Decorator / Adapter patterns

Advanced Java Programming 2012-13

OO Design Patterns and Principles

Lecture Outline

- Decorator pattern
- Adapter pattern

The Decorator Pattern

The Decorator pattern

- A burger restaurant called *BurgerQueen* offer their customers the chance to design their own burger
  - 'Have it your way!'
- The customer starts with a basic beef, chicken or veggie patty, then adds an arbitrary number of optional toppings (bacon, cheese, mayonnaise and/or ketchup)
- *BurgerQueen* come to you for advice because their software is causing them problems
- When a burger is sold, *BurgerQueen* need two things
  - The overall cost (burger price + topping(s))
  - An alphabetic code which represents the burger, for stock-keeping purposes e.g. CBmb means 'a chicken burger with mayonnaise and bacon'

The Decorator pattern

ITEM	TILL PRICE	TILL CODE
Beef burger	1.70	BB
Vegetable burger	1.60	VB
Chicken burger	1.55	CB
Cheese	0.10	c
Bacon	0.20	b
Mayonnaise	0.05	m
Ketchup	0.05	k

Examples

Beef burger with cheese, mayonnaise	**BBcm**	**£1.85**
Vegetable burger with cheese, bacon	**VBcb**	**£1.90**
Chicken burger with everything	**Cbcbmk**	**£1.95**

The Decorator pattern

- The existing software solution uses inheritance to model the problem
- BQ have created an abstract class, called *Burger*, that defines the state and behavior of all burgers
  - All *Burgers* have a *getPrice*() and *getCode*() method
- This abstract class is extended three times (*BeefBurger*, *ChickenBurger*, *VeggieBurger)*
- Each of these concrete classes is then sub-classed again, to represent a particular burger with a particular topping
  - e.g. *Burger* → *VegetableBurger* → *VegetableBurgerCheese*

The Decorator pattern

```
class Burger
{
    protected String code;

    protected double price;

    public String getCode()
    {
        return code;
    }

    public double getPrice()
    {
        return price;
    }
}
```

The abstract class, which represents all burgers

The Decorator pattern

```
class ChickenBurger extends Burger
{
    public ChickenBurger()
    {
        code  = “CB”;
        price = 1.55;
    }
}

class BeefBurger extends Burger
{
    public BeefBurger()
    {
        code  = “BB”;
        price = 1.70;
    }
}
```

Two sub-classes, which represents beef and chicken burgers

The Decorator pattern

```
class ChickenBurgerMayo extends Burger
{
    public ChickenBurgerMayo()
    {
        code  = “CBm”;
        price = 1.60;
    }
}

class BeefBurgerCheese extends Burger
{
    public BeefBurgerCheese()
    {
        code  = “BBc”;
        price = 1.80;
    }
}
```

Two sub-sub-classes. The price and code are always initialised in the constructor

The Decorator pattern

```
// Tills are ringing up the purchases, creating objects
tillSales1.add(new BeefBurger());
tillSales1.add(new ChickenBurgerMayo());
tillSales1.add(new BeefBurgerBacon());

// Caching up till at end of day
double totalSales=0.0;
for(Burger b : tillSales1) {
   totalSales += b.getPrice();
}

// Stock taking data for inventory
String productData="";
for(Burger b : tillSales1) {
   productsData += b.getCode();
}
```

Usage of class library (requires suspension of disbelief, bear with me)

The Decorator pattern

- The problem – the permutations of burger plus arbitrary toppings leads inevitably to a *class explosion*
  - Even assuming that BQ customers can have only one topping of each type, there are 12 classes
  - If BQ allows multiple toppings of the same kind (e.g. bacon double cheese burger) there are thousands of permutations
  - The addition of one extra topping to the menu (e.g. chili beef) leads to exponential increases in the class library
  - A change in the price of one of the toppings (e.g. cheese goes up to 15p due to cow shortage) requires multiple edits across the class library
- Clearly, this is a terrible solution
  - The code has a *bad smell* to it – instinctively it feels wrong
  - DRY – don't repeat yourself

The Decorator pattern

- The *BurgerQueen* software developers have just read a book on refactoring and they have another suggestion
- They want to revert to just 4 classes
  - Burger (abstract)
  - BeefBurger
  - ChickenBurger
  - VegetableBurger
- Toppings would be represented as *instance variables* of type int, declared in the superclass *Burger*
- These variables would be manipulated by the subclasses to create the correct burger
  - They suggest something like this

The Decorator pattern

```
abstract class Burger
{
        protected int cheese, mayo, bacon, ketchup;

        public Burger(int cheese, int mayo, int bacon, ketchup)
        { // Initialise all fields
        }

        public double getPrice()
        {
          return (mayo * 0.05) + (cheese * 0.10) (bacon *
            0.20) ( ketchup * 0.05)  + price;
        }

        public double getCode()
        {  String s = "";
           while(cheese > 0){ s += "c"; cheese--;} // etc.
           return code + s;
        }
}
```

Representing all toppings as instance variables in the superclass. Please never declare your instance variables like this.

The Decorator pattern

```
class BeefBurger extends Burger
{
      public Burger(int cheese, int mayo, int bacon, ketchup)
      {
         super(cheese, mayo, bacon, ketchup);
         price = 1.80;
      }
}

// Usage – basic beef burger
Burger b1 = new BeefBurger(0,0,0,0);

// Bacon double cheese burger with ketchup
Burger b2 = new BeefBurger(2,0,1,1);

b2.getPrice();
b2.getCode();
```

Variation – superclass has no-args constructor that initialises all toppings to 0. Sub-classes call super constructor to initialise then use methods to access fields e.g. `tillSales1.add(new BeefBurger().addCheese());`

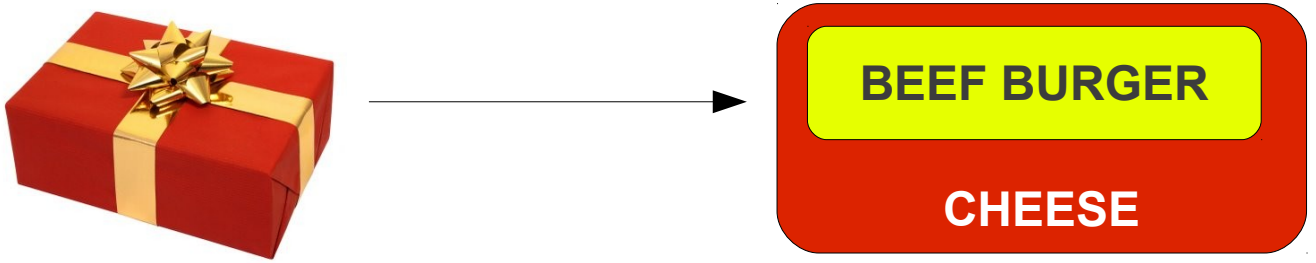
The Decorator pattern

- **Advantages**
  - Massive reduction in number of classes (from ? to 4)
  - We can represent multiple toppings of same kind
- **Disadvantages**
  - The addition of new toppings (e.g. chili beef) or changes in the price of a topping will require changes to the superclass
- Here we come across a fundamental design principle
  - Where possible, classes should be open for *extension*, but closed for modification
  - Translation: The *Burger* superclass should be written and then left alone – any changes to the class should be made via extension
  - We will cover this principle in more detail later in the term

So, how do we fix it?

The Decorator pattern

- This is the perfect scenario for the *Decorator* design pattern
- The *Decorator* pattern provides a flexible alternative to sub-classing when you need to extend functionality
- A decorated object is an object that is wrapped by another, in the same way you wrap a Christmas present
- In this metaphor, the base concrete class (e.g. a *BeefBurger*) is the present and the topping (e.g. *Cheese*) is the wrapping



Let's see the code..

The Decorator pattern

```
class Burger
{
    protected String code;

    protected double price;

    public String getCode()
    {
        return code;
    }

    public double getPrice()
    {
        return price;
    }
}
```

As before....

The Decorator pattern

```
class ChickenBurgerMayo extends Burger
{
    public ChickenBurgerMayo()
    {
        code  = “CBm”;
        price = 1.60;
    }
}

class BeefBurgerCheese extends Burger
{
    public BeefBurgerCheese()
    {
        code  = “BBc”;
        price = 1.80;
    }
}
```

Again, as before. But now comes the clever bit. Are you ready? Here it comes. Cleverness incoming....

The Decorator pattern

- Now we define another abstract class that represents all possible toppings
- There are three important things to note
  - This class extends *Burger*
  - It contains a reference to a (wrapped) *Burger* object which is initialised by the constructor
  - All method calls are delegated to the wrapped *Burger*
- AAAAaaaaahhhh. That makes less sense than *Prometheus*
  - Let's break it down

The Decorator pattern

Prepare to eat some polymorphism!

The Decorator pattern

```
public abstract class BurgerTopping extends Burger
{
    public BurgerTopping(Burger b)
    { this.b = b;}

     public double getCost()
     {
         return b.getCost() + cost;
     }

     public String getCode()
     {
         return b.getCode() + code;
     }
}
```

BurgerTopping **extends** ***Burger*****, therefore** ***BurgerTopping*** **is-a** ***Burger*****, as far as Java is concerned. This means we can use** ***BurgerTopping*** **objects in places we use** ***Burgers*** ***e.g.*** **a collection of** ***Burger*** **objects**

The Decorator pattern

```
public abstract class BurgerTopping extends Burger
{
    protected Burger b;
    public BurgerTopping(Burger b)
    { this.b = b;}

     public double getCost()
     {
         return b.getCost() + cost;
     }

     public String getCode()
     {
         return b.getCode() + code;
     }
}
```

BurgerTopping **has an instance variable of type** ***Burger*****. This is the wrapped object.** ***BurgerTopping*** **is decorating this wrapped object. We initialise this field in the constructor.**

The Decorator pattern

```
public abstract class BurgerTopping extends Burger
{
    protected Burger b;
    public BurgerTopping(Burger b)
    { this.b = b;}

     public double getCost()
     {
         return b.getCost() + cost;
     }

     public String getCode()
     {
         return b.getCode() + code;
     }
}
```

All calls to the the methods *getCost*() and *getCode*() are delegated to the wrapped object. The cost of a wrapped object is its cost plus the cost of the thing it wraps. Arbitrary number of layers are allowed.

The Decorator pattern

- The final step involves creating all of the concrete toppings, all of which extend *BurgerTopping*
  - Cheese
  - Bacon
  - Mayonnaise
  - Ketchup
- Each concrete topping must
  - Initialise its own *price* field
  - Initialise its own *code* field
  - Initialise the wrapped object, usually using the superconstructor

The Decorator pattern

```
public class Cheese extends BurgerTopping
{
    public Cheese(Burger b)
    {
        super(b);
        price = 0.10;
        code = "c";
    }
}
```

The other concrete toppings are almost identical

The Decorator pattern

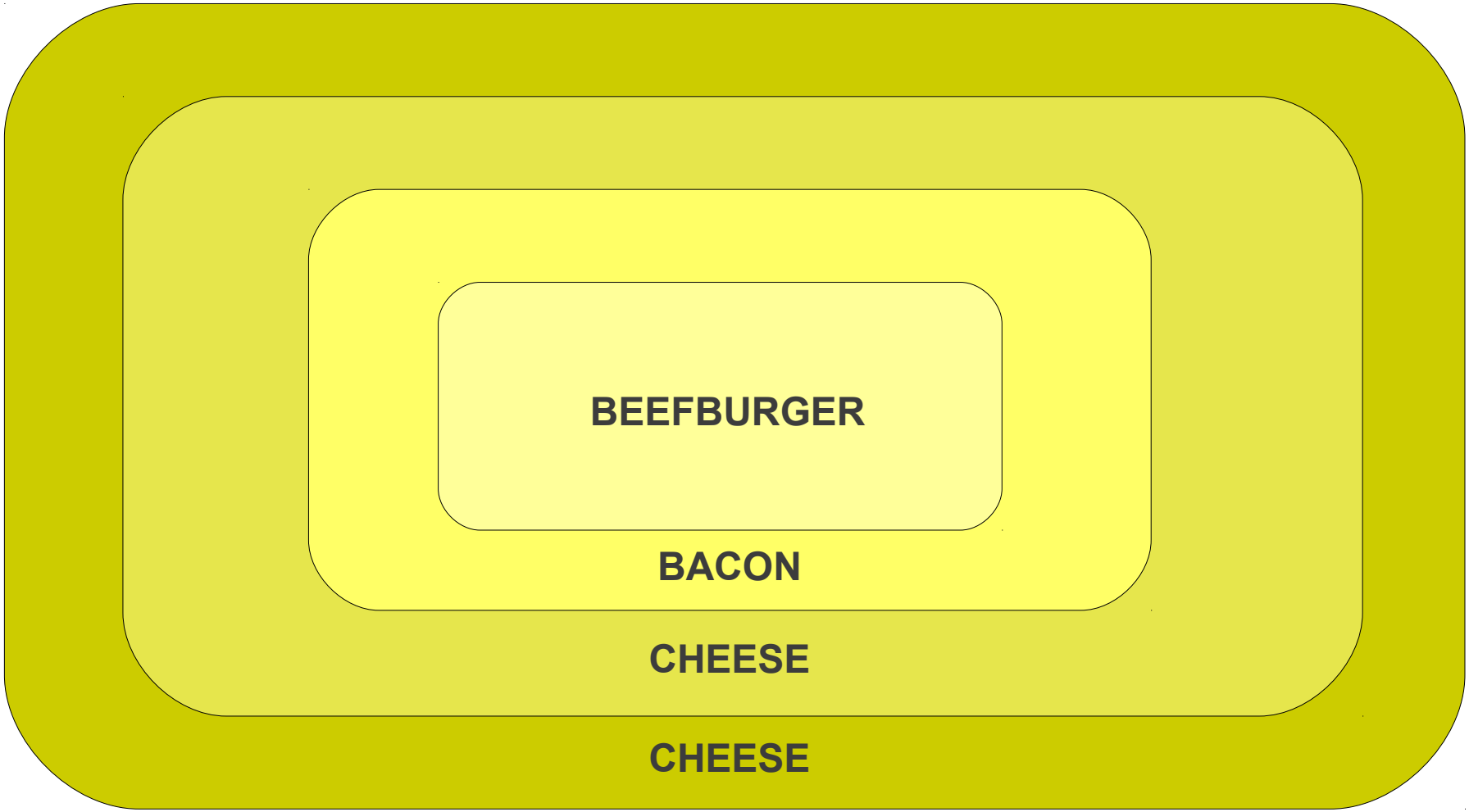
```
// Usage: Make a bacon double cheeseburger
Burger b = new BeefBurger();
b = new Bacon(b);
b = new Cheese(b);
b = new Cheese(b);

// Output price
System.out.println(b.getPrice());

// Output code
System.out.println(b.getCode());
```

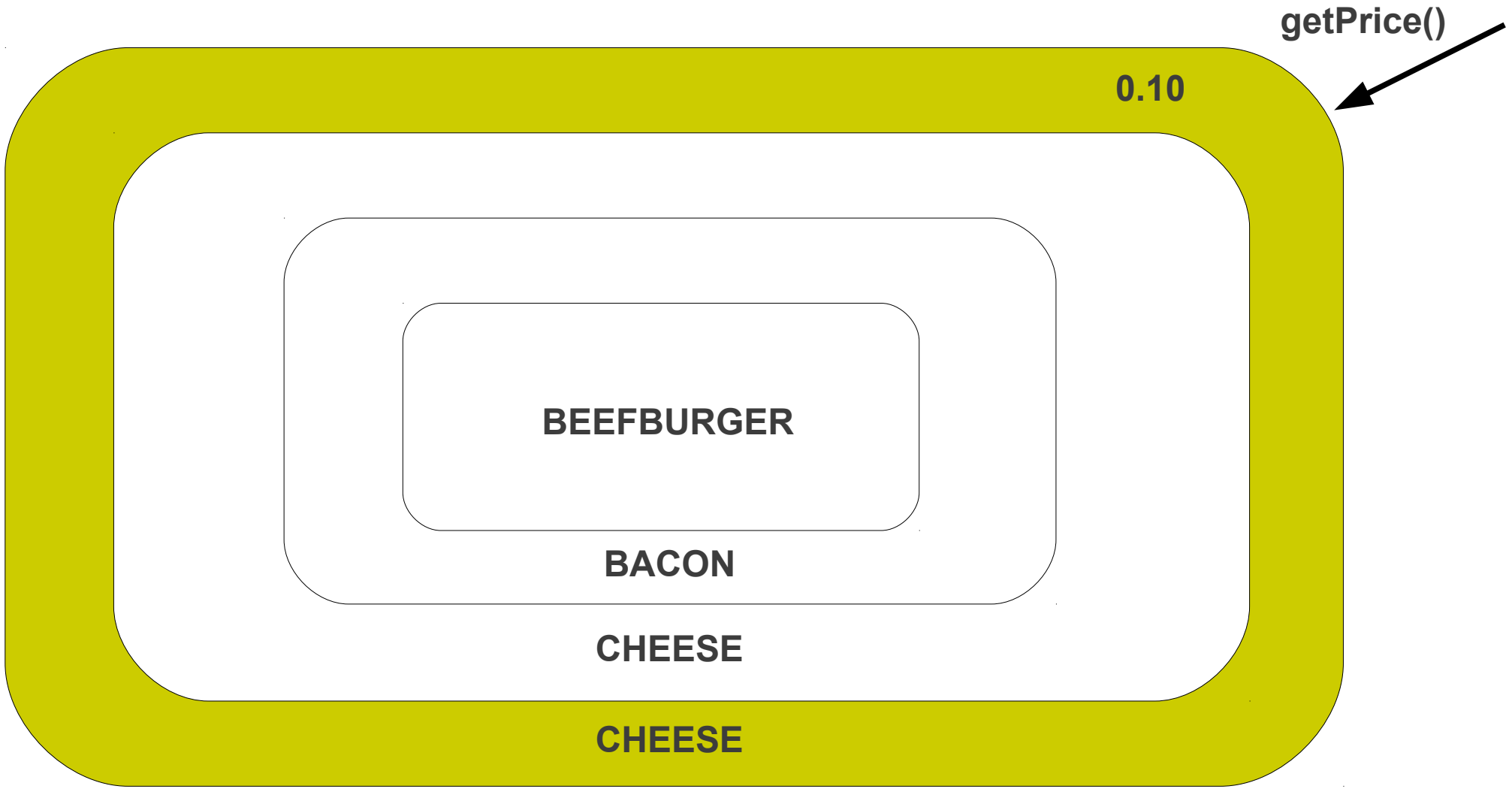
Order of operation when wrapping – Java evaluates right hand side first. So we pass a *BeefBurger* object (b) into the *Bacon* constructor to create a new wrapped object in memory. Then we assign that new object to the reference variable b. Simples.

The Decorator pattern



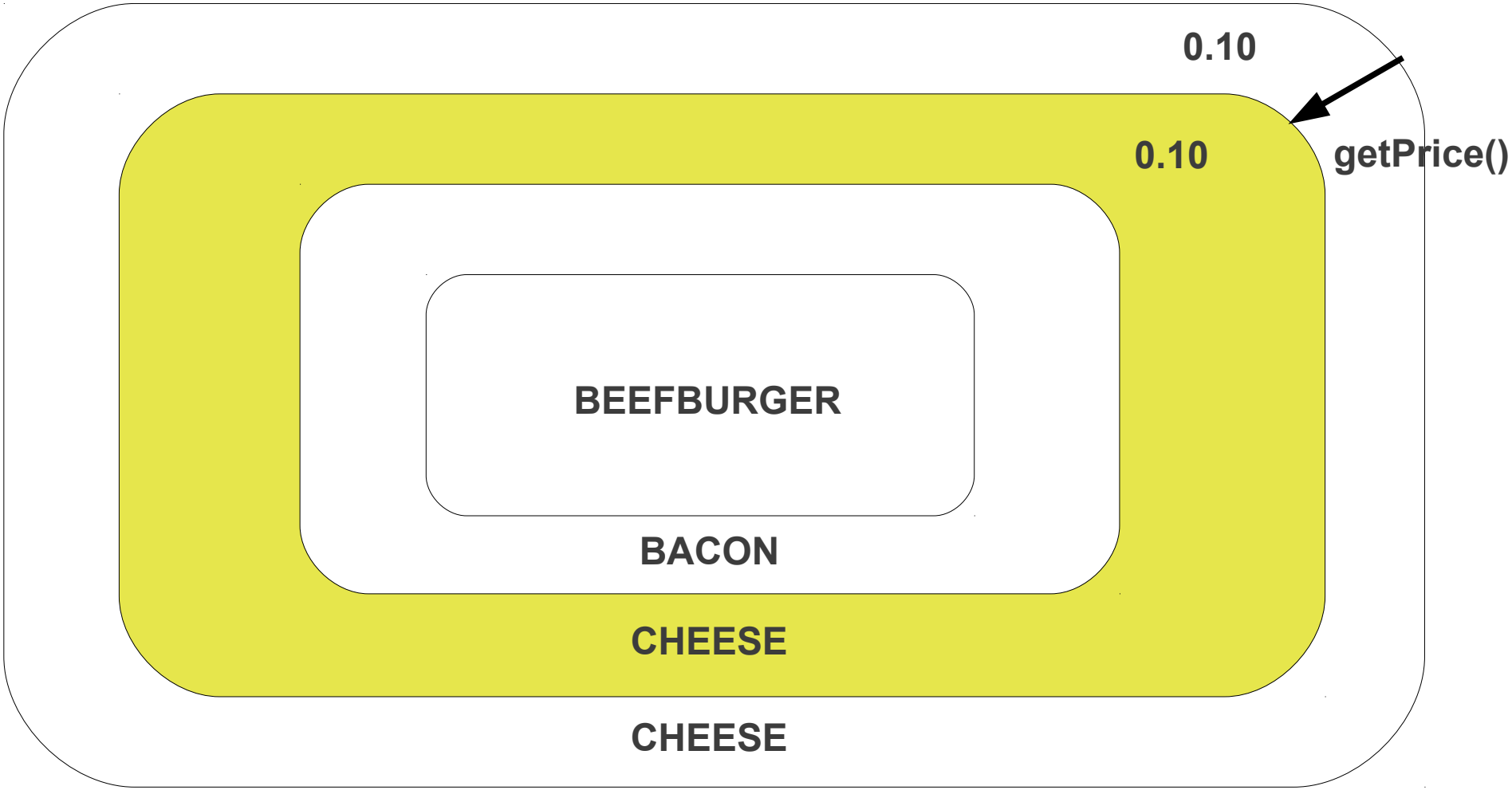
We have created a decorated object with 4 layers. The base object is a *BeefBurger*. That is wrapped by a *Bacon* object. That is wrapped by a *Cheese* object. That is wrapped by a another *Cheese* object.

The Decorator pattern



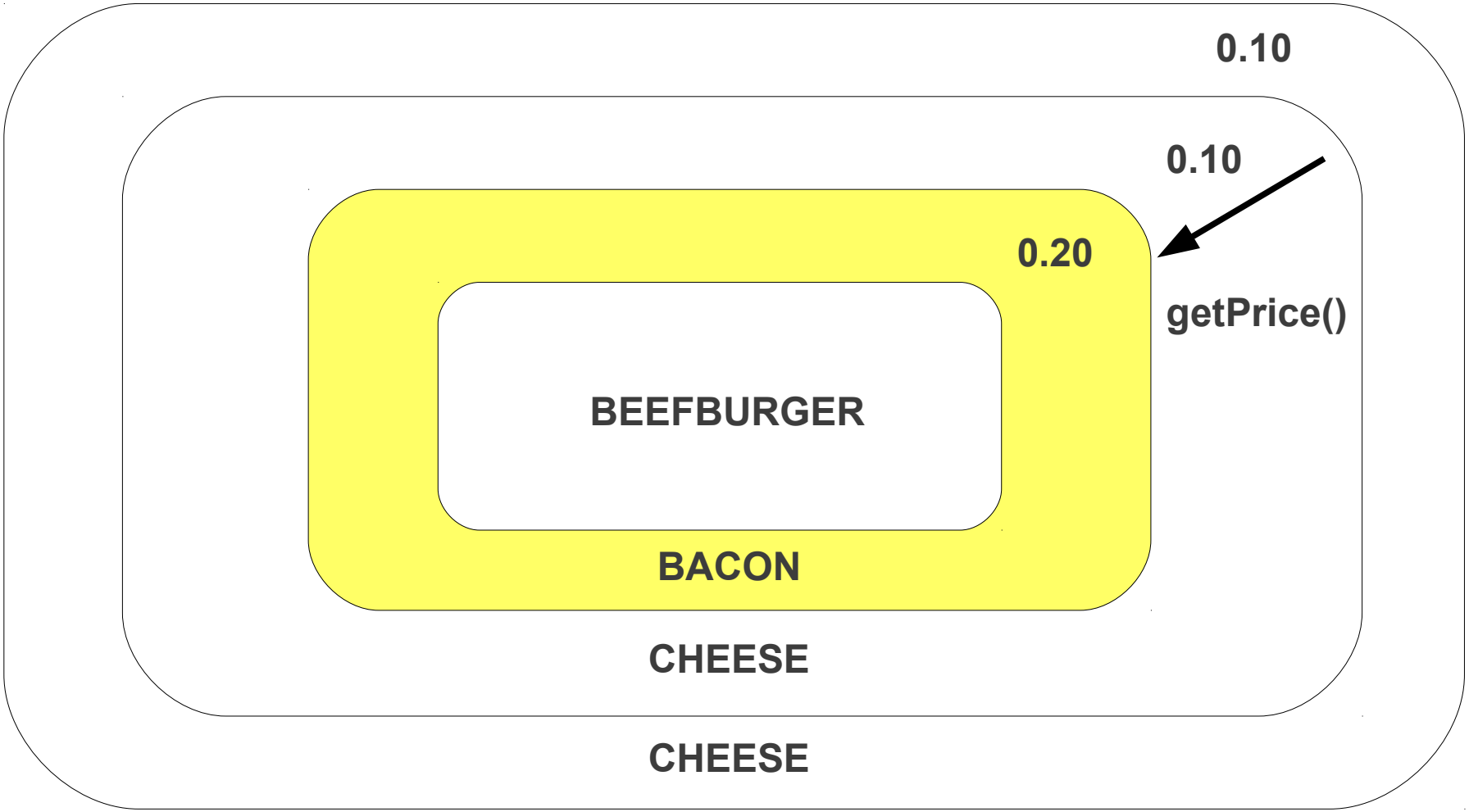
The *getPrice*() method of the outermost decorator *Cheese* object is called. This method says – my price is 0.10 plus the price of the object I wrap. To find that price, the *Cheese* object invokes the *getPrice*() of the wrapped object (another *Cheese*)

The Decorator pattern



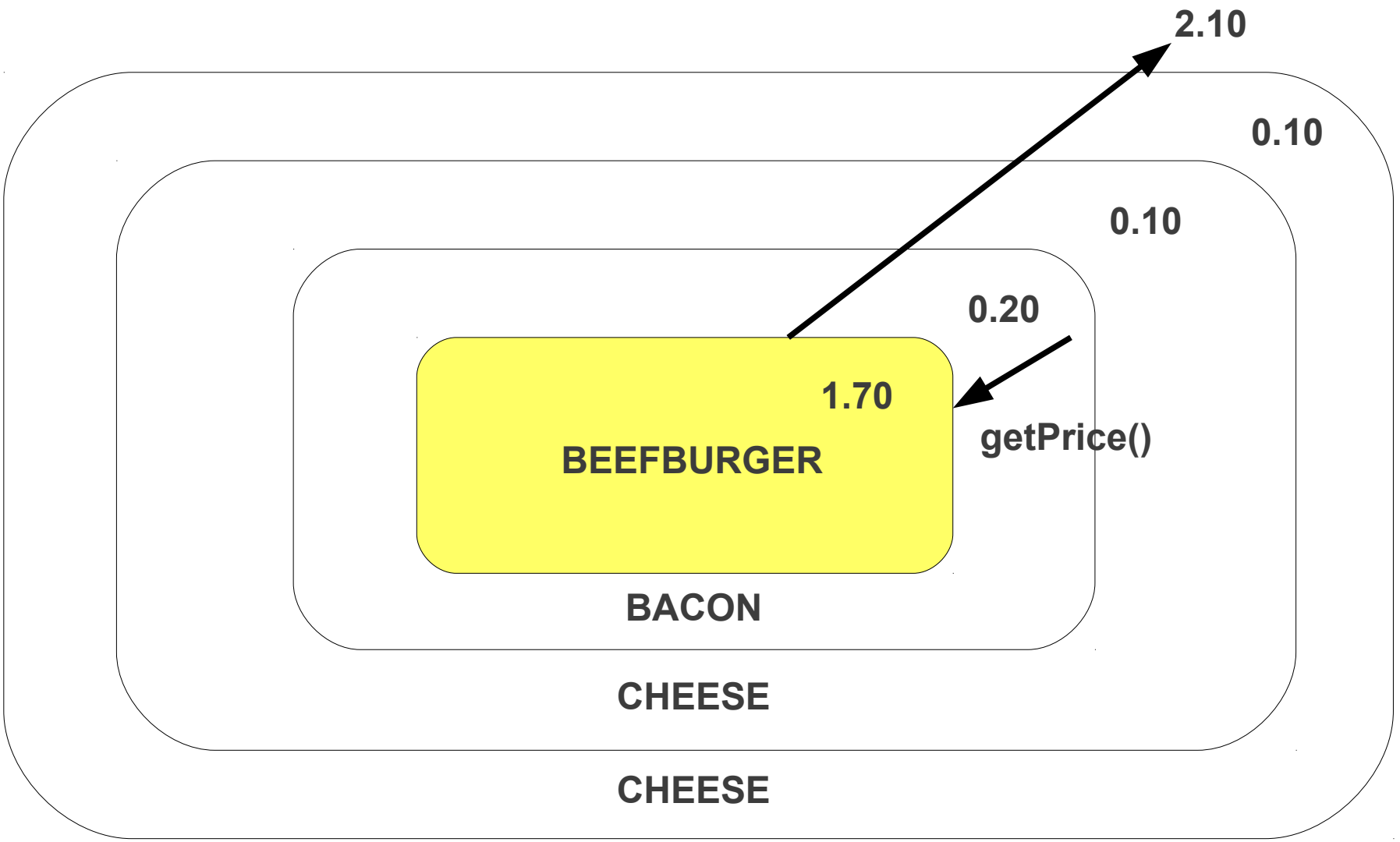
The *getPrice*() method of the next decorator is called. This method says – my price is 0.10 plus the price of the object I wrap. To find that price, the *Cheese* object invokes the *getPrice*() of the wrapped object (a *Bacon* object)

The Decorator pattern



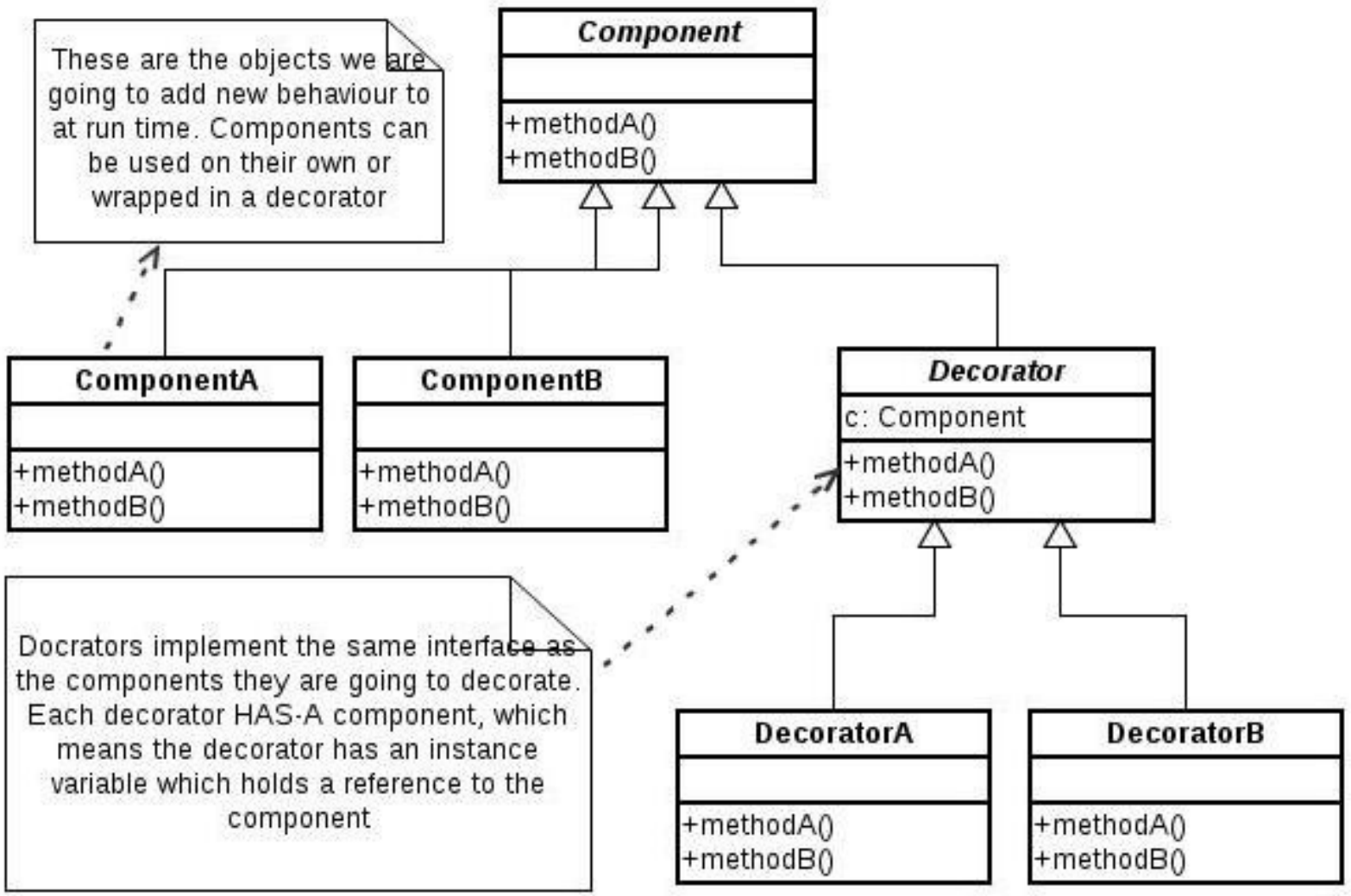
The *getPrice*() method of the next decorator is called. This method says – my price is 0.20 plus the price of the object I wrap. To find that price, the *Bacon* object invokes the *getPrice*() of the wrapped object (a *BeefBurger*)

The Decorator pattern



The *getPrice*() method of the innermost object is called. This method says – my price is 1.70. All 4 method calls complete, and the correct burger value (£2.10) is returned. Exactly the same approach would apply to the *getCode*() method

The Decorator pattern



UML diagram for the *Decorator* pattern. In our example, *Burger* is the abstract *Component*. *BeefBurger* and *ChickenBurger* are concrete components. Toppings are Decorators. The term interface is used in its loose sense

The Decorator pattern

- Advantages
  - Flexible alternative to subclassing
  - Components can be closed to modification
  - Decorators can be added and removed from components at run time (rather than design time)
- Disadvantage
  - Removing a nested decorator? Peeling an onion...
  - Large number of little classes
  - The *Decorator* must implement all methods of the wrapped class -- including methods that it is not decorating

The Decorator pattern

- Decorator IRL
  - This was a trivial example designed to demonstrate the pattern
  - In reality, the methods implemented by the *Decorators* can be quite complex
  - Java streams are just a giant collection of decorators decorating decorators that have decorated something that decorates. My face just melted

```
FileReader frdr = new FileReader(filename);
LineNumberReader lrdr = new LineNumberReader(frdr);
```

FileReader reads files containing characters. *LineNumberReader* wraps around this object to provide a buffered character-input stream that keeps track of line numbers. For more info and examples, see here.

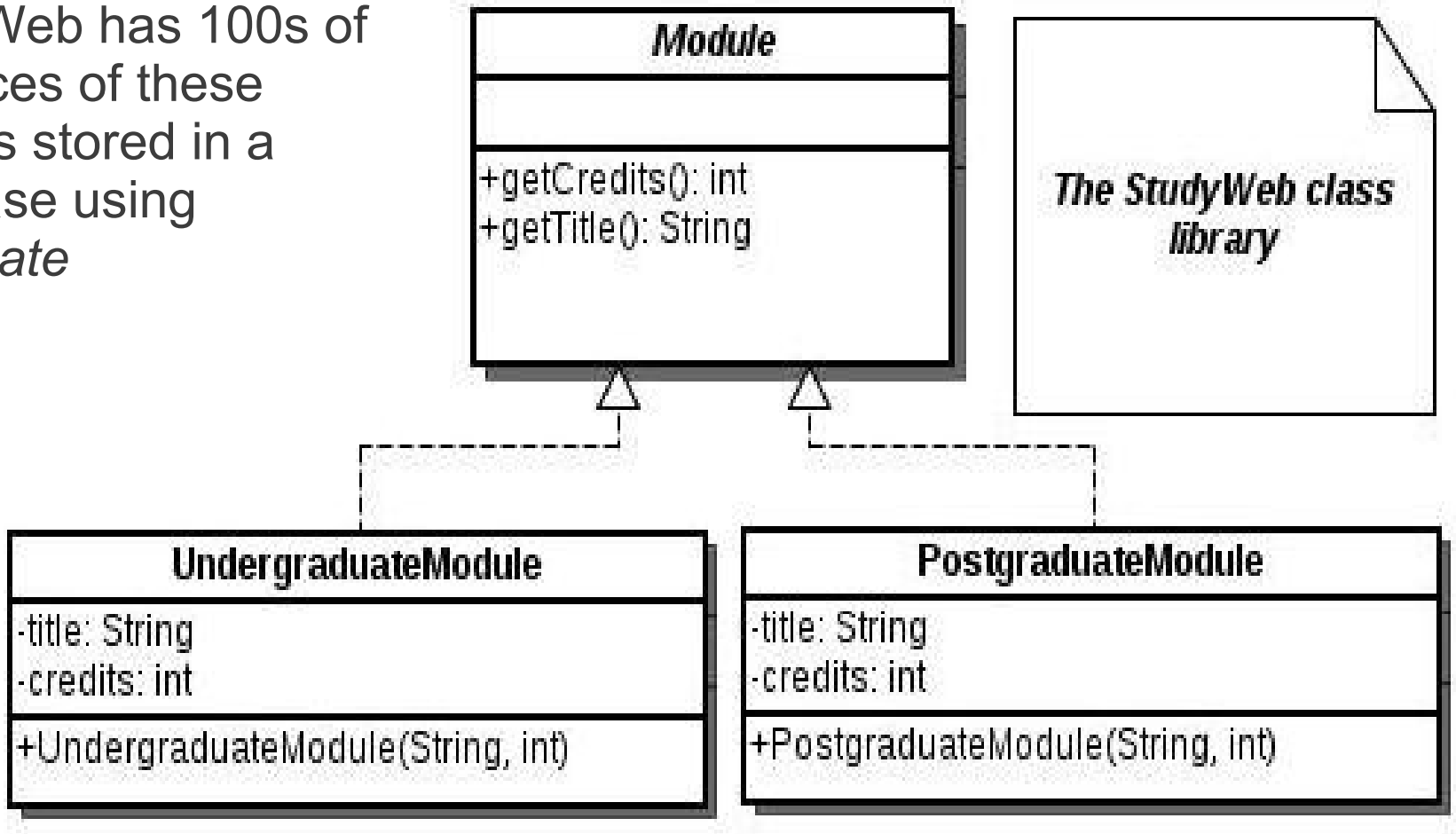

The Adapter Pattern

The Adapter Pattern

- Imagine that you work for a company that develops on-line learning materials (*StudyWeb*)
  - You have written classes that describe the various undergraduate / postgraduate modules you offer
- Your company collaborates with another eLearning company (*ScholarWeb*)
  - This company has also written classes that encapsulate their own stock of modules
- Although the class libraries are broadly similar, the method names and field names are different
- You have been given the job of producing an application that allows users to search *StudyWeb* and *ScholarWeb* modules
  - Your job - You need to make the libraries work together without changing to the underlying code

The Adapter Pattern

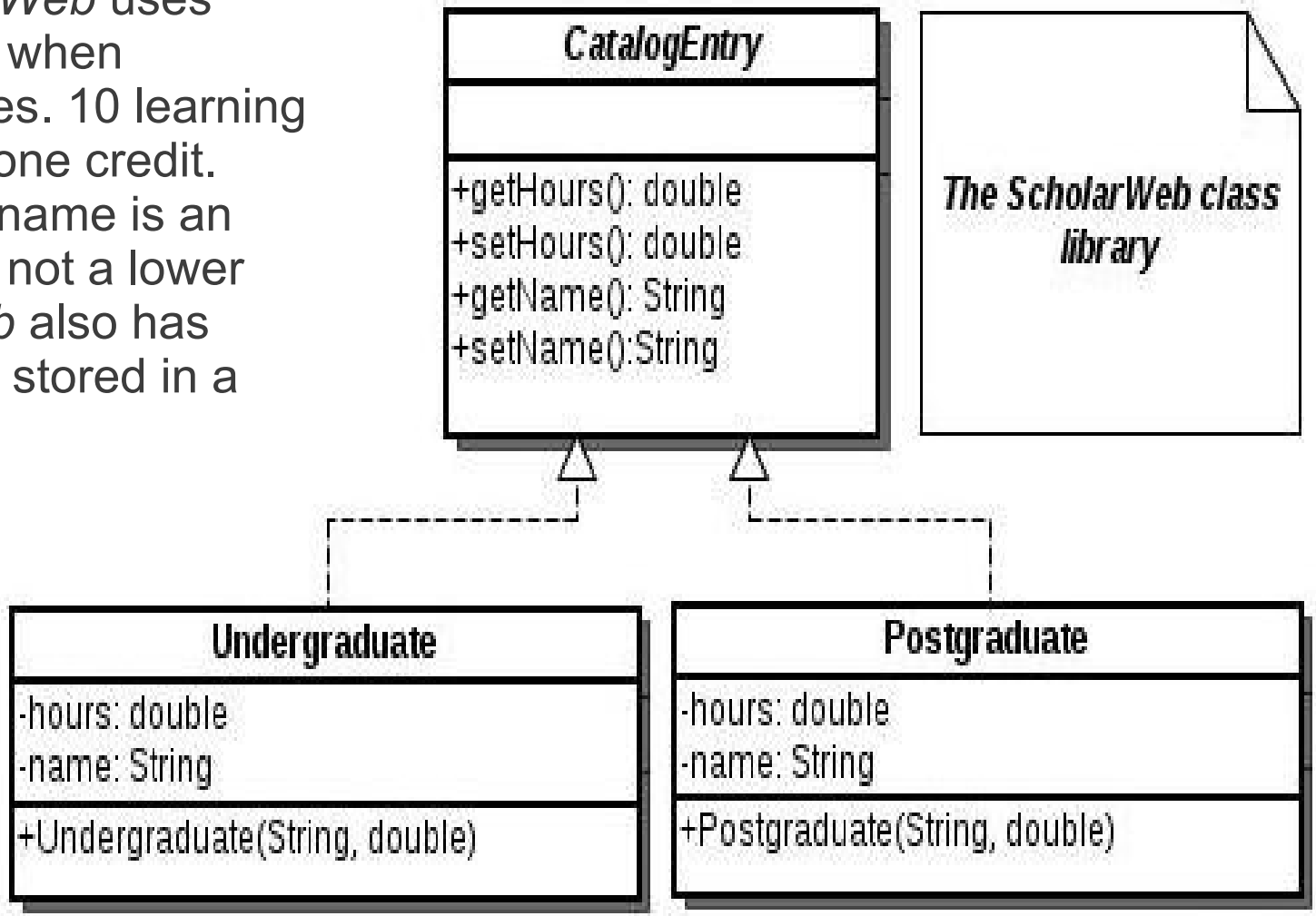
StudyWeb has 100s of instances of these classes stored in a database using *Hibernate*



Module m1 = new UndergraduateModule(“advanced java programming”, 20);

The Adapter Pattern

Note that *ScholarWeb* uses *hours*, not *credits* when describing modules. 10 learning hours is equal to one credit. Also, the module name is an uppercase string, not a lower case. *ScholarWeb* also has 100s of instances stored in a database



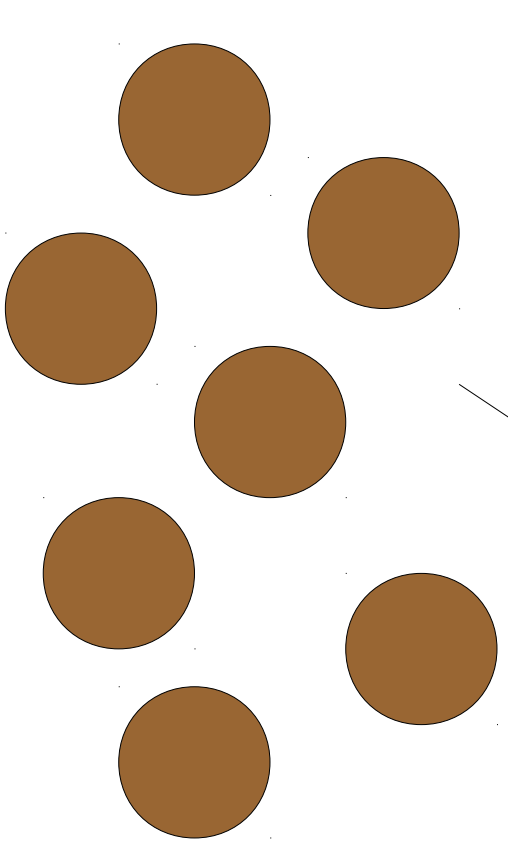

```
CatalogEntry c1 = new UnderGraduate("SCIENTIFIC METHOD", 200);
```

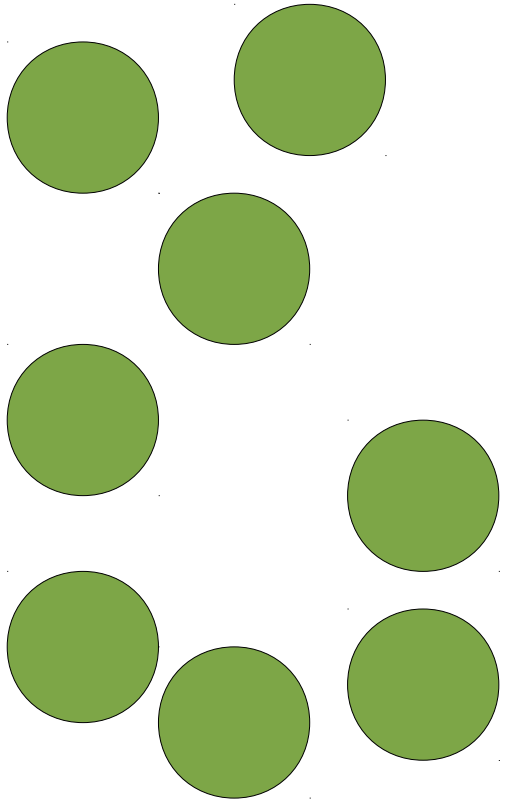
The Adapter Pattern

- So, we cannot change either class library
  - It might disturb other applications that rely on these class libraries
  - We might not even have access to all the source code – it is not unusual for companies to release a JAR file but no source code
  - It might take too long (imagine there are 200 classes)
- But we need to be able to use these two class libraries in our application in the easiest possible way
  - Ideally, we want to mix *ScholarWeb* objects in with *StudyWeb* objects and treat them as objects of the same type
  - We need an adapter class

The Adapter Pattern

We don't want to deal with two different type of class that basically describe the same thing. We need an adapter to convert them all to the same type





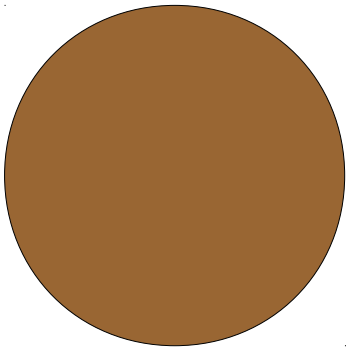
MODULE BROWSER

STUDYWEB OBJECTS

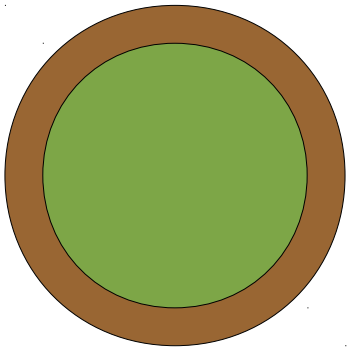
SCHOLARWEB OBJECTS

The Adapter Pattern

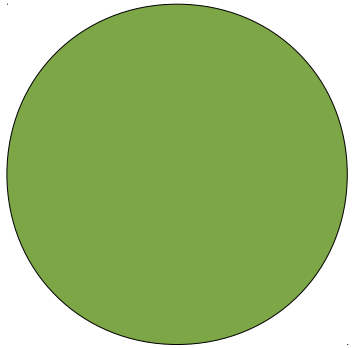
An adapter class has the *exterior* of one type of class, but inside it is another



STUDYWEB OBJECT



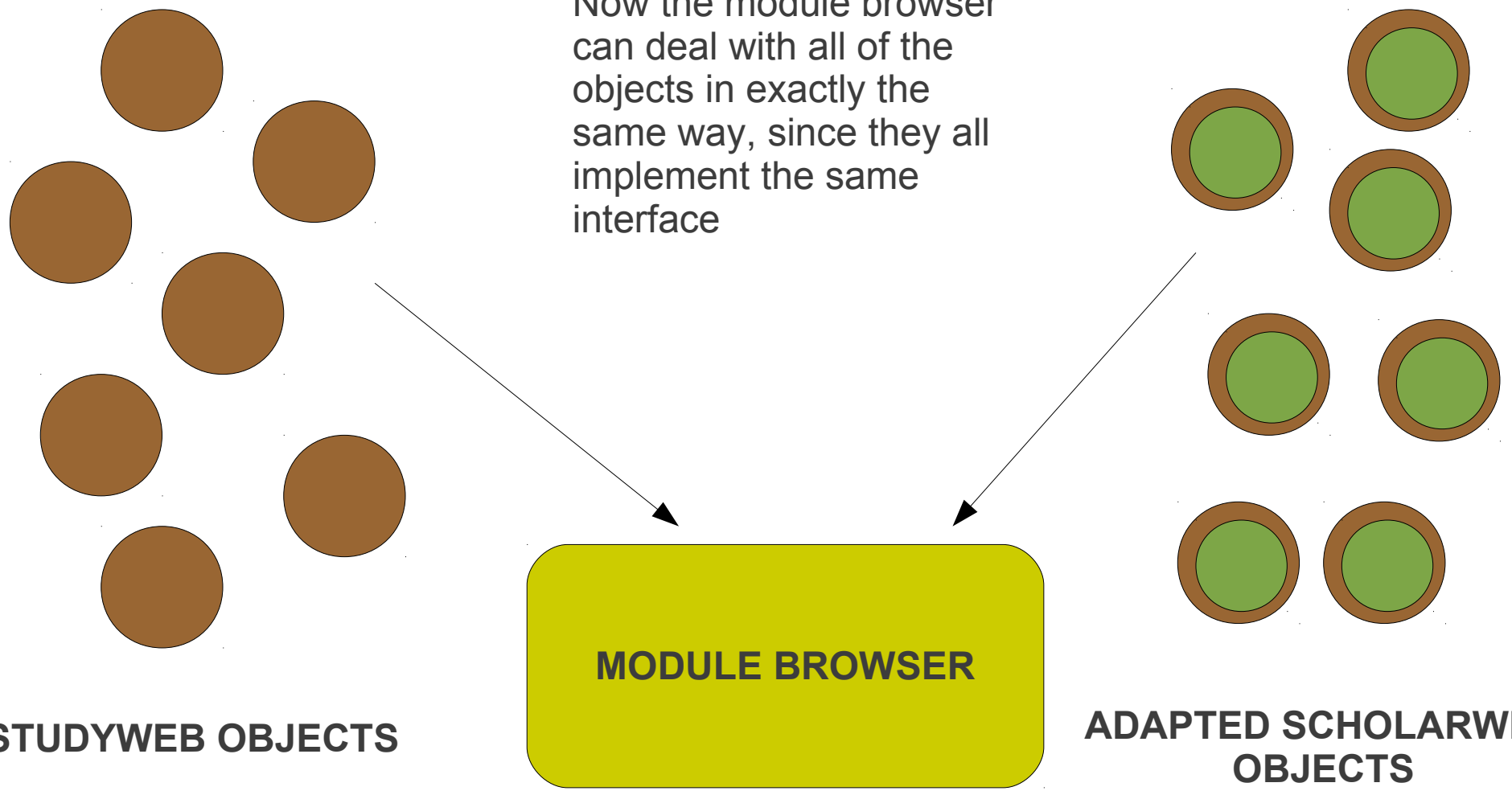
ADAPTER CLASS



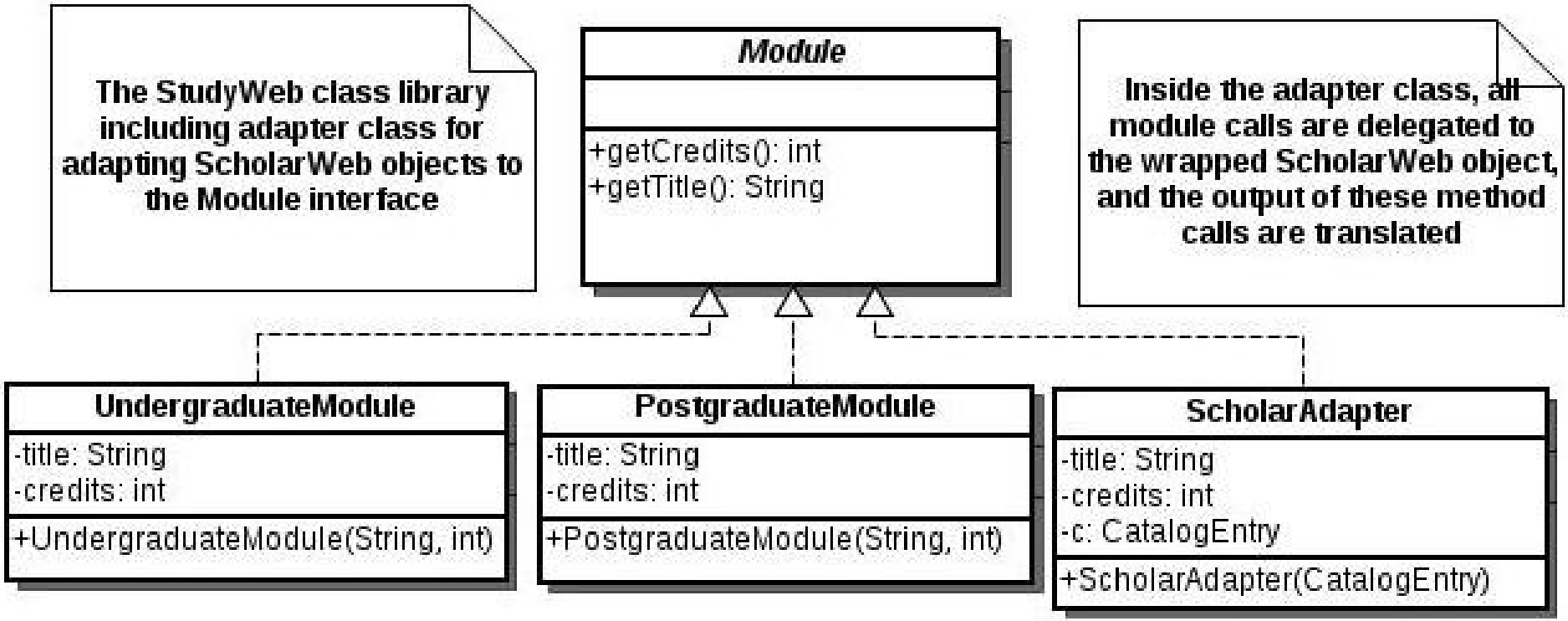
SCHOLARWEB OBJECT

The adapter class implements the target interface (in this case *Module* from the *StudyWeb* class library) but contains a *Scholarweb* object inside it.

The Adapter Pattern



The Adapter Pattern



Interface is used in the broadest sense i.e. an abstract type

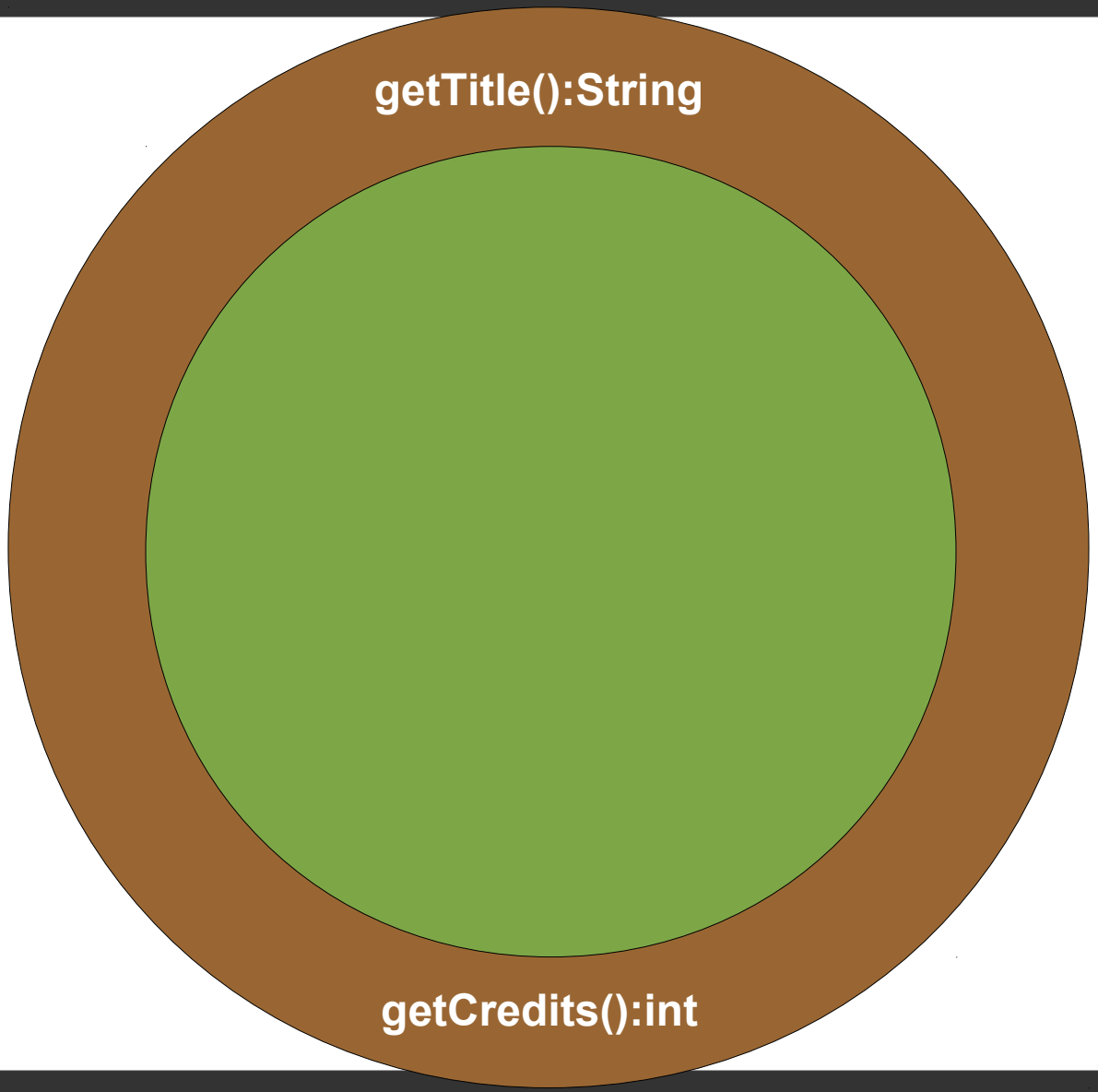
The Adapter Pattern

How does it work?

On the outside, the adapter class looks like a normal part of the *StudyWeb* class library. It implements the *Module* interface, so it has all the expected methods.

getCredits(): int
getTitle: String

These methods can be called as normal



Method call

The Adapter Pattern

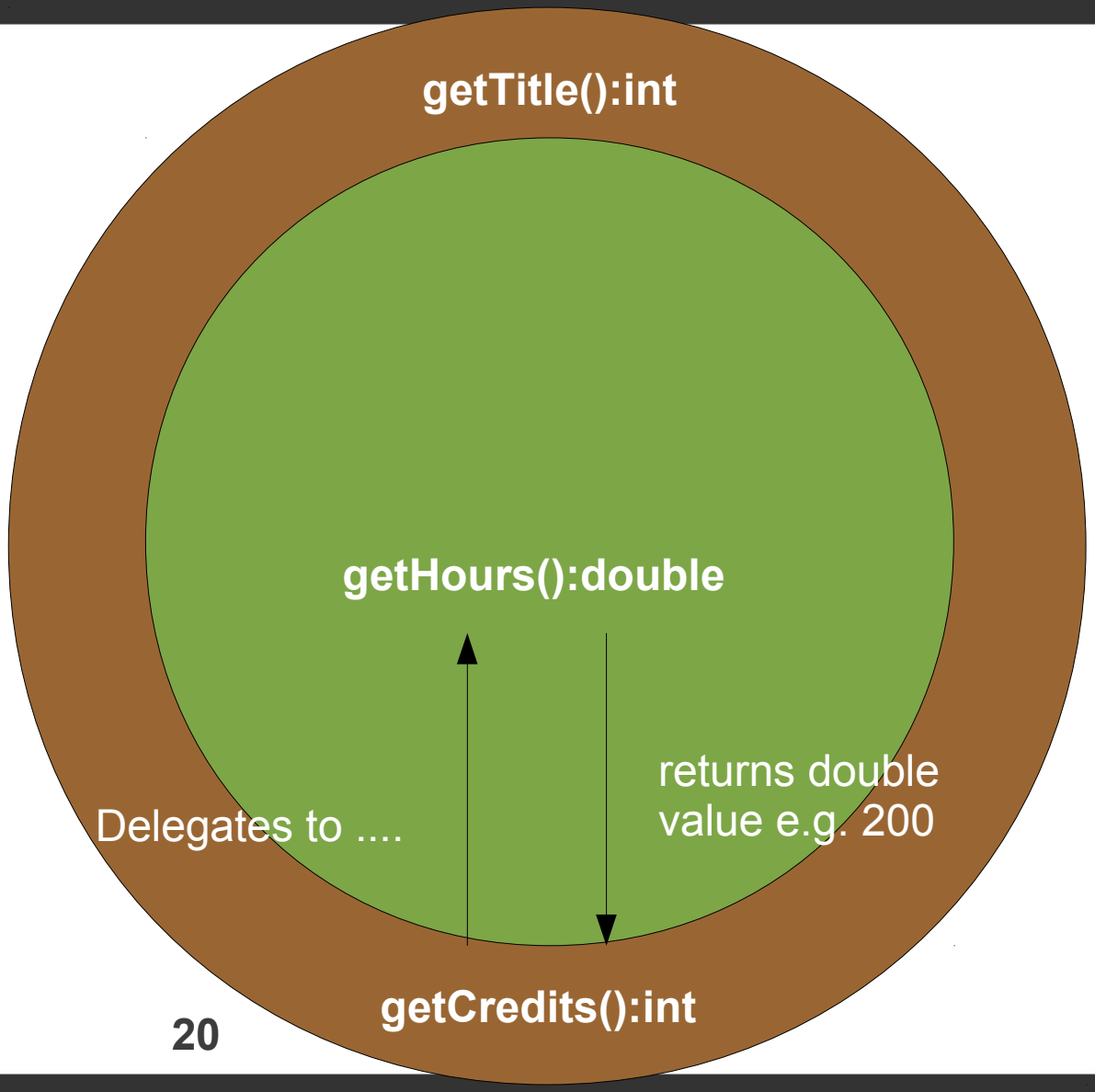
How does it work?

Let's assume that the *getCredits*() method is called on the adapter object

It then passes on (delegates) this call to the *getHours*() method of the wrapped object

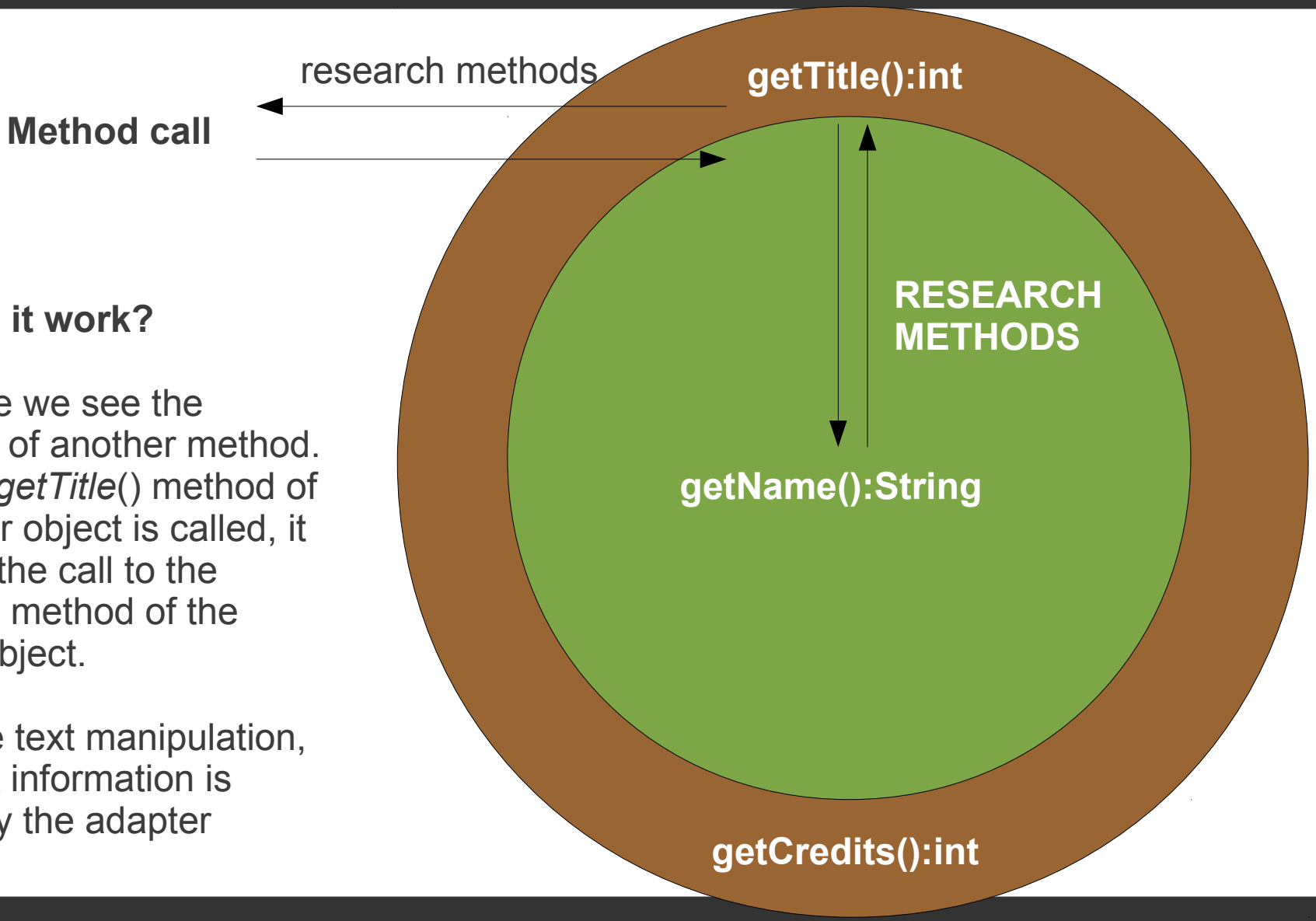
This method returns a double value e.g. 200 (hours)

The adapter object then converts this into the expected format. In this case, this involves dividing by 10 and converting to an int. This value is returned.



Method call

The Adapter Pattern



How does it work?

In this case we see the adaptation of another method. When the *getTitle*() method of the adapter object is called, it delegates the call to the *getName*() method of the wrapped object.

After some text manipulation, the correct information is returned by the adapter

The Adapter Pattern

```
public class ScholarWebAdapter implements Module
{
   private CatalogEntry c;
    public ScholarWebAdapter(CatalogEntry c)
    {
        this.c = c;
    }

    public String getTitle()
    {
        return c.getName().toLowerCase();
    }

    public int getCredits()
    {
        return (int) c.getHours() / 10;
    }
}
```

The wrapped object

Delegation of method call to wrapped object and conversion of return values to fit interface

The Adapter Pattern

```
// Create some StudyWeb objects
Module m1 = new UndergraduateModule("advanced java programming", 20);
Module m2 = new UndergraduateModule("programming for mobile devices",
20);

// Create some ScholarWeb objects
CatalogEntry c1 = new PostGraduate("MINDFULNESS", 100);
CatalogEntry c2 = new UnderGraduate("SCIENTIFIC METHOD", 200);

// Create adapter objects
Module m3 = new ScholarWebAdapter(c1);
Module m4 = new ScholarWebAdapter(c2);

// Mix them all up in a collection, treat them identically
ArrayList<Module> al = new ArrayList<Module>();
al.add(m1); al.add(m2); al.add(m3); al.add(m4);
```

The Adapter Pattern

- When you need to use an existing class and its interface is not the one you need, use the adapter pattern
- An adapter changes an interface into one a client expects
  - In this case, the client was our module browser
- Implementing an adapter may require little work or a great deal of work depending on the size and complexity of the target interface
  - In our example, providing the 'glue' only needed a few lines of code – in real life things are not always so straightforward
- Remember - An adapter wraps an object to change its interface, a decorator wraps an object to add new behaviors and responsibilities

Summary

- **The Decorator pattern**
  - A structural pattern which can add additional functionality to a *particular object* as opposed to a class of objects
  - It is easy to add functionality to an entire class of objects by subclassing an object, but it is impossible to extend a single object this way
  - With the *Decorator* Pattern, you can add functionality to a single object and leave others like it unmodified
- **The Adapter pattern**
  - Another structural design pattern that translates one interface for a class into a compatible interface
- **Pre-reading**
  - Factory pattern in HFDP and DPFD
  - Read up on dependency inversion