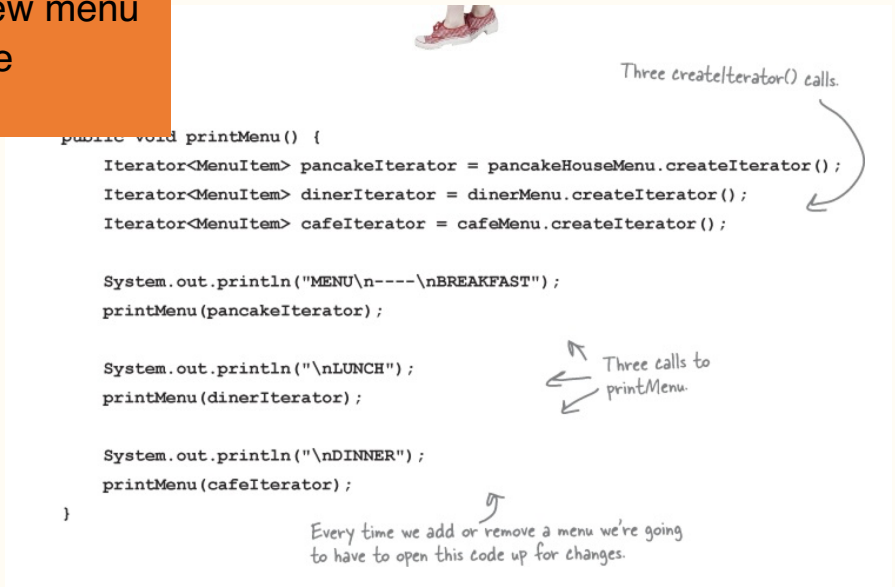
- Java collections Framework
  PriorityQueue
  - Implementation of java.util.C



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

Consideration over the waitress

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



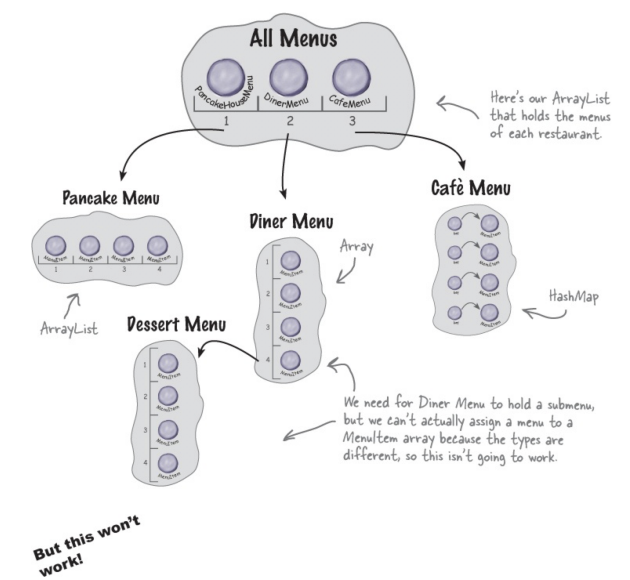
One call for every menu??

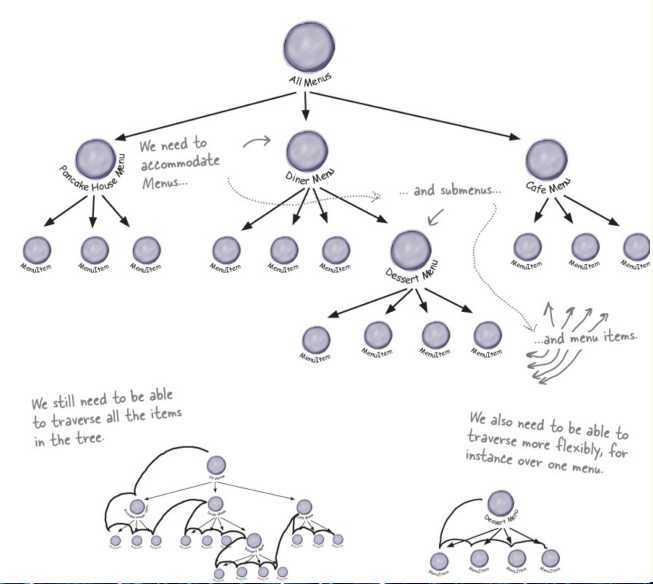
```
public class Waitress {
    ArrayList<Menu> menus;

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

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

    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());
        }
    }
}
```

Now we just take an ArrayList of menus.

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

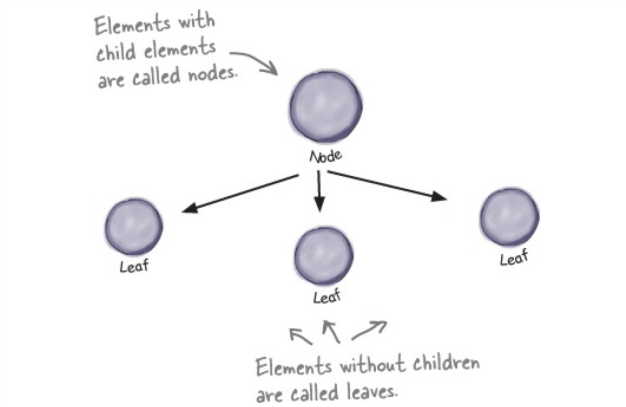
No code changes here.

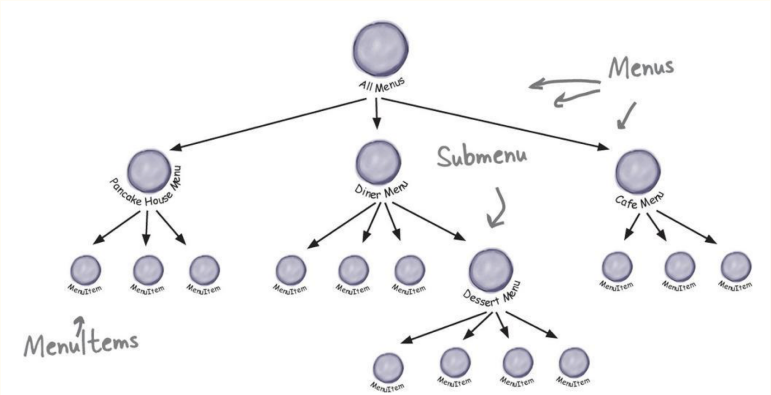

All Menus
PancakeHouseMenu
DinerMenu
CafeMenu
1
2
3
Here's our ArrayList that holds the menus of each restaurant.
Pancake Menu
ArrayList
Diner Menu
Array
Cafè Menu
HashMap
Dessert Menu
We need for Diner Menu to hold a submenu, but we can't actually assign a menu to a MenuItem array because the types are different, so this isn't going to work.
But this won't work!


All Menus
Pancake House Menu
We need to accommodate Menus...
Diner Menu
... and submenus...
Cafe Menu
Dessert Menu
...and menu items.
We still need to be able to traverse all the items in the tree.
We also need to be able to traverse more flexibly, for instance over one menu.

Definition of Composite Pattern

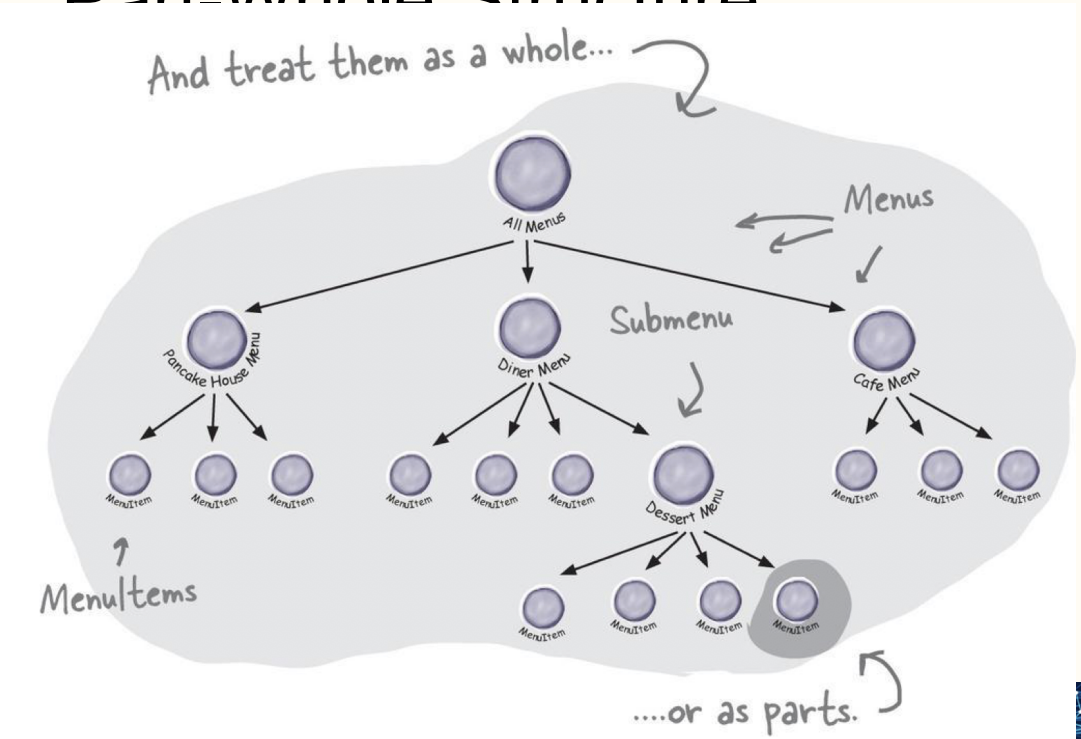
- Allows you to compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly





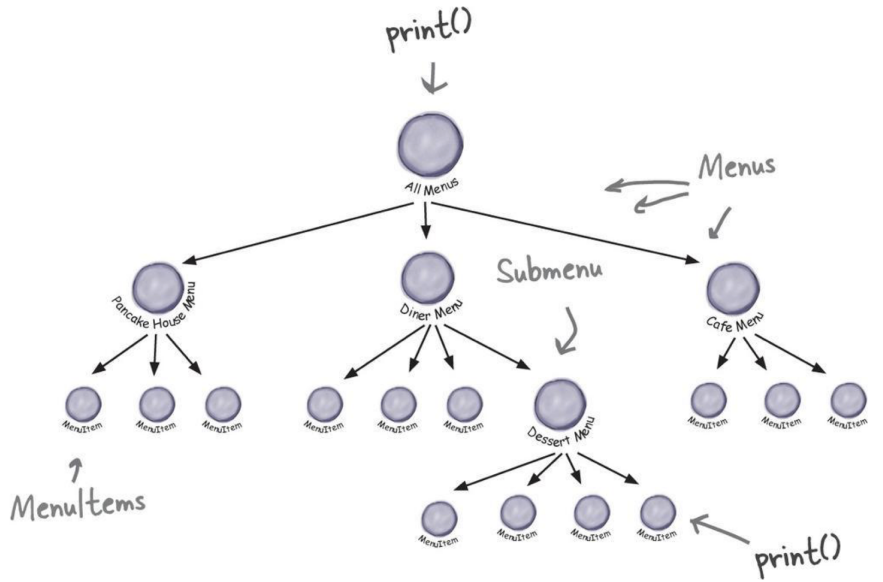
Homogeneous or Heterogeneous Tree??

Part-Whole Structure



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

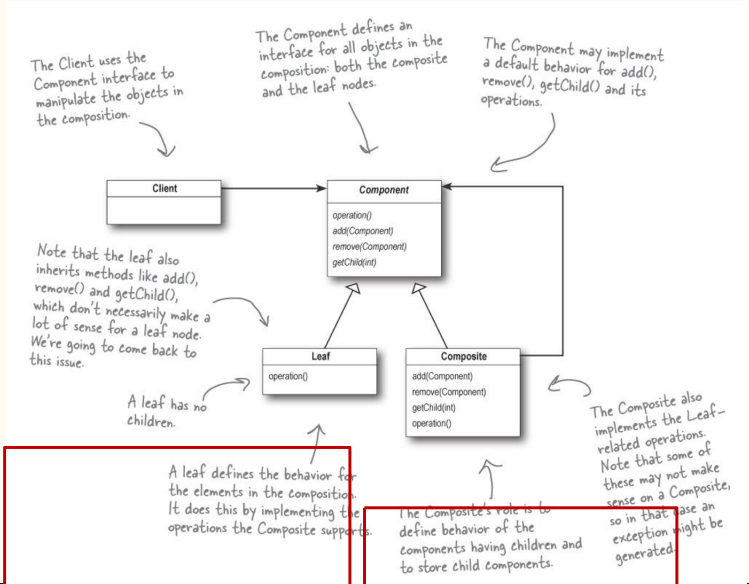
Within a collection, another level of collections as item


print()
All Menus
Menus
Pancake House Menu
Diner Menu
Submenu
Cafe Menu
MenuItem
Dessert Menu
MenuItems
print()

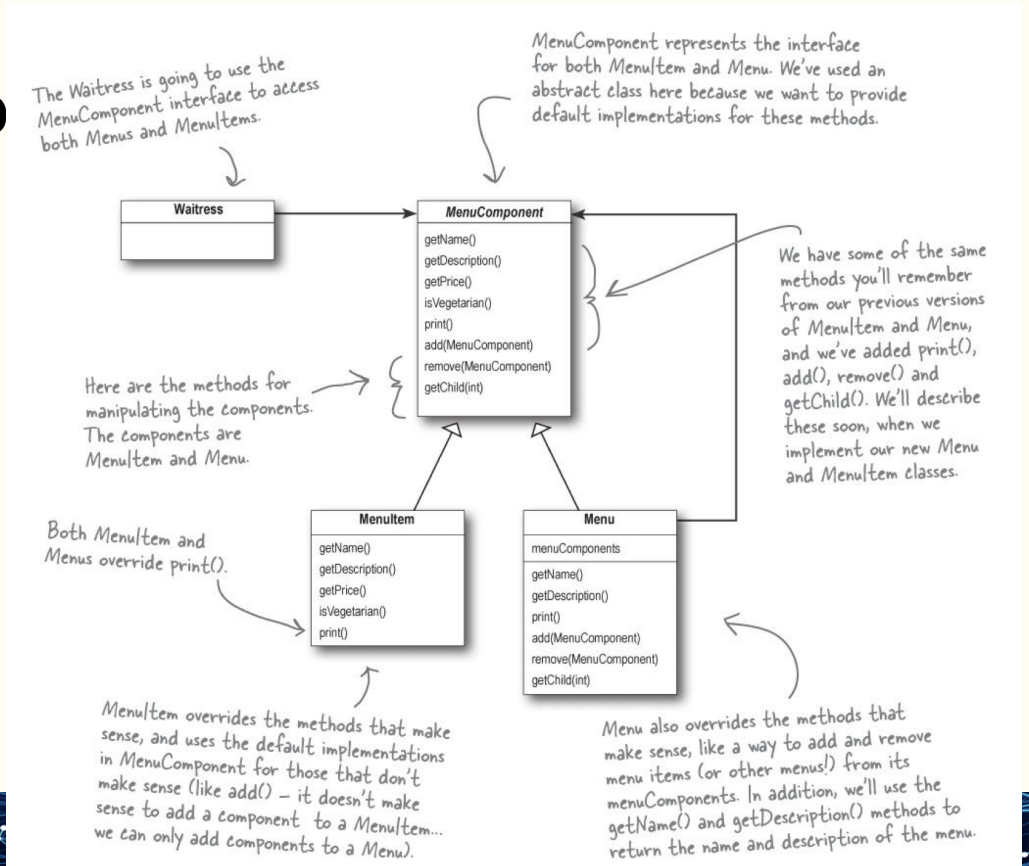
Definition of Composition Pattern

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.


The Client uses the Component interface to manipulate the objects in the composition.
The Component defines an interface for all objects in the composition: both the composite and the leaf nodes.
The Component may implement a default behavior for add(), remove(), getChild() and its operations.
Client
Component
operation()
add(Component)
remove(Component)
getChild(int)
Note that the leaf also inherits methods like add(), remove() and getChild(), which don't necessarily make a lot of sense for a leaf node. We're going to come back to this issue.
Leaf
operation()
Composite
add(Component)
remove(Component)
getChild(int)
operation()
A leaf has no children.
The Composite also implements the Leaf-related operations. Note that some of these may not make sense on a Composite, so in that case an exception might be generated.
A leaf defines the behavior for the elements in the composition. It does this by implementing the operations the Composite supports.
The Composite's role is to define behavior of the components having children and to store child components.

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 MenuItems. 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 MenuItems 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.


```
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) {
        returnmenuComponents.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("---------------------");
    }
}
```

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);
        allMenus.add(dinerMenu);
        allMenus.add(cafeMenu);

        // add menu items here

        dinerMenu.add(new MenuItem(
            "Pasta",
            "Spaghetti with Marinara Sauce, and a slice of sourdough bread",
            true,
            3.89));

        dinerMenu.add(dessertMenu);

        dessertMenu.add(new MenuItem(
            "Apple Pie",
            "Apple pie with a flakey crust, topped with vanilla icecream",
            true,
            1.59));

        // add more menu items here

        Waitress waitress = new Waitress(allMenus);

        waitress.printMenu();
    }
}
```

Let's first create all the menu objects.

We also need a top-level menu that we'll name allMenus.

We're using the Composite add() method to add each menu to the top-level menu, allMenus.

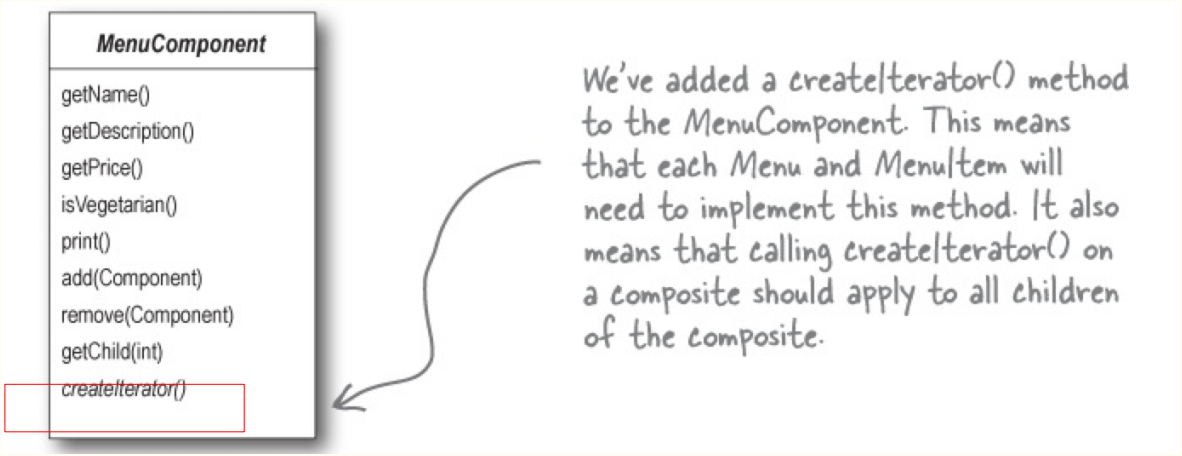
Now we need to add all the menu items. Here's one example; for the rest, look at the complete source code.

And we're also adding a menu to a menu. All dinerMenu cares about is that 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



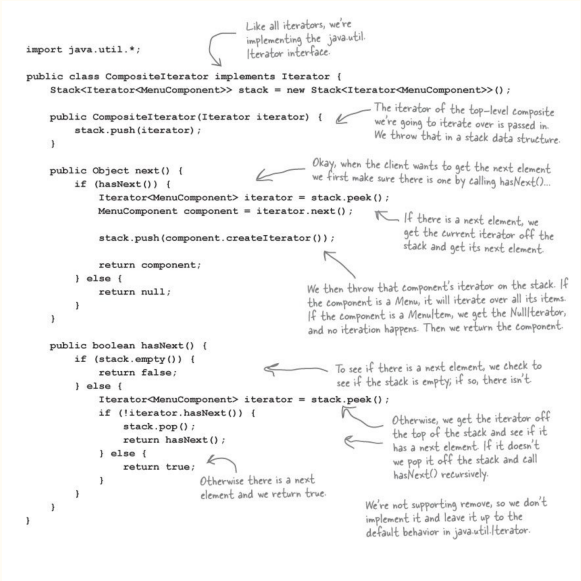
```
public class Menu extends MenuComponent {
    Iterator<MenuComponent> iterator = null;
    // other code here doesn't change

    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

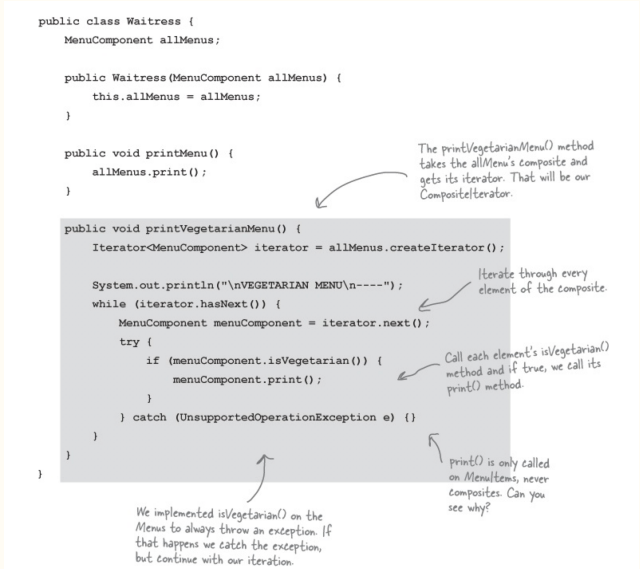
```
import java.util.Iterator;

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 allMenu's composite and gets its iterator. That will be our CompositeIterator.
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 MenuItems, 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.