Chain of responsibility + Template

Advanced Java Programming 2012-13

OO Design Patterns and Principles

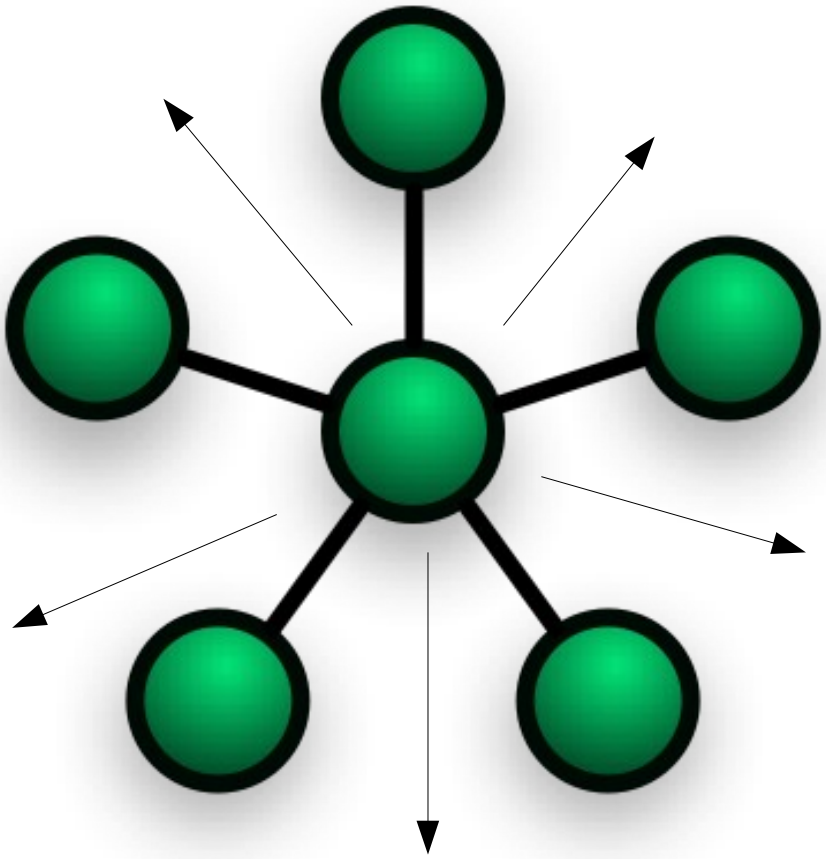
Lecture Outline

- Chain of responsibility design pattern
- Template method design pattern

Chain of Responsibility pattern

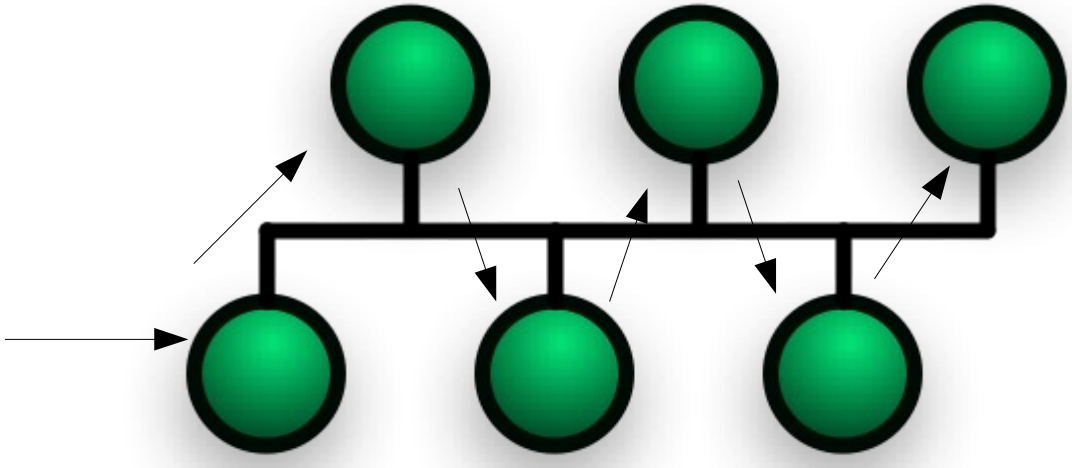
- Last week we covered the *Observer* pattern
  - Observers were notified about changes in the state of a subject
- This week we move on to another pattern that involves notification
  - This time the notification (message) is passed to a linear *sequence* of objects
  - The first object that is qualified to handle the notification 'deals with it'
- There is usually some aspect of escalation within this pattern
  - The chain of objects usually form a hierarchy
  - A real world example: Your problems could be escalated from Module Leader -> Program Director -> Assistant Dean -> Dean

Chain of Responsibility pattern



Name: Observer pattern
Type: Behavioural (communication between objects)
Definition: Defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically

Chain of Responsibility pattern

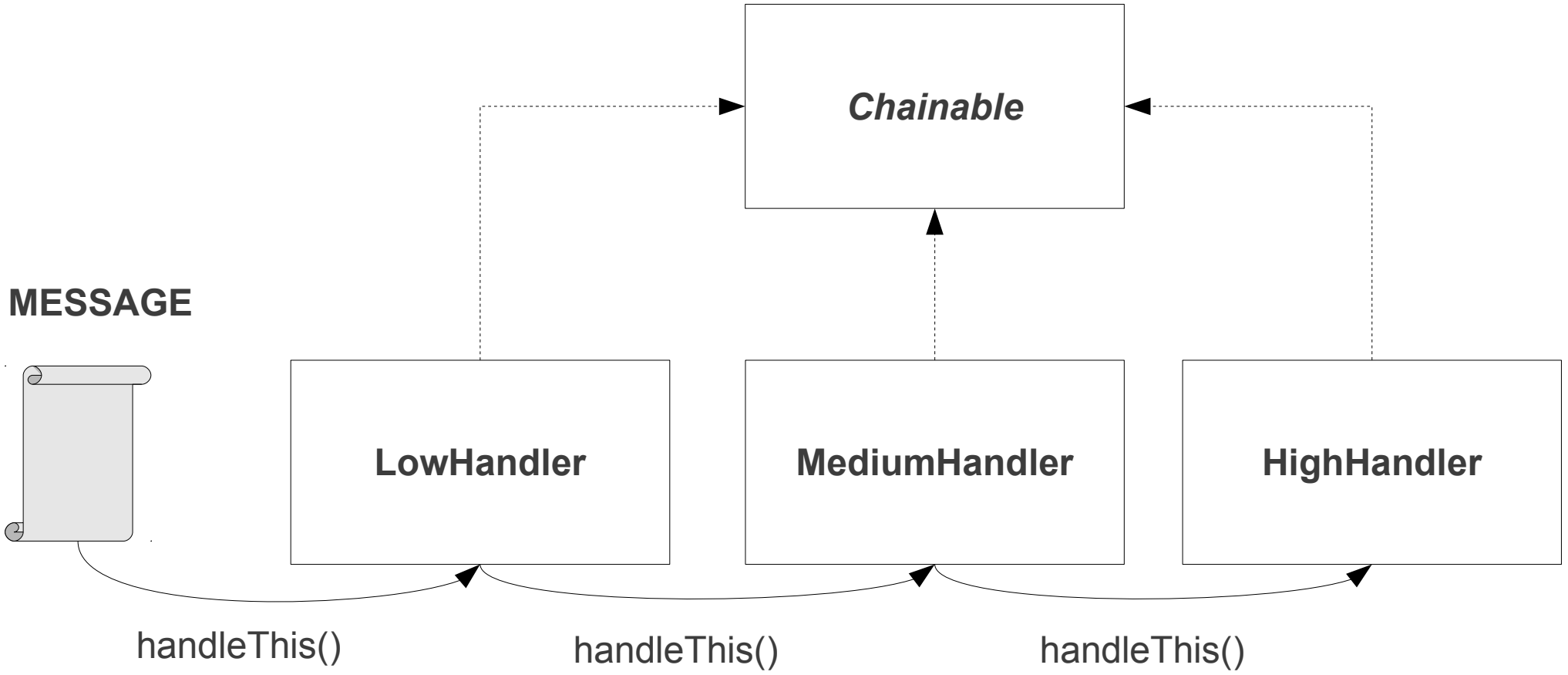


Name: Chain of Responsibility
Type: Behavioural
Definition: Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

Chain of Responsibility pattern

- **How do I use the pattern?**
  - We start by defining an interface, that will be implemented by all elements in our chain
  - Let's call the interface *Chainable*
- This interface must define an abstract method which is called when a notification is passed on
  - We will call the method *handleThis(int level)*
- Notifications have different levels of importance - low (1), medium level (2) and high level (3)
- Let's assume there are three types of objects in the chain
  - They are objects of type *LowHandler*, *MediumHandler*, and *HighHandler* respectively
- We want the notification handled by the correct handler

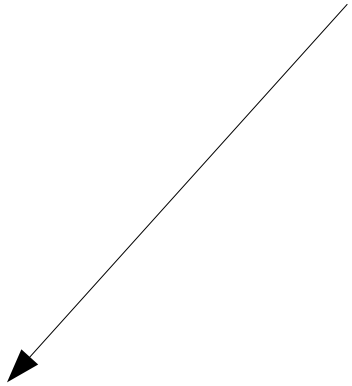
Chain of Responsibility pattern



Chain of Responsibility pattern

This method is called when a notification is passed to a handler in the chain. Notifications can be low level (1), medium level (2) or high level (3)

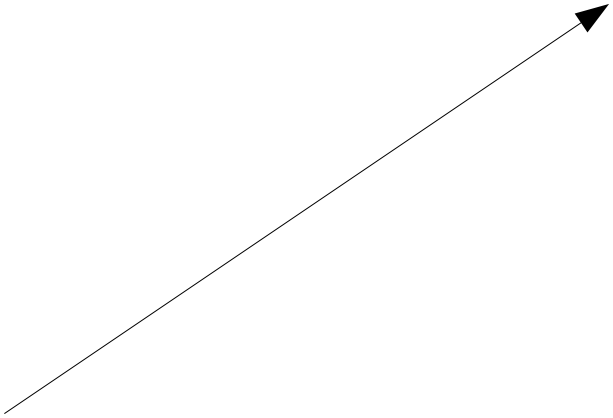
```
a.processNotification(1);  // Object a is passed a low level message
```



```
interface Chainable
{
    public void processNotification(int level);

    public void setNextElement(Chainable next);

}
```

Chain of Responsibility pattern

```
interface Chainable
{
    public void processNotification(int level);

    public void setNextElement(Chainable next);

}
```



This method is used to chain handlers together

```
a.setNextElement(b);  // Object a is chained to object b
```

Chain of Responsibility pattern

```
class LowHandler implements Chainable
{
    Chainable next;
    final int level = 1;

    public void handleThis(int level)
    {
        if (this.level == level)
        {
            System.out.println("LH: I suppose I can do this.");
        } else
        {
            next.handleThis(level);   // Or pass it on….
        }
    }

    public void setNextElement(Chainable next)
    {
        this.next=next;
    }
}
```

Similar to a linked list. Each *Chainable* object holds a reference to the next element in the chain.

Here is where the decision gets made. Can I handle this?

The method that adds a new link in the chain

Chain of Responsibility pattern

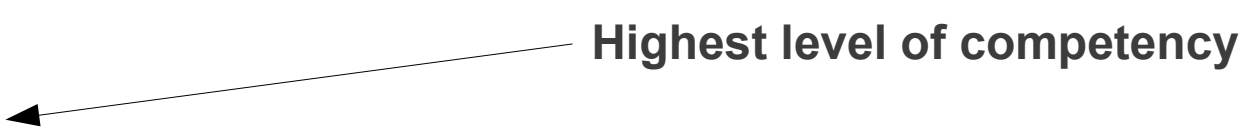
```
class MediumHandler implements Chainable
{
    Chainable next;
    final int level = 2;

    public void handleThis(int level)
    {
        if (this.level == level)
        {
            System.out.println("MH: I have got this one.");
        } else
        {
            next.handleThis(level);   // Or pass it on….
        }
    }

    public void setNextElement(Chainable next)
    {
        this.next=next;
    }
}
```

Higher level of competency / responsibility

Different handling code. Handlers in the chain will take different actions.

Chain of Responsibility pattern

```
class HighHandler implements Chainable
{
    Chainable next;
    final int level = 3;

    public void handleThis(int level)
    {
        if (this.level == level)
        {
            System.out.println("HH: This one is all mine.");
        } else
        {
            next.handleThis(level);   // Or pass it on….
        }
    }

    public void setNextElement(Chainable next)
    {
        this.next=next;
    }
}
```



Chain of Responsibility pattern

```
class TestCOR
{

    public static void main(String[] args)
    {
        // Instanciate the elements
        LowHandler lh = new LowHandler();
        MediumHandler lh = new MediumHandler();
        HighHandler lh = new HighHandler();

        // Chain them together
        lh.setNextElement(mh);
        mh.setNextElement(hh);

        // Pass the first element in the chain a message
        lh.handleThis(2);
    }

}
```

Output will be

```
MH: I have got this one
```

Chain of Responsibility pattern

- **Refinements**
  - Implement *Chainable* as an abstract class
  - Let's call this *ChainElement*
  - Remove instance specific code into a separate method, called *handle()*
  - All chain elements extend *ChainElement*, implement *handle()*
- Note that interfaces and abstract classes can often be used interchangeably on most design patterns
  - The choice is sometimes a matter of taste

Chain of Responsibility pattern

```
abstract class ChainElement
{
     protected Chainable next;
     protected int level

     public ChainElement(int level)
     { this.level = level; }

     public void handleThis(int level)
     {
          if (this.level == level)
          {
               handle();
          }
          else
          {
               next.handleThis(level);
          }
     }

     public void setNextElement(Chainable next)
     {
          this.next=next;
     }

     abstract void handle();
}
```

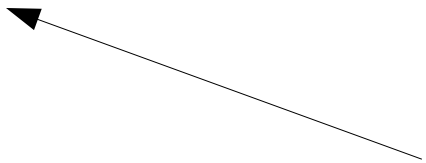
Chain of Responsibility pattern

```
class LowHandler extends ChainElement
{
   public LowHandler()
   {
      super(1);  // initialise handler
   }

   // Instance specific method
   public void handle()
   {
      System.out.println("LH: I suppose I can do this.");
   }

}
```

Chain of Responsibility pattern

- **Refinements**
  - Limit the range of possible value for the notification using an *enum*
  - An *enum* type is a type whose fields consist of a fixed fixed number of well known values
  - Using an *enum* forces a compile time check on the values passed to the *handleThis()* method

```
enum ImportanceLevel {
     LOW, MEDIUM, HIGH
}
```

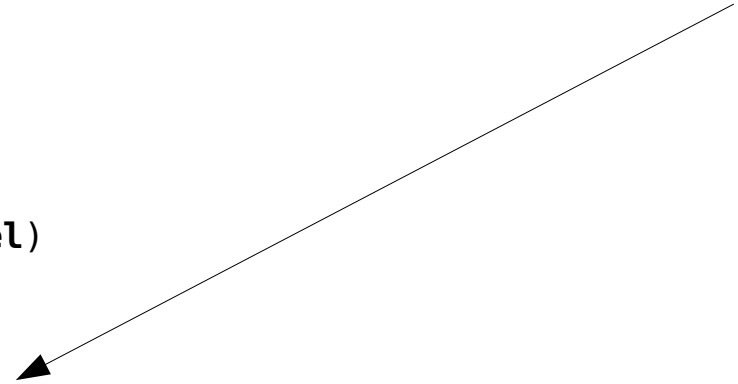


Here we define 3 constants. LOW (0), MEDIUM(1), HIGH(2). They are used like this `ImportanceLevel.HIGH`

Chain of Responsibility pattern

```
abstract class ChainElement
{
     protected Chainable next;
     protected ImportanceLevel level;

     public ChainElement(ImportanceLevel level)
     { this.level = level; }

     public void handleThis(ImportanceLevel level)
     {
          if (this.level == level)
          {
               handle();
          }
          else
          {
               next.handleThis(level);
          }
     }

     public void setNextElement(Chainable next)
     {
          this.next=next;
     }

     abstract void handle();
}
```

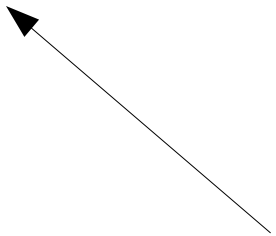
a.handleThis(ImportanceLevel.HIGH);



Chain of Responsibility pattern

- **Refinements**
  - Rather than passing around integers or simple *enum*, pass around an object (more information)

```
class StudentProblem
{
   private int severity;
   private String description;
}
```

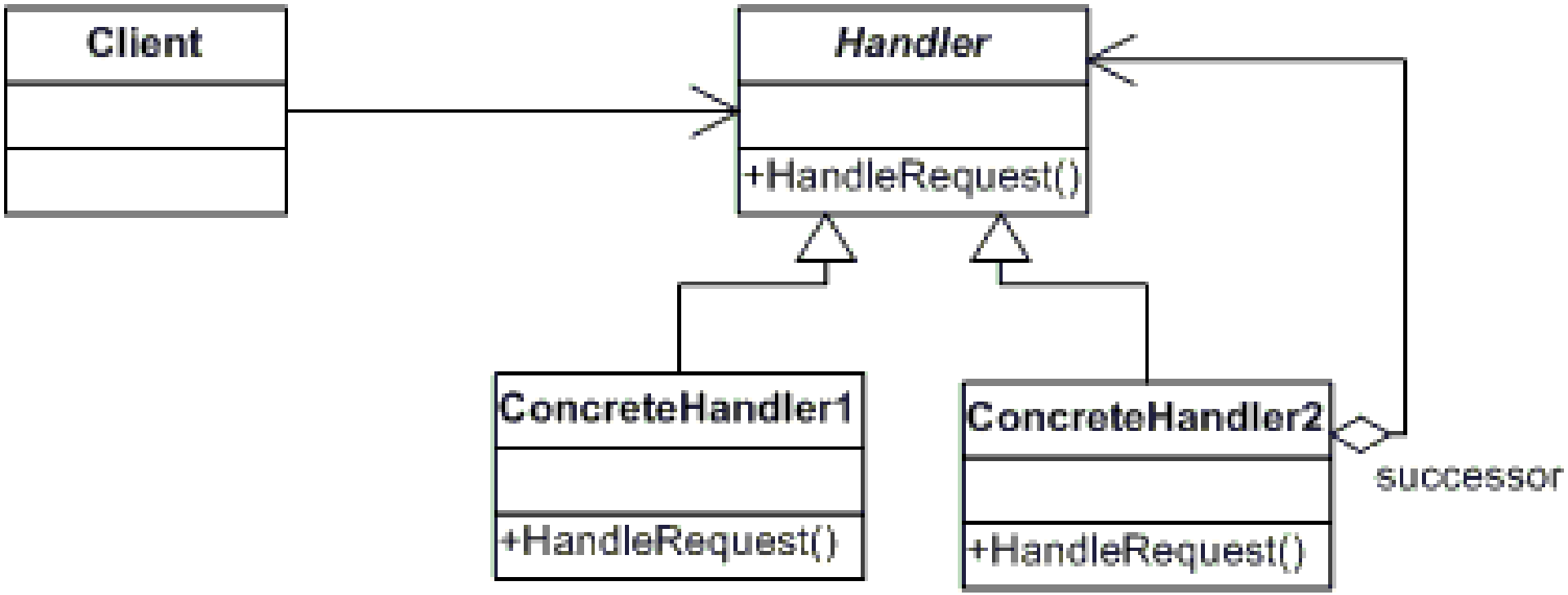


Message handlers could look at severity level and keywords inside free text description e.g. If description contains the term ‘disability' pass to Carol Mooney.

Chain of Responsibility pattern

- **Refinements**
- Jumps in the chain
  - Use where certain predictable escalations are known
  - If text description contains ‘violence’, escalate to Dean
- Branches in the chain
  - If financial problem, route to accounting chain
  - If attendance problem, route to student retention officer
- Simplify construction of chain
  - Always add new chain elements to first element in chain
  - The new chain element is passed *along* the chain until it finds the right place (which is usually the end)

Chain of Responsibility pattern



Chain of Responsibility UML diagram

Diamond indicates part-of relationship i.e. a *Handler* object is part-of a concrete handler. This makes sense. All concrete handlers hold a reference to the next handler in the chain.

The Template Method Pattern

Template Method pattern

- Sometimes you want to control the *order* of operations that a method uses, but allow subclasses to provide their own implementations of some of these operations
  - In this case, you need the *template method* pattern
- The template method pattern allows you to
  - Define the skeleton of an algorithm in an operation, deferring some steps to subclasses
  - Let subclasses redefine certain steps of an algorithm without changing the algorithm's structure

Template Method pattern

- Sounds a bit *abstract*, let's look at an example
  - The simplest implementation of the template method has a superclass, and two sub-classes
  - This superclass contains a method (called the *template method*) which defines a number of steps i.e. an algorithm for doing something
  - These steps can be implemented within the template method, or (more likely) as separate methods
  - At least one of these steps involves a call to an abstract method
  - The sub-classes implement the abstract method

Template Method pattern

This is a Sheperd’s Pie. It is made from lamb mince, vegetables and herbs, topped with potato

Template Method pattern

This is a Cottage Pie. It is made from beef mince, vegetables and herbs, topped with potato. Very similar

Template Method pattern

- The high level algorithm for creating both pies is almost identical
  - Prepare meat
  - Prepare potatoes
  - Assemble in dish
  - Bake
- The only invariance (difference between the two pies) is the *type* of meat used
  - Let’s turn this into code

Template Method pattern

```
class PieMaker
    final public void makePie()
    {
        prepareMeat();
        preparePotatoes();
        assembleInDish();
        bake();
    }
    public final private void preparePotatoes()
    {
        System.out.println(“Boiling and mashing the spuds”);
    }
    public final void assembleInDish()
    {
        System.out.println(“Piping the mash on to the meat”);
    }
    public final void bake()
    {
        System.out.println(“Baking in a hot oven”);
    }
    public abstract prepareMeat();
}
```

Notice that the template method is **final**, so it cannot be overriden. Also notice that the prepareMeat() is **abstract**

Template Method pattern

```
class CottagePieMaker extends PieMaker
{
   // Inherits the template method, implements
  // abstract method

   public void prepareMeat()
   {
      System.out.println(“Frying the beef”);

   }
}
```

Template Method pattern

```
class SheperdsPieMaker extends PieMaker
{

    public void prepareMeat()
    {
        System.out.println("Browning off the lamb");

    }
}
```

```
PieMaker pm1 = new CottagePieMaker();
PieMaker pm2 = new SheperdsPieMaker();
pm1.makePie();
pm2.makePie();
```

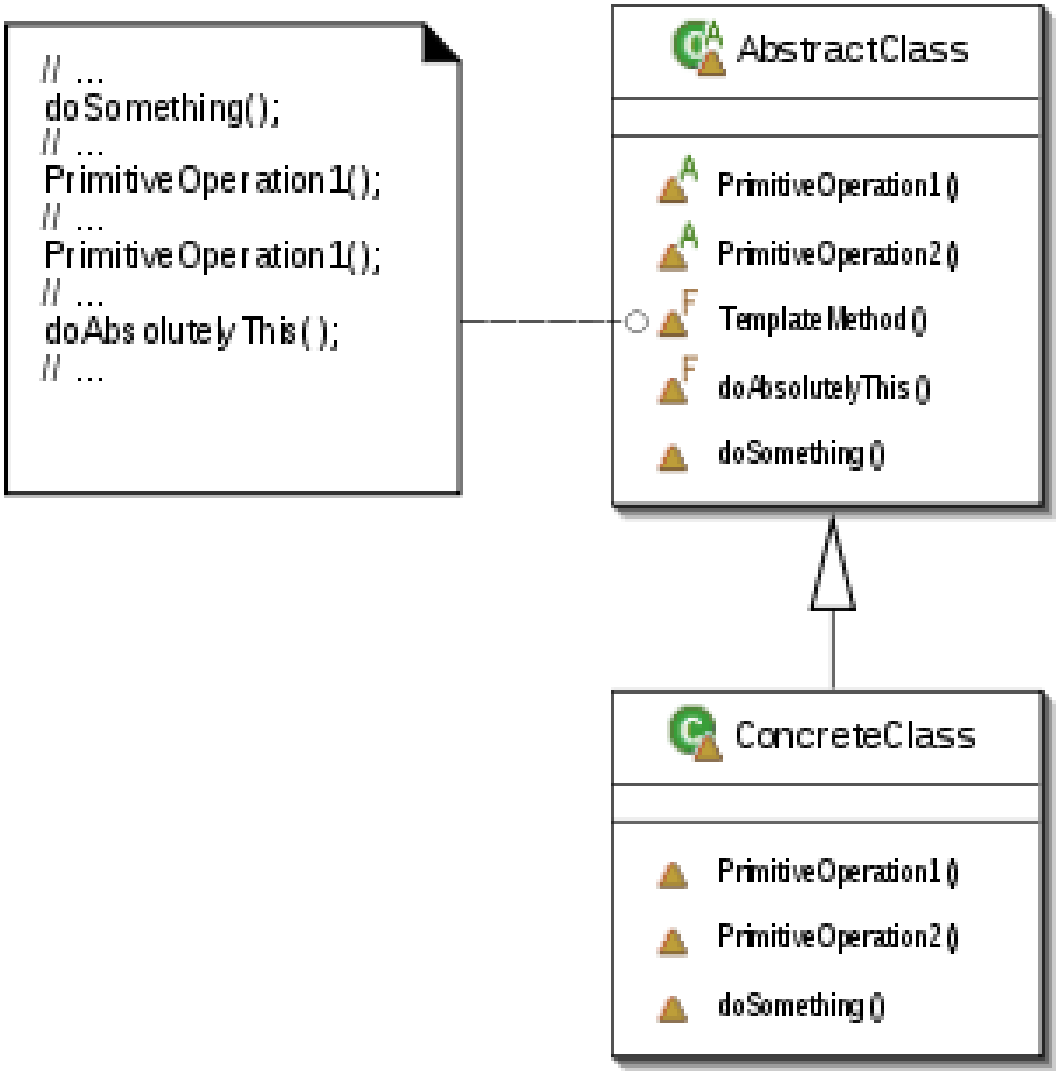
Template method is called

Template Method pattern

```
public abstract class AbstractClass
{
   final void templateMethod()
   {
      primitiveOperation1();
      primitiveOperation2();
      concreteOperation();
   }
   abstract void primitiveOperation1();
   abstract void primitiveOperation2();

   void concreteOperation()
   {
     //Implementation
   }
}
```

The generalised form

Template Method pattern



Template Method UML diagram

Template Method pattern

- Refinements
  - Allow the subclasses to *choose* whether they want to implement an operation in the template method
  - The superclass provides an implementation of a hook operation, which may do nothing by default
  - The sub-classes can extend the template method by implementing the hook, or not
  - So, of example, we could define an *addTopping*() hook to the PieMaker.makePie() method
    - *CottagePieMaker* could implement this hook, providing a cheddar cheese topping
    - *SheperdsPieMaker* could choose not to

Template Method pattern

```
class PieMaker
{
   final public void makePie()
   {
      prepareMeat();
      preparePotatoes();
      assembleInDish();
      addTopping();
      bake();
   }

   … // Other methods

   public void addTopping()
   {
      // Do nothing, hook operation
   }
}
```

Template Method pattern

```
class CottagePieMaker extends PieMaker
{
   public void prepareMeat()
   {
      System.out.println(“Frying the beef”);

   }

   // Chooses to override hook operation
   public void addTopping()
   {
      System.out.println(“Sprinkling cheddar cheese”);
   }

}
```

Template Method pattern

```
class SheperdsPieMaker extends PieMaker
{

    public void prepareMeat()
    {
        System.out.println(“Browning off the lamb”);

    }


    // Does not override hook operation. Plain potato topping.
}
```

Template Method pattern

- The control structure that you get when you apply the template pattern is often called the *Hollywood Principle*
  - Don't call us, we'll call you.
- Using this principle, the template method in a parent class *controls* the overall process by calling subclass methods as required
- This is also known as *inversion of control*
  - Usually, you call superclass methods from the subclass

Summary

- Use the Chain of Responsibility pattern
  - When you want to give more than one object a chance to handle a request
  - When the request handlers form a natural hierarchy
- Use the Template Method pattern
  - To implement the invariant parts of an algorithm once and leave it up to subclasses to implement the behaviour that can vary
- Reading
  - *State* and *Strategy* patterns, in both HFDP and DPFD