

# CS525

# Advanced Software Development

## Lesson 9 – The Iterator & Composite Patterns (Part 1: Iterator)

Design Patterns  
*Elements of Reusable Object-Oriented Software*

Payman Salek, M.S.  
March 2022

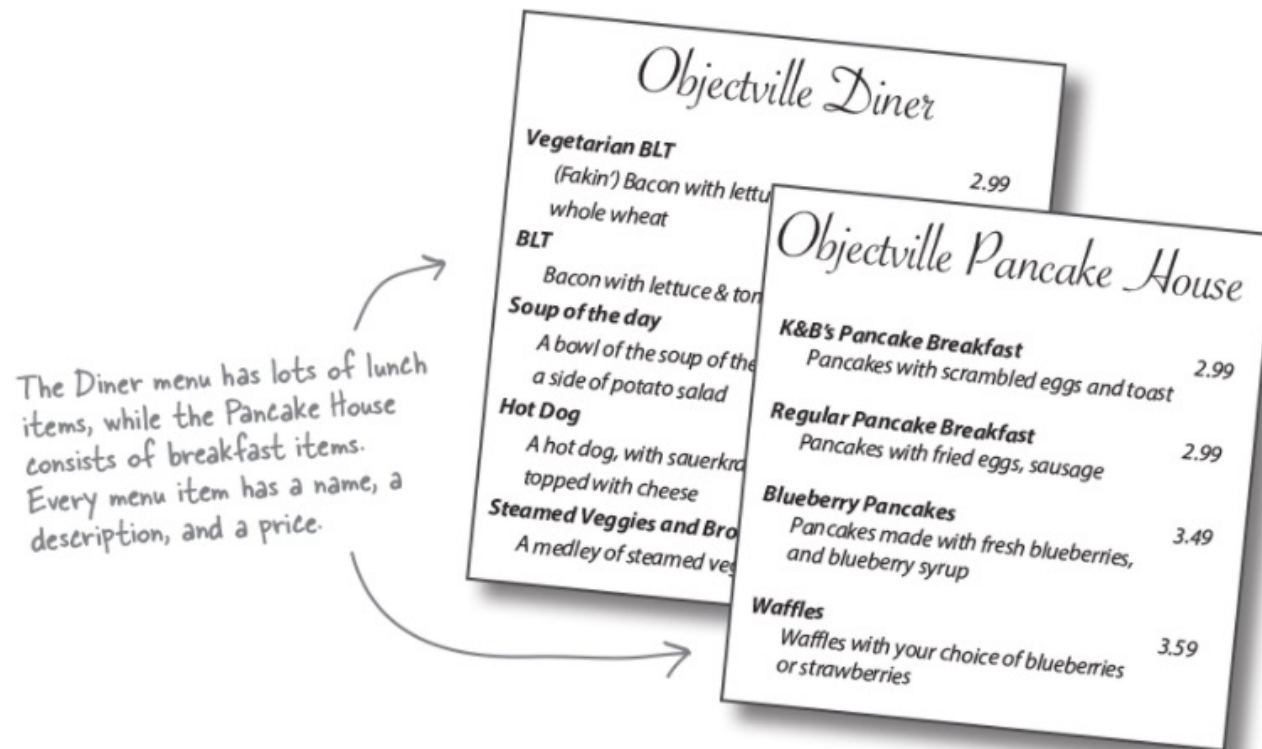
© 2022 Maharishi International University



# The Inspiration

There are lots of ways to stuff objects into a collection. Put them into an Array, a Stack, a List, a hash map—take your pick. Each has its own advantages and tradeoffs. But at some point your clients are going to want to iterate over those objects, and when they do, are you going to show them your implementation? We certainly hope not!

# Setting the stage (Menu Items)



# MenuItem Class

3/3/22

```
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.

# The Pancake House Menu

3/3/22

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

```
public class PancakeHouseMenu {
    List<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 class 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!

# The Diner Menu

3/3/22

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

```
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 sauerkraut, 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
}
```

Mel takes a different approach; he's using an Array class 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.

# The Java-Enabled Waitress (Prints the menu for you)

3/3/22

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

printMenu()
- prints every item on the breakfast and
  lunch menus

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 Waitress is getting  
Java-enabled.

The spec for  
the Waitress

# Step 1: Retrieve the menus

```
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.



## Step 2: Print the menus

```
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...

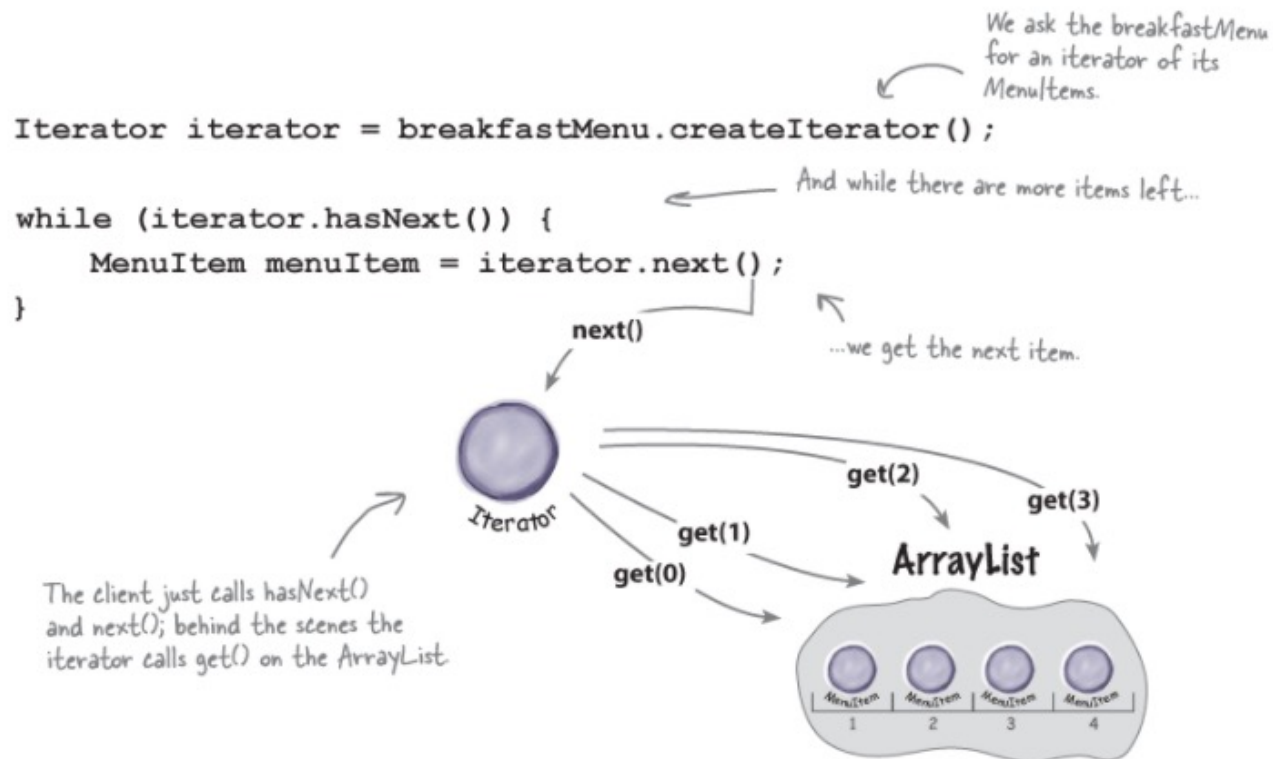
← ...one loop for the ArrayList...

← ...and another for the Array.

# Open-Closed Out the Window!

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.

# Iterator to the Rescue - Breakfast



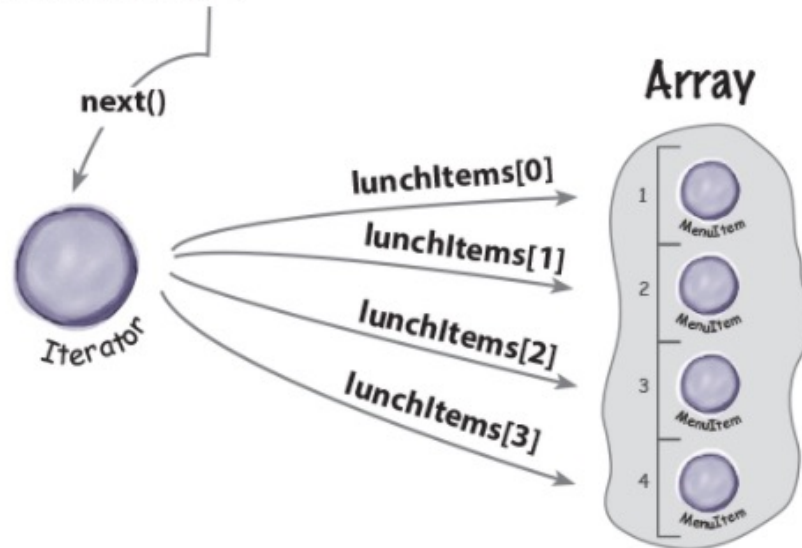
# Iterator to the Rescue - Lunch

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

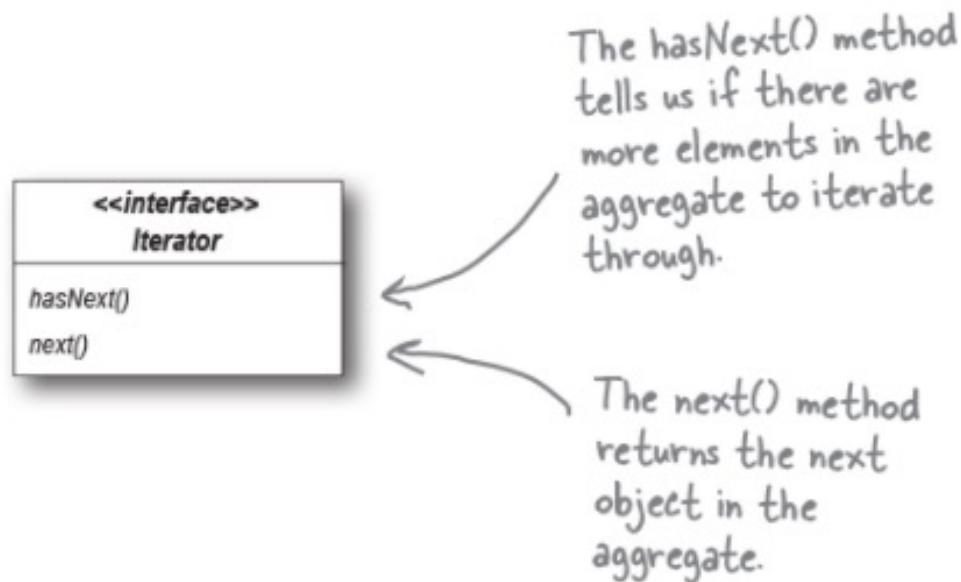
```
while (iterator.hasNext()) {  
    MenuItem menuItem = iterator.next();  
}
```

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.



# The Iterator Class Diagram



# Diner Menu Iterator

```
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

```
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; 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.

# Practice – Pancake House Iterator

See if you can write it yourself...



# Fixing up the Waitress Code

3/3/22

```
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 class 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.

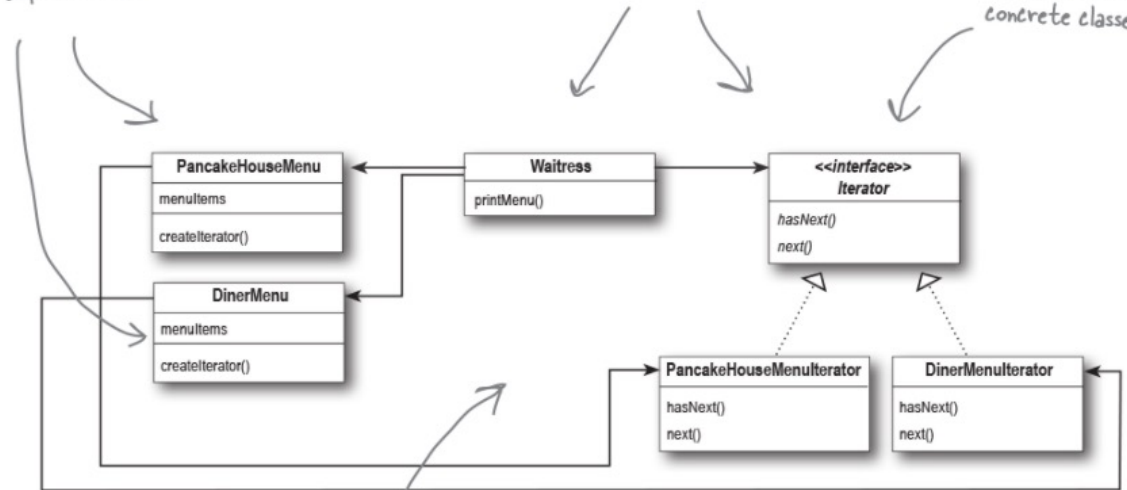
# Reviewing our current design...

3/3/22

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 about 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.

# Cleaning up with java.util.Iterator

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



Instead of creating our own iterator now, we just call the `iterator()` method on the `menuItems` `ArrayList` (more on this in a bit).

# Cleaning up with java.util.Iterator

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

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

```
public class DinerMenuIterator implements Iterator<MenuItem> {
```

```
    MenuItem[] items;  
    int position = 0;
```

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

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

None of our current implementation changes...

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

Remember, the remove() method is optional in the Iterator interface. Having our waitress remove menu items really doesn't make sense, so we'll just throw an exception if she tries.

```
    public void remove() {  
        throw new UnsupportedOperationException  
            ("You shouldn't be trying to remove menu items.");  
    }
```

```
}
```

# Reworked Waitress

3/3/22

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

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

```
public class Waitress {
```

```
    Menu pancakeHouseMenu;
```

```
    Menu dinerMenu;
```

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

```
    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);  
    }
```

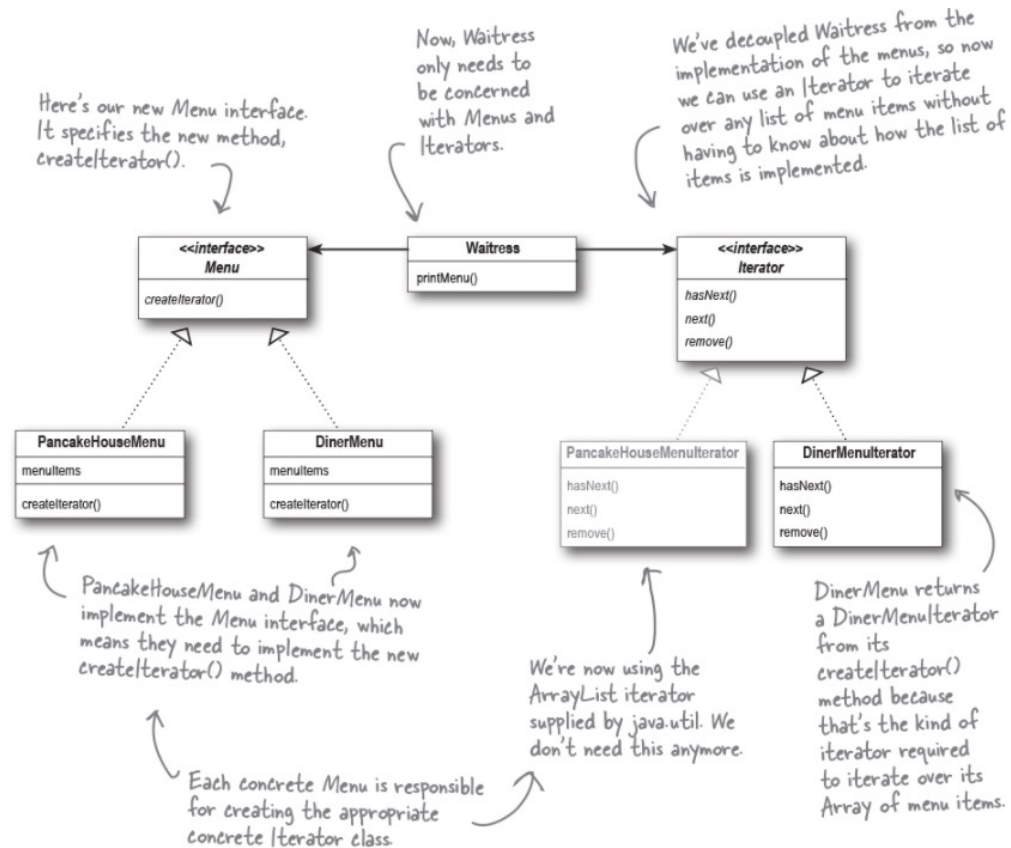
```
    private void printMenu(Iterator iterator) {  
        while (iterator.hasNext()) {  
            MenuItem menuItem = iterator.next();  
            System.out.print(menuItem.getName() + ", ");  
            System.out.print(menuItem.getPrice() + " -- ");  
            System.out.println(menuItem.getDescription());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

```
    // other methods here
```

```
}
```

Nothing changes here.

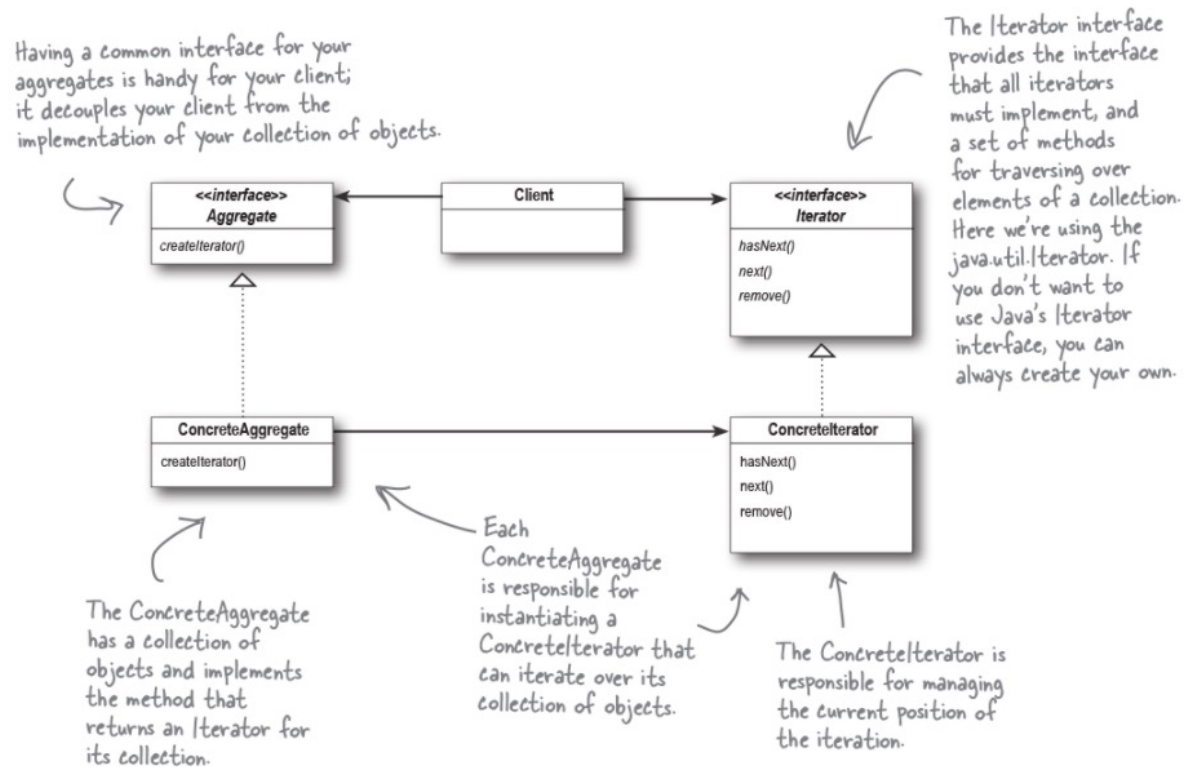
# Reworked Design



# The Iterator Pattern

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

# The Iterator Class Diagram





# Single Responsibility Principle

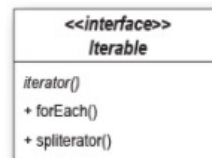
A class should have only one reason to change.

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

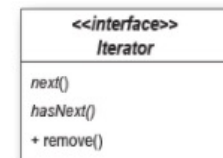
# Java's Iterable Interface

3/3/22

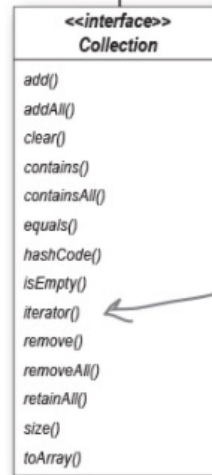
Here's the Iterable interface.



The Iterable interface includes an iterator() method that returns an iterator that implements the Iterator interface.



You already know about the Iterator interface; that's the same interface we've been using with our Diner and Pancake house iterators.



All Collection classes, like ArrayList, implement the Collection interface, which inherits from the Iterable interface, so all Collection classes are Iterables.

# Using Java's Iterator

```
Iterator 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());  
}
```



This is the way we've been doing iteration over our collections, using an iterator along with the `hasNext()` and `next()` methods.

# Using Java's Enhanced For Loop

```
for (MenuItem item: menu) {  
    System.out.print(menuItem.getName() + ", ");  
    System.out.print(menuItem.getPrice() + " -- ");  
    System.out.println(menuItem.getDescription());  
}
```

← Here we can dispense with the explicit iterator as the hasNext() and next() methods.

# External vs. Internal Iterator

- We have seen so far an external iterator, which means that the client controls the iteration by calling `next()` to get the next element.
- An internal iterator is controlled by the iterator itself. In that case, because it's the iterator that's stepping through the elements, you have to tell the iterator what to do with those elements as it goes through them. That means you need a way to pass an operation to an iterator.

# Example: External Iterator

```
Iterator 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());  
}
```

← This is the way we've been doing iteration over our collections, using an iterator along with the `hasNext()` and `next()` methods.

# Example: Internal Iterator

Here's an Iterable, in this case  
our Pancake House ArrayList  
of menu items.

We're calling `forEach()`...

...and passing a lambda that takes a  
menu item, and just prints it.

```
breakfastItems.forEach(item -> System.out.println(item));
```

So this code will print every  
item in the collection.

```
default void forEach(Consumer<? super T> action) {  
    Objects.requireNonNull(action);  
    for (T t : this) {  
        action.accept(t);  
    }  
}
```

# Acquisition: The Café Menu

3/3/22

```
public class CafeMenu {  
    Map<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(name, menuItem);  
    }  
  
    public Map<String, MenuItem> getMenuItems() {  
        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 HashMap.

The key is the item name.

The value is the menuItem object.

We're not going to need this anymore.



# Reworking the Café Menu

```
public class CafeMenu implements Menu {  
    Map<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(name, menuItem);  
    }  
  
    public Map<String, MenuItem> getMenuItems() {  
            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 values.

Just like before, we can get rid of getMenuItems() 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.

# Adding the Café Menu to the Waitress

3/3/22

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

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

```
    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);  
    }
```

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!

```
    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());  
        }  
    }  
}
```

Nothing changes here.

# Summary

3/3/22

