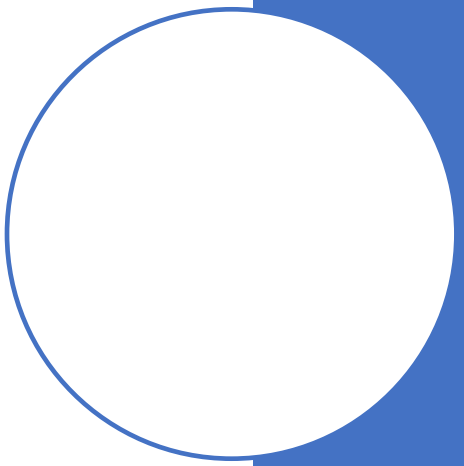
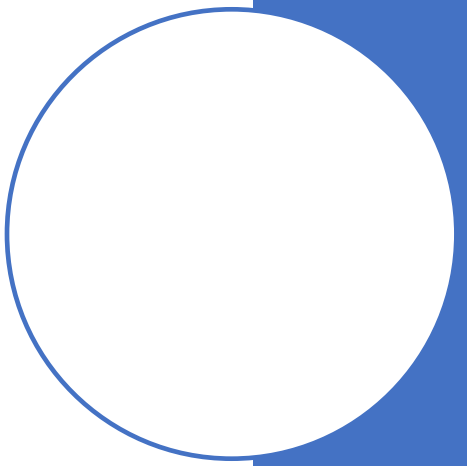
Modeling Your Deployed System: Deployment Diagrams



Modeling Your Deployed System: Deployment Diagrams

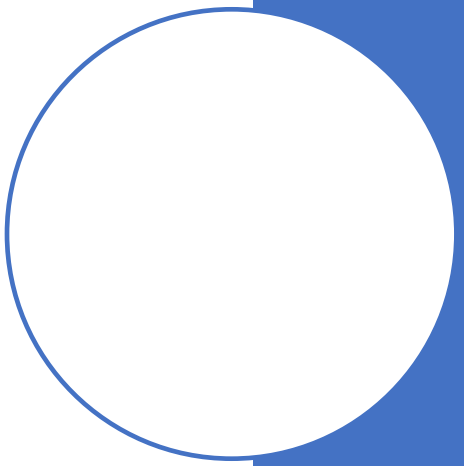
UML deployment diagrams show the physical view of your system, bringing your software into the real world by showing how software gets assigned to hardware and how the pieces communicate.



Modeling Your Deployed System: Deployment Diagrams

Let's start by showing a deployment diagram of a very simple system.

In this simplest of cases, your software will be delivered as a single executable file that will reside on one computer.



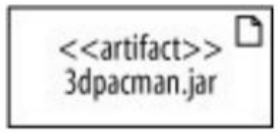
Modeling Your Deployed System: Deployment Diagrams

To show computer hardware, we use a **node**. Can a node also represent software?

To show software, we use **artifact**.

Use nodes to represent hardware in your system

A physical software file such as a jar file is modeled with an artifact



Putting Pieces Together

Drawing an artifact inside a node shows that the artifact is deployed to the node

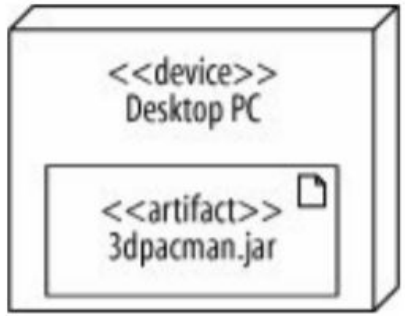
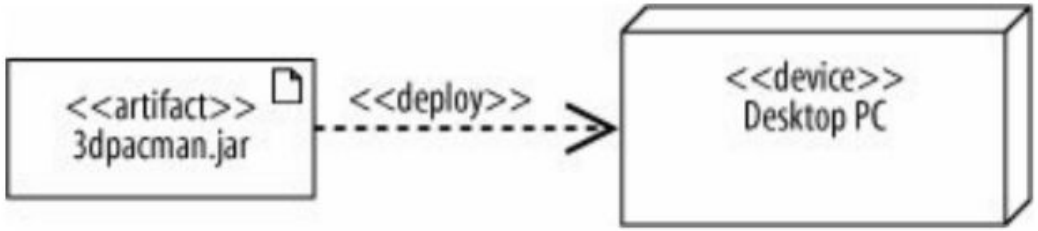


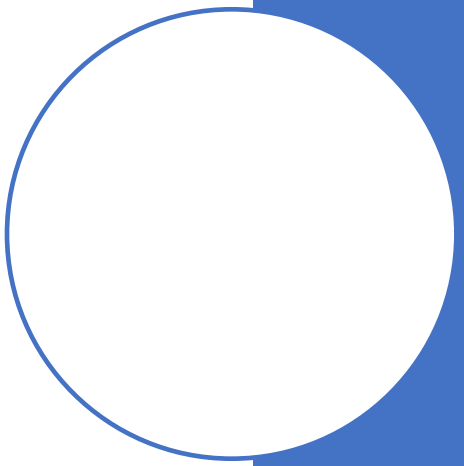
Figure 15-7. An alternate way to model the relationship deployment



Artifacts

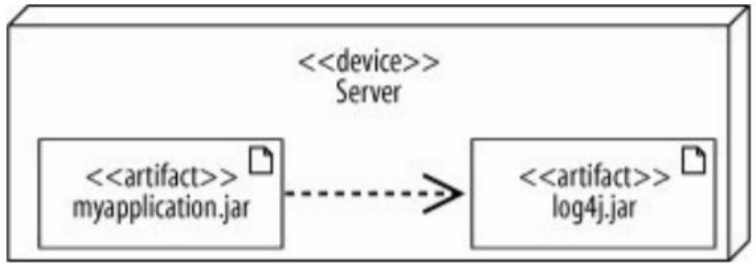
Artifacts are physical files that execute or are used by your software. Common artifacts you'll encounter include:

- Executable files, such as .exe or .jar files
- Library files, such as .dlls (or support .jar files)
- Source files, such as .java or .cpp files
- Configuration files that are used by your software at runtime, commonly in formats such as .xml, .properties, or .txt



Managing Dependencies between Artifacts

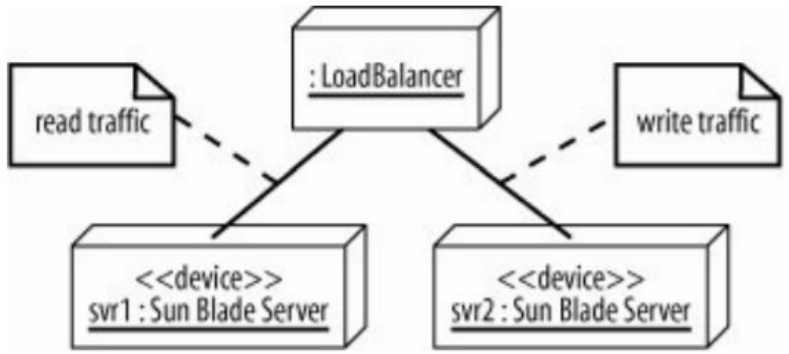
A deployment notation that uses artifact symbols (instead of listing artifact names) allows you to show artifact dependencies



Communication Between Nodes Example

Communication paths are used to show that nodes communicate with each other at runtime. A communication path is drawn as a solid line connecting two nodes

Figure 15-16. One node gets read traffic and the other gets write traffic



Communication Between Nodes

For example, a client application running on a desktop PC may retrieve data from a server using TCP/IP.

When to Use a Deployment Diagram

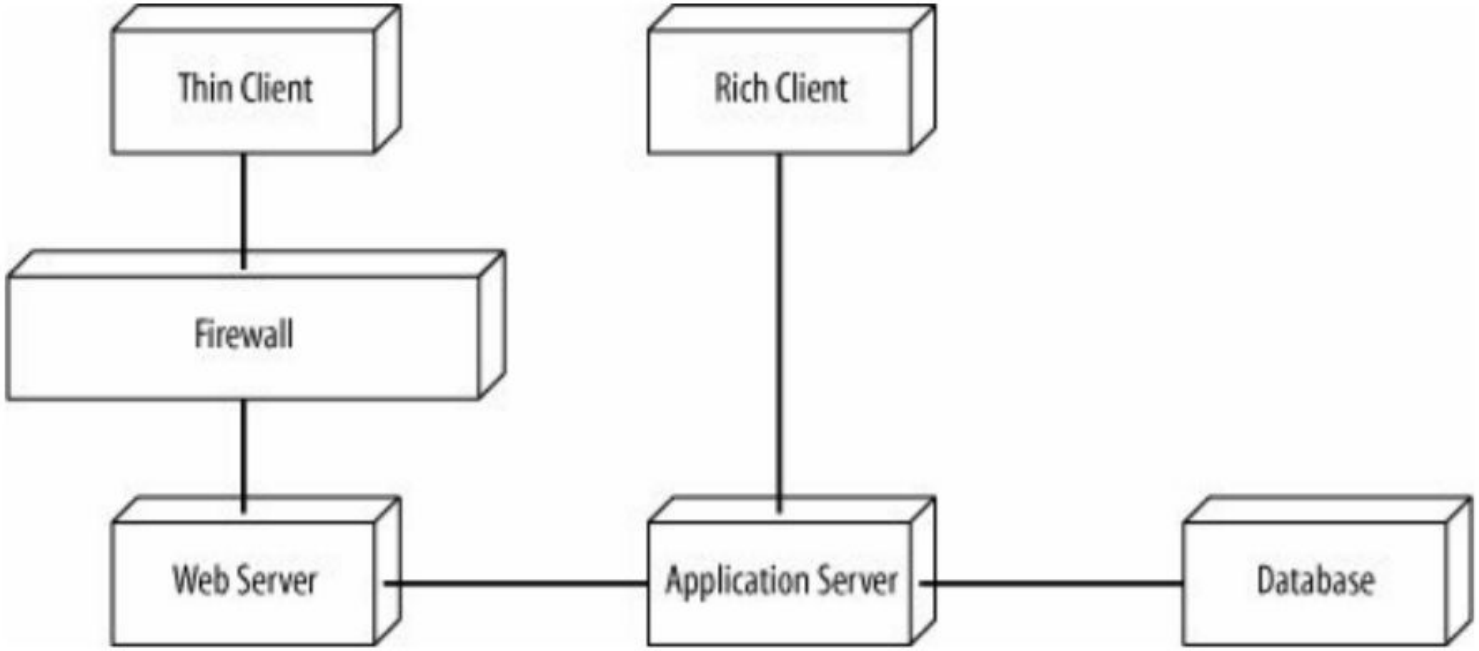
Deployment diagrams are useful at all stages of the design process. Let suppose you want to communicate important characteristics of your system, such as the following:

- Your architecture includes a web server, application server, and database.
- Clients can access your application through a browser or through a richer GUI interface.
- The web server is protected with a firewall.

Even at the early stage (when we are not very sure about certain things e.g., you may not have decided which hardware to use) you can use deployment diagrams to model these characteristics.

When to Use a Deployment Diagram

Figure 15-21. A rough sketch of your web application



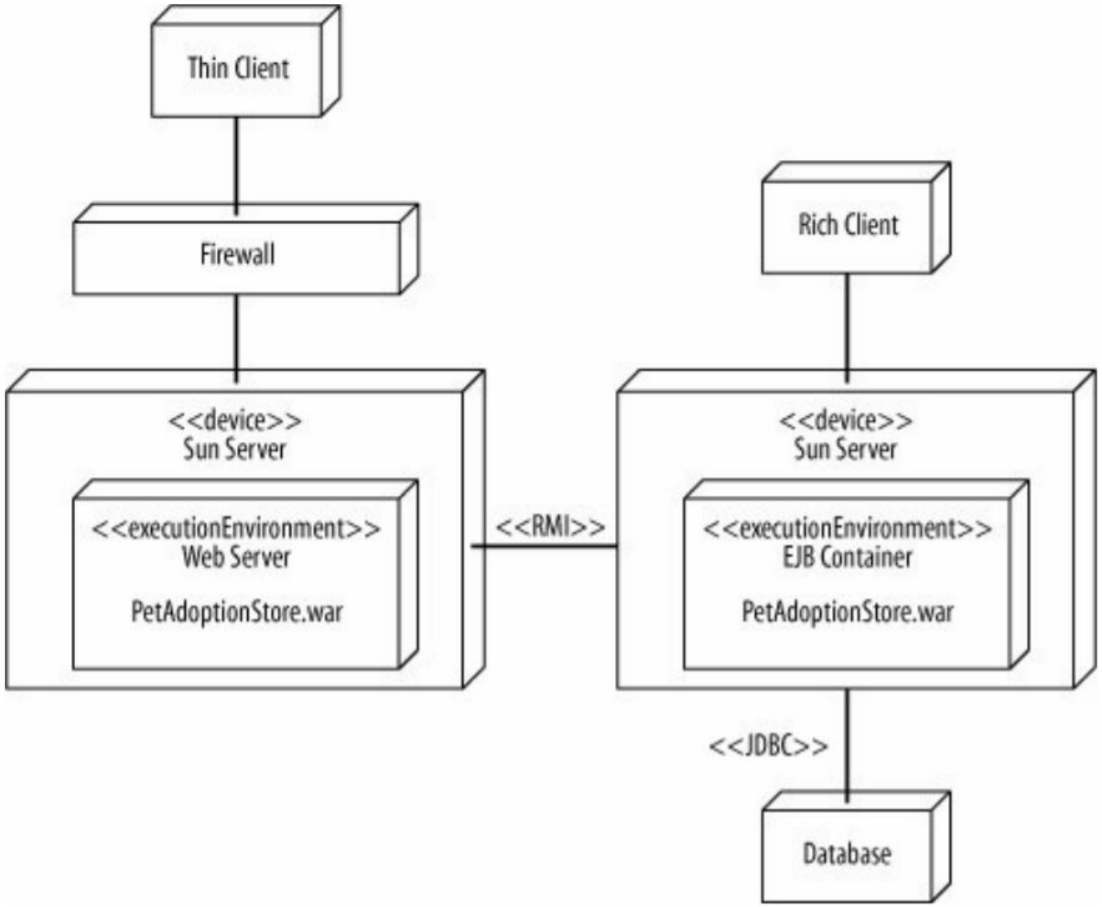
When to Use a Deployment Diagram

You can revisit your deployment diagrams throughout the design of your system to refine the rough initial sketches, adding detail as you decide which technologies, communication protocols, and software artifacts will be used.

The next one is the detailed deployment diagram which is more specific about the hardware types, the communication protocols, and the allocation of software artifacts to nodes

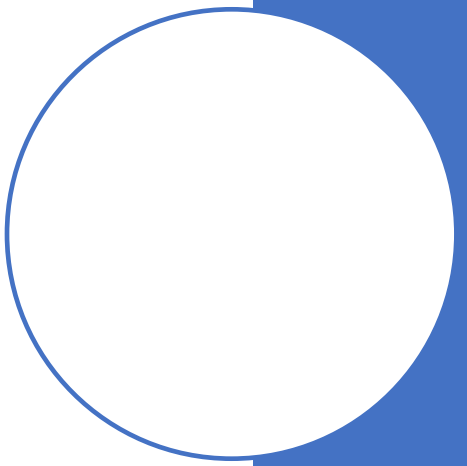
When to Use a Deployment Diagram

A detailed deployment diagram specifying a J2EE implementation of the system.



Design Patterns

Iterator Pattern



Using a Simple Approach

1. To print all the items on each menu, you'll need to call the getMenuItem() method on the PancakeHouseMenu and the DinerMenu to retrieve their respective menu items. Note that each returns a different type:

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

DinerMenu dinerMenu = new DinerMenu();
MenuItem[] lunchItems = dinerMenu.getMenuItems();
```

The method looks the same, but the calls are returning different types.

The implementation is showing through, breakfast items are in an ArrayList, lunch items are in an Array.

Using a Simple Approach

2. Now, to print out the items from the PancakeHouseMenu, we'll loop through the items on the breakfastItems ArrayList. And to print out the Diner items we'll loop through the Array.

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

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

Now, we have to implement two different loops to step through the two implementations of the menu items...

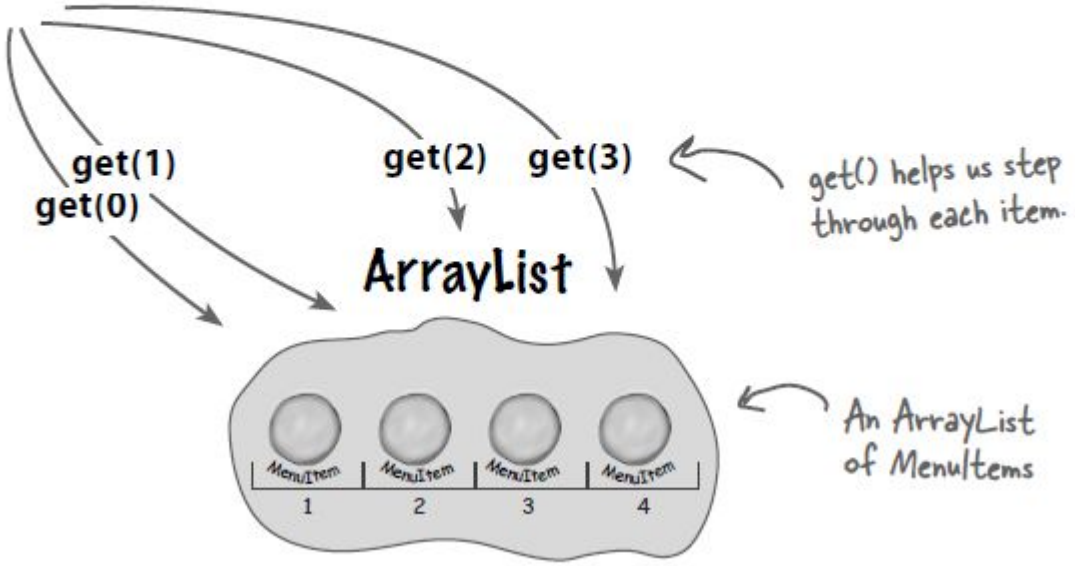
...one loop for the ArrayList...

and another for the Array.

Using a Simple Approach

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

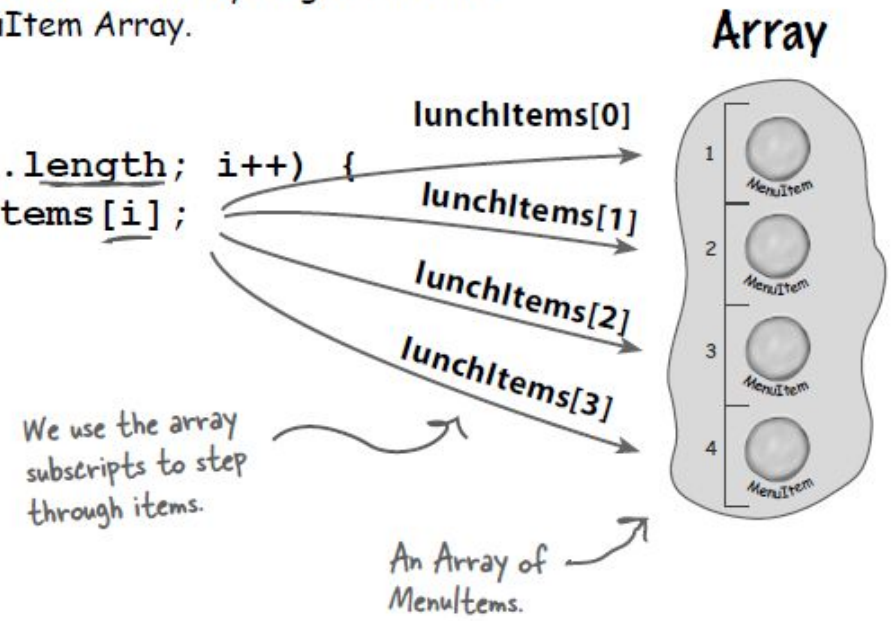
```
for (int i = 0; i < breakfastItems.size(); i++) {
    MenuItem menuItem = (MenuItem)breakfastItems.get(i);
}
```



Using a Simple Approach

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

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



Let see how can we solve the problem using the Iterator Pattern?

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

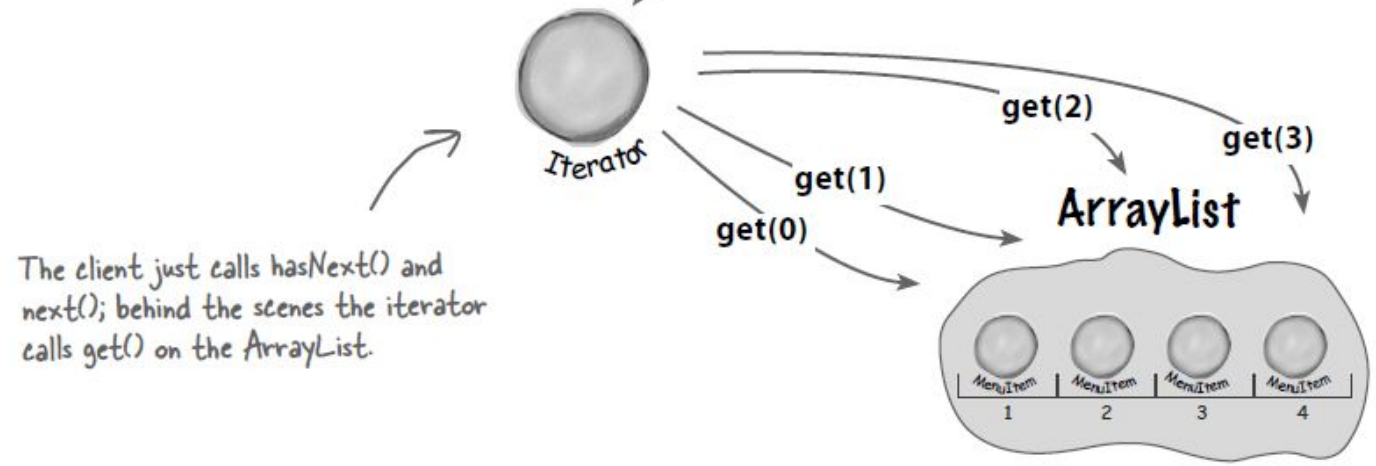
```
Iterator iterator = breakfastMenu.createIterator();

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

We ask the breakfastMenu for an iterator of its MenuItems.

And while there are more items left...

next()

We get the next item.



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

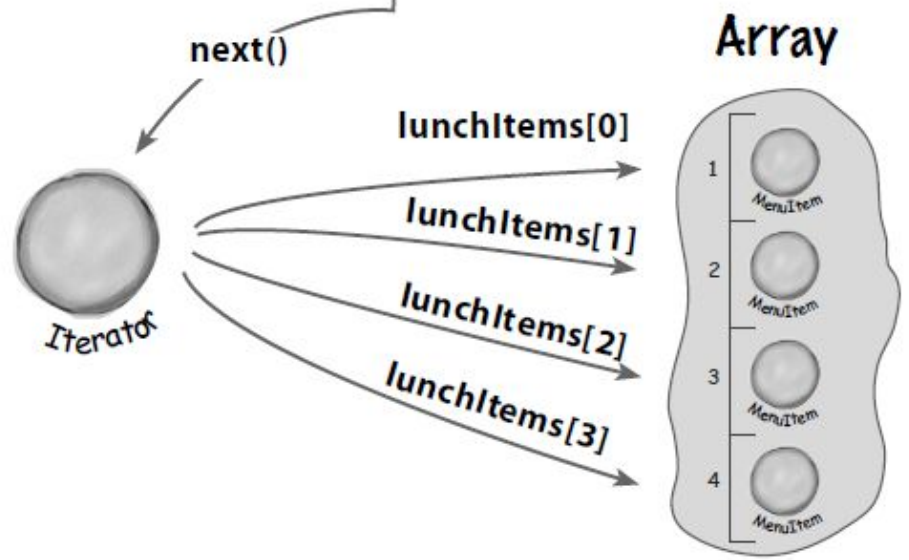
Let see how can we solve the problem using the Iterator Pattern?

4 Let's try that on the Array too:

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

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

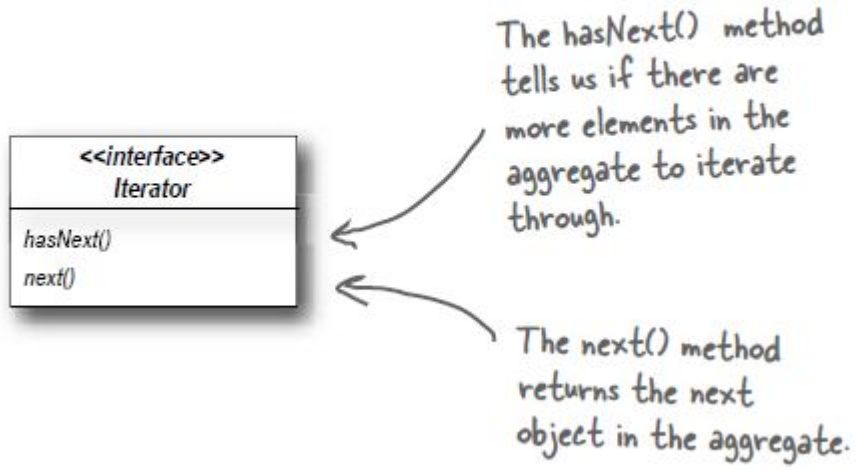
Wow, this code is exactly the same as the breakfastMenu code.

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



Step by Step Class Diagram

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



Step by Step Class Diagram

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

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

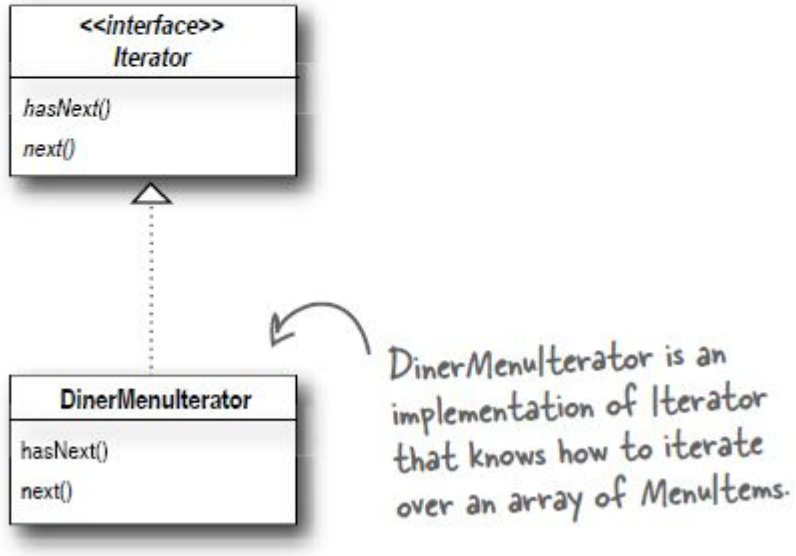
Here's our two methods:

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

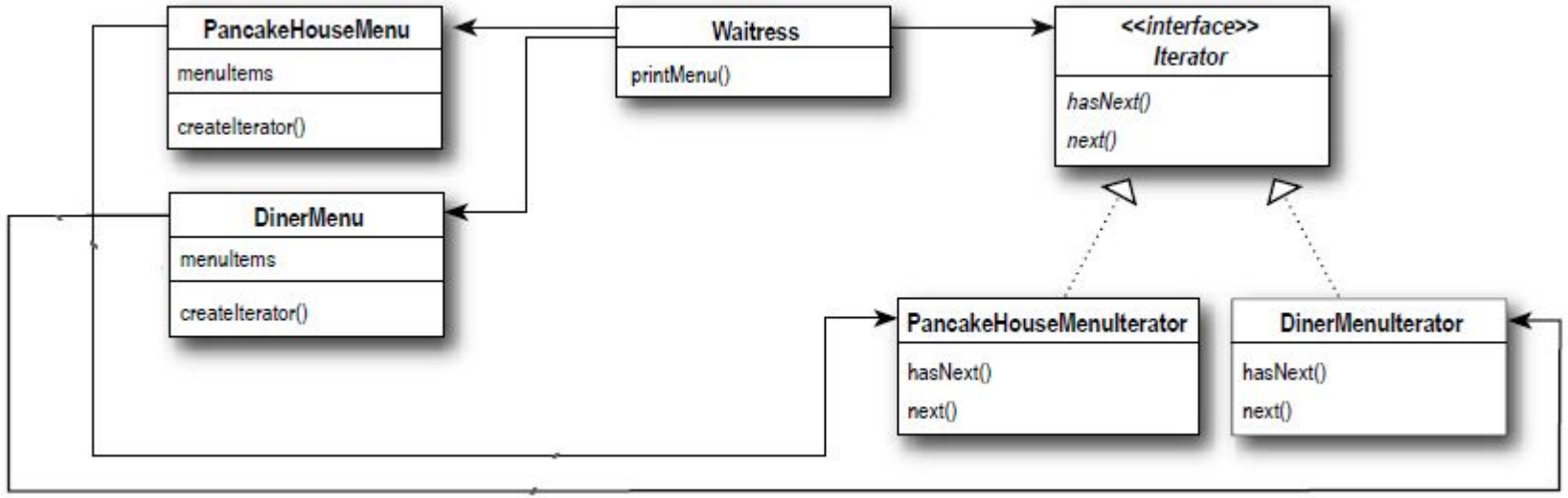
...and the next() method returns the next element.

How the implementation starts?

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



Class Diagram



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

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

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

    public Object next() {
        MenuItem menuItem = items[position];
        position = position + 1;
        return menuItem;
    }

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

We implement the Iterator interface.

position maintains the current position of the iteration over the array.

The constructor takes the array of menu items we are going to iterate over.

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

The hasNext() method checks to see if we've seen all the elements of the array and returns true if there are more to iterate through.

Because the diner chef went ahead and allocated a max sized array, we need to check not only if we are at the end of the array, but also if the next item is null, which indicates there are no more items.

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

```
public class DinerMenu {
    static final int MAX_ITEMS = 6;
    int numberOfItems = 0;
    MenuItem[] menuItems;

    // constructor here

    // addItem here

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

    public Iterator createIterator() {
        return new DinerMenuIterator(menuItems);
    }

    // other menu methods here
}
```

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

Here's the createIterator() method. It creates a DinerMenuIterator from the menuItems array and returns it to the client.

We're returning the Iterator interface. The client doesn't need to know how the menuItems are maintained in the DinerMenu, nor does it need to know how the DinerMenuIterator is implemented. It just needs to use the iterators to step through the items in the menu.

Waitress Class / Client

```
public class Waitress {
    PancakeHouseMenu pancakeHouseMenu;
    DinerMenu dinerMenu;

    public Waitress(PancakeHouseMenu pancakeHouseMenu, DinerMenu dinerMenu) {
        this.pancakeHouseMenu = pancakeHouseMenu;
        this.dinerMenu = dinerMenu;
    }

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

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

    // other methods here
}
```

In the constructor the Waitress takes the two menus.

The printMenu() method now creates two iterators, one for each menu.

And then calls the overloaded printMenu() with each iterator.

Test if there are any more items.

Get the next item.

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

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

Note that we're down to one loop.