Iterator Pattern

- Use a list iterator to iterate through the elements of a linked list (good method below for traverse)

```
LinkedList<String> list = . . .;
ListIterator<String> iterator = list.listIterator();
while (iterator.hasNext())
{
   String current = iterator.next();
   . . .
}
```

- hasNext tests where end of iterator
- Next returns current element and advance iterator
- *Iterators do not expose internal structure of a class*
  - *Info hiding*
  - *Contracts*
- Other traversal methods are shit

EX. Suppose we have a queue class

```
void add(E x)
E peek()
E remove()
int size()
```

- E - type of queue elements

Array structure with random access

```
E get(int i)
void set(int i, E x)
void add(E x)
int size()
```

But its harder to add/remove elements from the MIDDLE of linked list

Cursor Implementation for Traversal

- Implement a list with a cursor
- List cursor marks a position like cursor in word
- Interface of list with cursor

```
E getCurrent()    // Get element at cursor
void set(E x)    // Set element at cursor to x
E remove()    // Remove element at cursor
void insert(E x)    // Insert x before cursor
void reset()    // Reset cursor to head
void next()    // Advance cursor
boolean hasNext()    // Check if cursor can be advanced
```

- Reset method resets cursor to start
- Next method advance it to next element

```
for (list.reset(); list.hasNext(); list.next())
{
   do something with list.getCurrent();
}
```

To traverse the list

Cons of cursor:

- Bc there is one cursor, you can't compare different list elements
- Cant print list
- USE ITERATOR

Pros of iterators:

- You can attach more than one iterator to collection
- Useful outside of collection classes
  - Scanner is an iterator (string tokenizers)
  - input streams

5.6-5.8 10.5

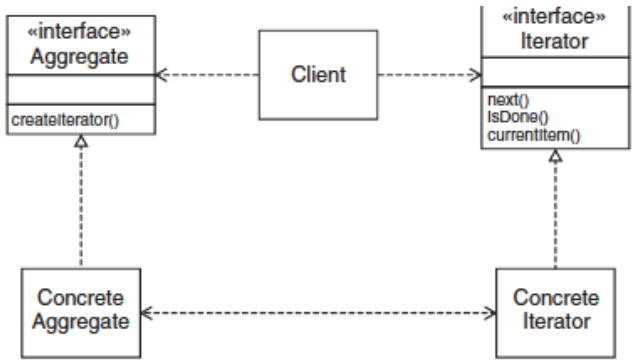
ITERATOR:

Context:

1. An object (aggregate) contains other objects (elements)
2. Clients (method that use the aggregate) need access to the elements
3. Aggregate should not expose its internal structure
4. There may be multiple clients that need simultaneous access

Solution:

1. Define an iterator class that fetches one element at a time
2. Each iterator object needs to keep track of the position of the next element to fetch
3. If there are several variations of the aggregate and iterator classes, it is best if they implement common interface types. Then he client only need to know the interface types, not concrete classes



- Names of interface types, classes and methods are examples, not actual methods within libraries

Linked list iterator example:

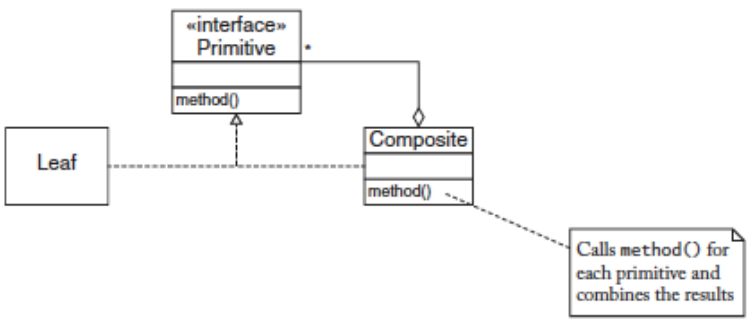
Name in Design Pattern	Actual Name
`Aggregate`	`List`
`ConcreteAggregate`	`LinkedList`
`Iterator`	`ListIterator`
`ConcreteIterator`	An anonymous class that implements the `ListIterator` interface type
`createIterator()`	`listIterator()`
`next()`	`next()`
`isDone()`	Opposite of `hasNext()`
`currentItem()`	Return value of `next()`

Important to have:

- Aggregate
- Concrete aggregate
- Iterator
- Concrete Iterator

COMPOSITE PATTERN

- The composite pattern teaches how to combine several objects into an object that has the same behaviour as its parts
- Where primitive objects can be grouped into composite objects, and the composites are considered primitive objets
- A method of the composite object must also be applied to its primitive objects and combining the results
  - Ex. To find the right size of a container, the container must get the right size of all the components of the container and combine the results



Name in Design Pattern	Actual Name
`Primitive`	`Component`
`Composite`	`Container` or a subclass such as `JPanel`
`Leaf`	A component that has no children such as `JButton` or `JTextArea`
`method()`	A method of the `Component` interface such as `getPreferredSize`

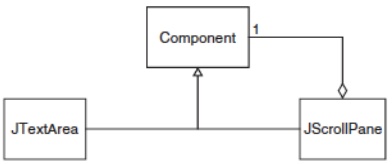
Context

1. Primitive objects can be combined into composite objects
2. Clients treat a composite object as a primitive object

Solution

1. Define an interface type that is an abstraction for the primitive objects
2. A composite object contains a collection of primitive objects
3. Both primitive classes and composite classes implement the interface type
4. When implementing a method from the interface type, the composite class applies the method to its primitive objects and combines the results

DECORATOR PATTERN



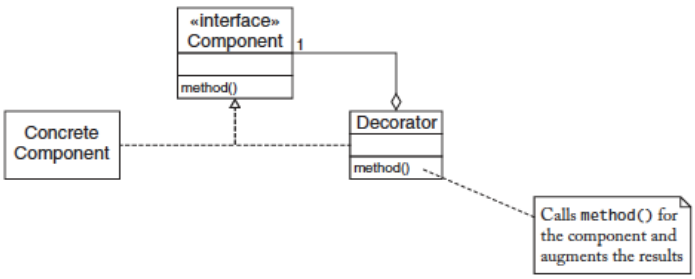
- Adding a scroll bar is a *decoration*
  - When components has more info that can be shown on screen
- The Decorator pattern teaches how to form a class that adds functionality to another class while keeping its interface
- Use decorator whenever a class enhances the functionality of another class while preserving its interface
- Decorator implements a method from the component interface by invoking the same method on the component and then augmenting the result.

Context

1. You want to enhance the behaviour of a class. We'll call it the component class
2. A decorated component can be used in the same way as a plain component
3. The component class does not want to take on the responsibility of the decoration
4. There may be an open-ended set of possible decorations

Solution

1. Define an interface type that is an abstraction for the component
2. Concrete component classes implement this interface type
3. Decorator classes also implement this interface type
4. A decorator object manages the component object that it decorates
5. When implementing a method from the component interface type, the decorator class applies the method to the decorated component and combines the result with the effect of the decoration



Notes:

- Decorator looks very similar to composite pattern
- Min difference:
  - Decorator: enhances the behaviour of a single component
    - Enhance and modify the behaviour of ONE component
  - Composite collects multiple components
    - Collects and treat them all the same
- The differ in the number of concrete components
- SWING decorator example

Name in Design Pattern	Actual Name
`Component`	`Component`
`ConcreteComponent`	`JTextArea`
`Decorator`	`JScrollPane`
`method()`	A method of the `Component` interface. For example, the `paint` method paints a part of the decorated component and the scroll bars.

`How to Recognize Patterns` (composite and decorator looks alike)

1. Look at intent
   - Composite: group components into a whole
   - Decorator: decorate a component

2. Where is pattern used?

- Decorator: scroll bar

3. Follow contexts...they are that pattern if all context is true

- Swing border is not a decorator pattern

OBSERVER PATTERN

Ex. When u have a present and edit window (ex. Web development). When you type text into one window, how does it show up in another window?

- They implemented a *model/view/controller architecture*
- The **model** object holds info in some data structure...holds raw data
    - It has no visual appearance
- The **view** object draw the visible parts of the data
- Each view has a **controller object** (that processes user interaction)
    - May process mouse and keyboard events from he windowing system etc.

Ex. User types text into one of the window

- Controller tells model to insert text somewhere
- Model notifies all views of a change in the model
- All views repaint themselves
- During painting, view asks model for the change to update

Notification mechanism

- Model knows a number of **observer objects**
- **Observers** are interested in state changes in the model

Observer pattern teaches how an object can tell other objects about events.

- **Notification patter = observer pattern**

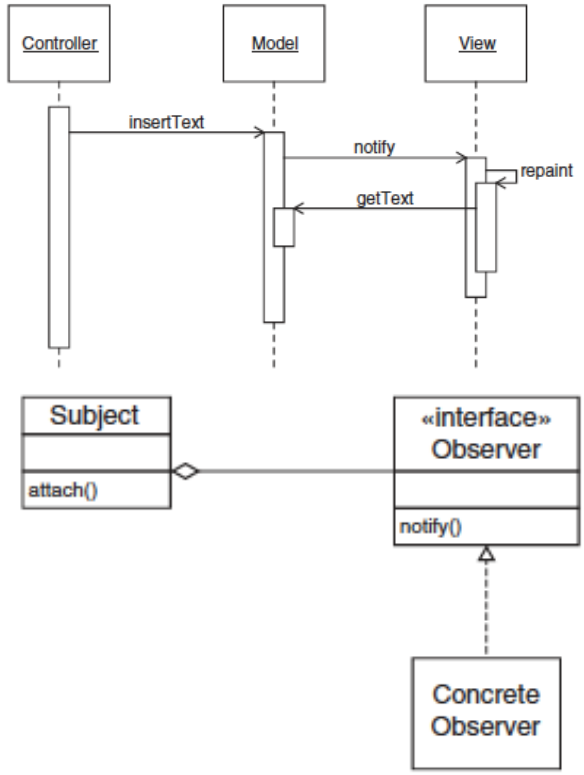
Context

1. An object (subject) is the source of events(ex. My data has changed)
2. One or more observer objects want to know when an event occurs

Solution

1. Define an observer interface type. Observer classes must implement this interface type
2. The subject maintains a collection of observer objects
3. The subject class supplies methods for attaching observers
4. Whenever an event occurs, the subject notifies all observers.

Fact: ex. All user interface elements: buttons, menus etc.



Name in Design Pattern	Actual Name
`Subject`	`JButton`
`Observer`	`ActionListener`
`ConcreteObserver`	The class that implements the `ActionListener` interface type
`attach()`	`addActionListener`
`notify()`	`actionPerformed`

SINGLETON PATTERN

- A singleton class has exactly one instance
- A singleton class is a class that has a single object
  - The object is a global facility for all clients
- Ex. A program for various classes that need to generate random numbers
- Computer-generated random numbers should really be called pseudo-random numbers
  - you need to start with a SEED value and transform it to obtain the first value of the sequence . Then you apply the transformations again for the next value etc.
- Lets design a class SingleRandom that provides a single random number generator
- Make a **constructor private** to make sure that the class has a single instance
- Class constructs the instance and returns it in the static getInstance method

```
Public class SingleRandom{
    private SingleRandom(){generator = new Random()}
    public void setSeed(int seed){generator.setSeed(seed);}
    Public int nextInt(){return generator.nextInt();}
    public static SingleRandom getInstance(){return instance;}

    private Random generator;
    private static SingleRandom instance = new SingleRandom();
}
```

- Static field `instance` stores a reference to the unique `SingleRandom` object.
  - **A static field is a global variable**
- Clients have only ONE WAY to obtain a `SingleRandom` object: call static `getInstance` method

```
Int randomNumber = SingleRandom.getInstance().nextInt();
```

- The static instance field is initialized with the singleton object before the first call to the getInstance method occurs.

Context

1. All clients need to access a single shared instance of a class
2. You want to ensure that no additional instances can be created accidentally

Solution

1. Define a class with a private constructor
2. The class constructs a single instance of itself
3. Supply a static method that returns a reference to the single instance

Notes:

- Uncommon; only applies to classes that are guaranteed to have a unique instance.
- Ex. Main method

MVC
Observer pattern (5.3)
Strategy pattern
Template method pattern
Chapter 9
Chapter 10

- Visitor
- Factory method pattern

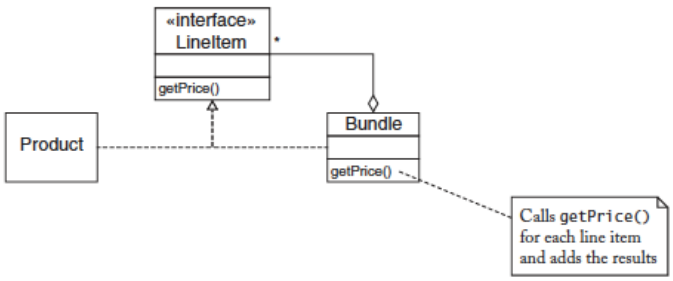
Example using multiple data patterns

- Implement invoice composed of line items
- Line items has a (1) description and (2) price

```
Class interface LineItems{
    String toString(); //@return description
    double getPrince(); //@return price
}
```

- There are different kinds of line items - ex. **Product**

```
// a product with price and description
Public class Product implements LineItems{
    public Product(double price, String description){//constructor
        this.price = price;
        this.description = description;
    }
    public double getPrice(){return price;}
    public String toString(){return
description;}
    private double price;
    private String description;
}
```

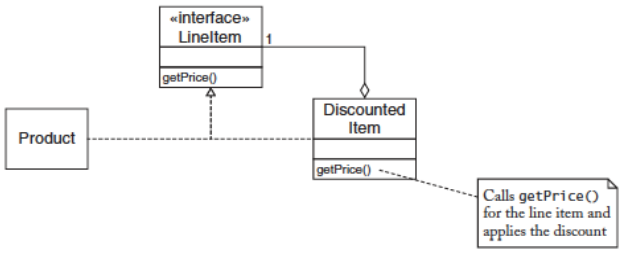


- Now consider something more complex; what if we sell *bundles **of related items***
  - Add a bundle to invoice
  - *so a bundle contains line items and is a line item*
    - Composite pattern
- Composite pattern teaches us that the Bundle class should implement the LineItem interface type
  - When implementing a LineItem method, the bundle class should apply the method to the individual items and combine the result.
  - Ex. How `getPrice` method of `Bundle` class adds the prices of the items in the bundle

```
Bundle Class
Import java.util.*

//a bundle of items that is also a line item
Public class Bundle implements LineItem{
    public Bundle(){ items = new ArrayList<LineItem>();}
    public void add(LineItem item){items.add(item);}
    public double getPrince(){
        double price =0;
        for(LineItem item: items)
            price += item.getPrice();
        return price;
    }
    public String toString(){
        String description = "Bundle Items: ";
        for(int i=0; i< items.size(); i++){
            if(i>0) description += ", ";
            description += items.get(i).toString();
        }
    return description;
    }
    private ArrayList<ListItems> items;
}
```

- We can use **decorators** to implement discounts to a bundle
- Ex. `getPrice` method of `DiscountedItem` class calls the `getPrice` method of the discounted item and then applies the discount




```
DiscountedItem.java
//a decorator for an item that applies
//a discount
Public class DiscountedItem implements
LineItem{
    public DiscountedItem(LineItem item, double Discount){
        this.item = item;
        this.Discount = Discount;
    }
    public String toString(){
        return item.toString + "Discount: "+ Discount;
    }
    public getPrice(){
        return item.getPrice() * Discount;
    }
    private LineItem item;
    private double Discount;
}
```

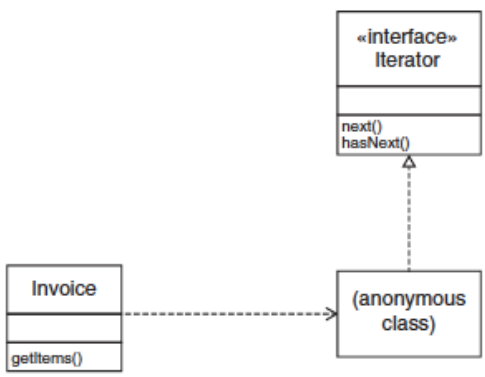
- Invoice class. Invoice holds a *collection* of line items

```
Public class Invoice{
    public void addItem(LineItem item){items.add(item);}
    ...
    private ArrayList<LineItem> items;
}
```

- Clients of Invoice class may need to know the line items inside an invoice
  - But we dont want to reveal the structure of Invoice class

```
Public interface Iterator<E>{
    boolean hasNext();
    E next();
    void remove();
}
```

- Remove is an optional operation
- But need to use bc interface implementation




```
Public Iterator<LineItem> getItems(){
    return new
        Iterator<LineItem>(){
            public boolean hasNext(){
                return current < items.size();
            }
            public LineItem next(){
                LineItem r = items.get(current);
                current++;
                return r;
            }
            public void remove(){
                throw new UnsupportedOperationException();
            }private int current=0;
        };
}
```

Full code of class Invoice in 10.8

Layout Managers and the Strategy Pattern

Build user interfaces from individual user interface **components**: buttons, text fields etc.

- *You place components in **containers***

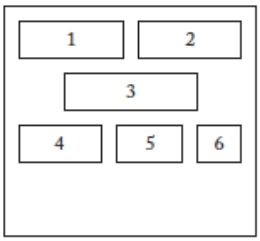
When you add components to container, need to determine:

1. Size of component
2. Where to put it

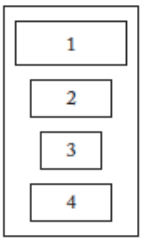
A layout manager arranges the components in a container

- It looks at the sizes of the components and computes their positions

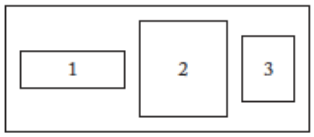
Using predefined layout managers



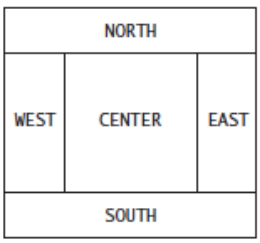
FlowLayout
- lay components L→R



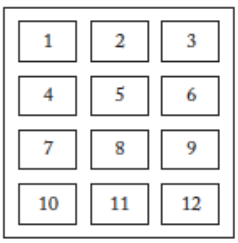
BoxLayout (vertical)
- vertical or horizontal in one column or row



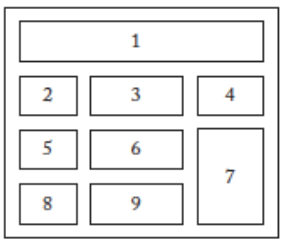
BoxLayout (horizontal)



BorderLayout
- 5 areas
- specify which area component should be put
- grows component to fill area
- can have empty areas



GridLayout
- all components are resized
1) all same size



GridBagLayout
- components can have diff size.

Ex. Voice mail system

Panel 1: Keypad arrange in grid layout

```
JPanel keyPanel = new JPanel();
keyPanel.setLayout(new GridLayout(4.3));
For(int i=0; i<12; I++){
    Button keyButton = newJButton(...);
    keyPanel.add(keyButton);
    keyButton.addActionListener(...);
}
```

Panel 2: for text area for simulated speaker:

- Use border layout (label NORTH, text CENTER)

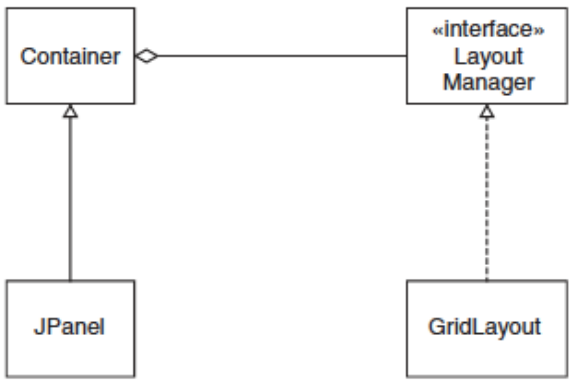
```
JPanel speakerPanel = new JPanel();
speakerPanel.setLayout(new BorderLayout());
speakerPanel.add(new Label("Speaker:"),
BorderLayout.NORTH);
speakerField = new JTextArea(10,25);
speakerPanel.add(speakerField, BorderLayout.CENTER);
```

- Microphone panel and button
    - Botton inside a separate panel and add to South go Microphone

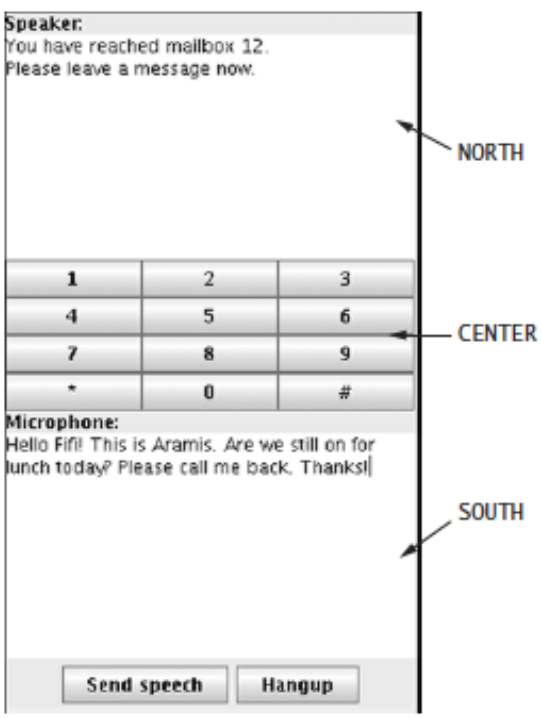
Pg. 186

Ex.

```
JPanel keyPanel = new KPanel();
keyPanel.setLayout(new GridLayout(4.3));
```



- Panel is just a container visit visible decorations that can hold components



Ch5/mailgui/Telephone.java

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

/**
   Presents a phone GUI for the voice mail system.
*/
public class Telephone
{
   /**
      Constructs a telephone with a speaker, keypad,
      and microphone.
   */
   public Telephone()
   {
      JPanel speakerPanel = new JPanel();
      speakerPanel.setLayout(new BorderLayout());
      speakerPanel.add(new JLabel("Speaker:"),
            BorderLayout.NORTH);
      speakerField = new JTextArea(10, 25);
      speakerPanel.add(speakerField,
            BorderLayout.CENTER);
      String keyLabels = "123456789*0#";
      JPanel keyPanel = new JPanel();
      keyPanel.setLayout(new GridLayout(4, 3));
      for (int i = 0; i < keyLabels.length(); i++)
      {
         final String label = keyLabels.substring(i, i + 1);
         JButton keyButton = new JButton(label);
         keyPanel.add(keyButton);
         keyButton.addActionListener(new
            ActionListener()
            {
               public void actionPerformed(ActionEvent event)
               {
                  connect.dial(label);
               }
            });
      }

      final JTextArea microphoneField = new JTextArea(10,25);

      JButton speechButton = new JButton("Send speech");
      speechButton.addActionListener(new
         ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               connect.record(microphoneField.getText());
               microphoneField.setText("");
            }
         });

      JButton hangupButton = new JButton("Hangup");
      hangupButton.addActionListener(new
         ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               connect.hangup();
            }
         });

      JPanel buttonPanel = new JPanel();
      buttonPanel.add(speechButton);
      buttonPanel.add(hangupButton);

      JPanel microphonePanel = new JPanel();
      microphonePanel.setLayout(new BorderLayout());
      microphonePanel.add(new JLabel("Microphone:"),
            BorderLayout.NORTH);
      microphonePanel.add(microphoneField, BorderLayout.CENTER);
      microphonePanel.add(buttonPanel, BorderLayout.SOUTH);

      JFrame frame = new JFrame();
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.add(speakerPanel, BorderLayout.NORTH);
      frame.add(keyPanel, BorderLayout.CENTER);
      frame.add(microphonePanel, BorderLayout.SOUTH);

      frame.pack();
      frame.setVisible(true);
   }

   /**
      Give instructions to the mail system user.
   */
   public void speak(String output)
   {
      speakerField.setText(output);
   }

   public void run(Connection c)
   {
      connect = c;
   }

   private JTextArea speakerField;
   private Connection connect;
}
```

Implementing a Custom Layout Manager

A layout manager must support the LayoutManager interface type:

```
public interface LayoutManager
{
   Dimension minimumLayoutSize(Container parent);
   Dimension preferredLayoutSize(Container parent);
   void layoutContainer(Container parent);
   void addLayoutComponent(String name, Component comp);
   void removeLayoutComponent(Component comp);
}
```

`minimumLayoutSize` and `preferredLayoutSize` methods determine the minimum and preferred size of the container when the components are laid out
`layoutContainer` method lays out the components in the container

- Setting position and size for each component

Last 2 methods can be do-nothing methods

To write a layout manager, **start out** with `preferredLayoutSize` method

- Compute preferred width and height by combining W and L of each component within container
- Calculate width: widest component on the left and widest component on the right; add them together with gap
- Calculate height: add heights of all rows

When container is ready to lay out its contents, call `layoutContainer` method

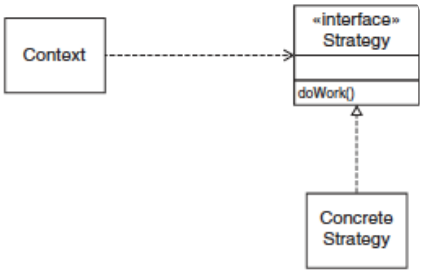
- Computes the positions of each component and then calls `setBounds` to put component in correct location

ex. Strategy: `Class formLayout implement LayoutManager`

`Code on pg.191`

STRATEGY PATTERN

Strategy pattern teaches how to supply variants to an algorithm
To produce a particular layout, make an object of the layout manager class and give it to the container
When container needs to execute the layout algorithm, it calls the appropriate methods of the layout manager object.



- Strategy pattern applies whenever u want to allow a client to supply an algorithm
- The many objects of strategy interface type, can have different algorithms

Name in Design Pattern	Actual Name
Context	Container
Strategy	LayoutManager
ConcreteStrategy	A layout manager such as BorderLayout
doWork()	A method of the LayoutManager interface type such as layoutContainer

Context

1. A class (context class) can benefit from different variants of an algorithm
2. Clients of the context class sometimes want to supply custom versions of the algorithm

Solution

1. Define an interface type that is an abstraction for the algorithm. We'll call this interface type the *strategy*
2. Concrete strategy classes implement the strategy interface type. Each strategy class implements a version of the algorithm
3. The client supplies a concrete strategy object to the context class
4. Whenever the algorithm needs to be executed, the context class calls the appropriate methods of the strategy object.

Ex. Comparator and country method

- Pass `Comparator` object to `Collections.sort` method to specify how elements are compared.
- Comparator object encapsulates then comparison algorithm
- By varying the comparator, you can sort based on different criteria

Name in Design Pattern	Actual Name
Context	Collections
Strategy	Comparator
ConcreteStrategy	A class that implements the Comparator interface type
doWork()	compare()

```
Comparator comp = new CountryComparatorByName();
Collections.sort(countries, comp);
```

Class can benefit from multiple algorithms that implement a similar activity

TEMPLATE METHOD PATTERN

- Not only based on common methods but <u>common steps within a method</u>
  - Cant be done with Interface
- Suppose we're drawing shapes...selected shapes need to be drawn in a specific way to they can be visually distinguished.
  - Bad solution: where each shape class have to provide a separate mechanism to draw the decoration.

Below method supplied in SelectableShape class:

```
public void drawSelection(Graphics2D g2)
{
   translate(1, 1);
   draw(g2);
   translate(1, 1);
   draw(g2);
   translate(-2, -2);
}
```

- Abstract SelectableShape class doesn't need to know how subclass will draw and translate

<- Template Pattern Example

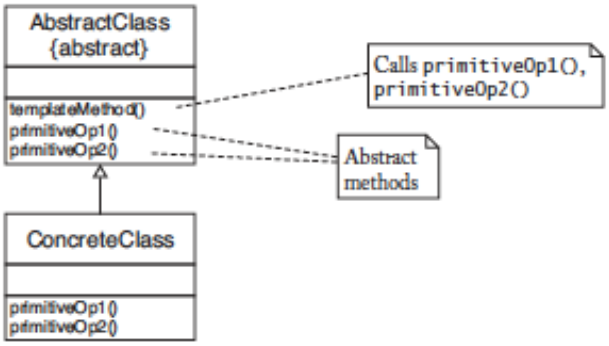
- Superclass defines a method that calls <u>primitive operations</u> that a subclass needs to supply.
- Each subclass can supply the primitive operations as is most appropriate for it
- Template method contains info on how to combine primitive operations into complex operations

Context

1. An algorithm is applicable for multiple types
2. Algorithm can be broken down into **primitive operations**. The primitive operations can be different for each type
3. The order of the primitive operations int he algorithm doesn't depend on the type.

Solution

1. Define an abstract superclass that has a method for the algorithm and abstract methods for the primitive operations
2. Implement the algorithm to call the primitive operations in the appropriate order.
3. Do not define the primitive operations in the superclass or define them to have appropriate default behaviour
4. Each subclass defines the primitive operations but not the algorithm



Name in Design Pattern	Actual Name
AbstractClass	SelectableShape
ConcreteClass	CarShape, HouseShape
templateMethod()	drawSelection()
primitiveOp1(), primitiveOp2()	translate(), draw()

Ch6/scene2/SelectableShape.java

```
1  import java.awt.*;
2  import java.awt.geom.*;
3
4  /**
5     A shape that manages its selection state.
6  */
7  public abstract class SelectableShape implements SceneShape
8  {
9     public void setSelected(boolean b)
10    {
11       selected = b;
12    }
13
14    public boolean isSelected()
15    {
16       return selected;
17    }
18
19    public void drawSelection(Graphics2D g2)
20    {
21       translate(1, 1);
22       draw(g2);
23       translate(1, 1);
24       draw(g2);
25       translate(-2, -2);
26    }
27
28    private boolean selected;
29 }
```

Ch6/scene2/HouseShape.java

```
1  import java.awt.*;
2  import java.awt.geom.*;
3
4  /**
5     A house shape.
6  */
7  public class HouseShape extends SelectableShape
8  {
   structs a house shape.
   am x  the left of the bounding rectangle
   am y  the top of the bounding rectangle
   am width  the width of the bounding rectangle

    HouseShape(int x, int y, int width)

   .x = x;
   .y = y;
   .width = width;


    void draw(Graphics2D g2)

   angle2D.Double base
      = new Rectangle2D.Double(x, y + width, width, width);

   The left bottom of the roof
   t2D.Double r1
      = new Point2D.Double(x, y + width);
30    // The top of the roof
31    Point2D.Double r2
32       = new Point2D.Double(x + width / 2, y);
33    // The right bottom of the roof
34    Point2D.Double r3
35       = new Point2D.Double(x + width, y + width);
36
37    Line2D.Double roofLeft
38       = new Line2D.Double(r1, r2);
39    Line2D.Double roofRight
40       = new Line2D.Double(r2, r3);
41
42    g2.draw(base);
43    g2.draw(roofLeft);
44    g2.draw(roofRight);
45 }
46
47 public boolean contains(Point2D p)
48 {
49    return x <= p.getX() && p.getX() <= x + width
50       && y <= p.getY() && p.getY() <= y + 2 * width;
51 }
52
53 public void translate(int dx, int dy)
54 {
55    x += dx;
56    y += dy;
57 }
58
59 private int x;
60 private int y;
61 private int width;
62 }
```

THREADS

= program units that can be executed in parallel

- How to start new threats
- How to coordinate threads; synchronizing

Ex. Run multiple programs at the same time.

- Your OS actually goes back and forth between processes, not run this simultaneously
- If you have multiple CPUs, then can run in parallel (one process per processor)
- The java virtual machine executes each thread for a short amount of time and switches to another thread.

Difference between processes and threads

- OS isolates each process from each other
  - Cant overwrite another process' memory
  - So switching between processes is SLOW
- Threads run within one process
  - Fast switching but share memory

Steps to run a thread:

1. Define a class that implements the `Runnable` interface type. That interface type has a **single** method called `run`.
2. Place the code for the task into the `run` method of the class
3. Create an object for the class
4. Construct a `Thread` object and supply the `Runnable` object in the constructor.
5. Call the start method of the `Thread` object to start the thread

```
public interface Runnable
{
   void run();
}
```

- Start method of `Thread` object starts a new thread the executes the `run` method of its `Runnable`

```
for (int i = 1; i <= REPETITIONS; i++)
{
   System.out.println(i + ": " + greeting);
   Thread.sleep(DELAY);
}
```

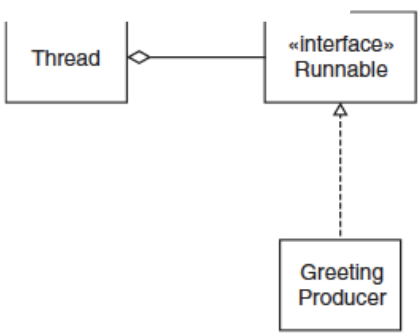
<<= Ex. Run 2 threads in parallel, each print ten greetings. Each thread executes this loop

```
public class MyRunnable implements Runnable
{
   public void run()
   {
      try
      {
         do work
      }
      catch (InterruptedException exception)
      {
      }
      clean up, if necessary
   }
   . . .
}
```

- After printing, each thread sleep for a while, lets other thread run
- **Sleeping might be risky, sleep too long = no long useful**
- Thread is terminated when you interrupt it:
  `InterruptedException`, catch in `run` method

GreetingProducer.java

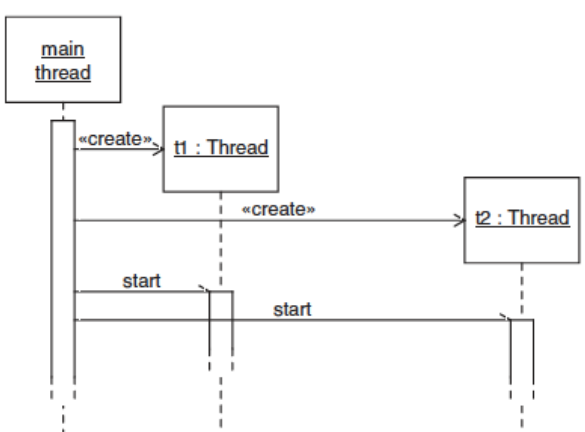
```
public class GreetingProducer implement Runnable{
    // @param aGreeting: the greeting to display
    public GreetingProducer(String aGreeting){
        aGreeting = greeting;
    }
    public void run(){
        try{
            for(int I=1; I<= REPETITIONS; I++){
                System.out.println(I + ": " + greeting);
                Thread.sleep(DELAY);
            }
        catch(InterruptedException exception){}
        }
    }
    private String greeting;
    private static final int REPETITION;
    private static final int DELAY;
}
```



The above is not a thread...its just a class with run method defined....

Create thread to run thread

```
Runnable r = new GreetingProducer("Hello, World!");
Thread t = new Thread(r);
T.start
```



Start method creates a new thread in virtual machine
- Thread calls the run method of Runnable object
- Thread terminate = run method terminates

Run two threads in parallel = make and start 2 thread objects

```
Public class ThreadTester{
    public static void main(String[] arg){
        Runnable r1 = new GreetingProducer("Hello, World!");
        Runnable r2 = new GreetingProducer("Bye");
        Thread t1 = new Thread(r1);
        Thread t2 = new Thread(r2);
        t1.start();t2.start();
    }
}
```

- **Main method runs in its own thread: MAIN THREAD**
  - Main terminates when start t2

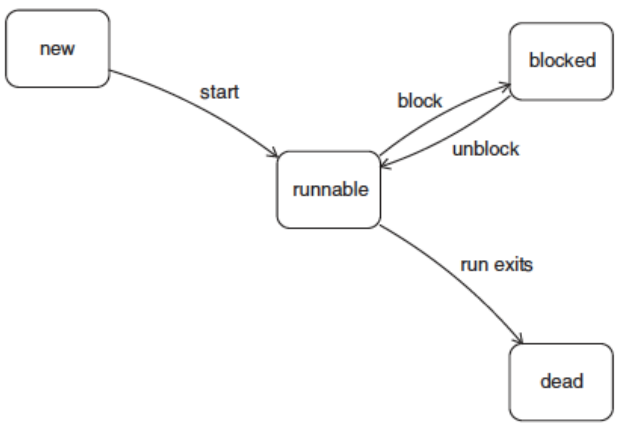
Scheduling Threads

```
1: Hello, World!
1: Goodbye, World!
2: Hello, World!
2: Goodbye, World!
3: Hello, World!
3: Goodbye, World!
4: Hello, World!
4: Goodbye, World!
5: Hello, World!
5: Goodbye, World!
6: Hello, World!
6: Goodbye, World!
7: Hello, World!
7: Goodbye, World!
8: Goodbye, World!
8: Hello, World!
9: Goodbye, World!
9: Hello, World!
10: Goodbye, World!
10: Hello, World!
```

<<= ex. Output of tester program
- See that 7-8, the threads didn't alternate
- Thread scheduler does not guarantee order
- So there will ALWAYS be variations, especially calling OS services (input/output)

Algorithm that scheduler uses to pick nest thread

- Each thread has:
  - A thread state
  - A priority
- Thread state is one of the following:
  - New (before start is called)
  - Runnable
  - Blocked
  - Dead (after the run method exists)
- **There is no separate state to indicate whether a runnable thread is actually running**



Reasons thread can enter **blocked state**
- Sleeping
- Waiting for input/output
- Waiting to acquire a lock
- Waiting for a condition

Thread will remain in blocked until event is over

Scheduler will activate a new thread when one of the 3 events occurs:
- A thread has completed its time slice
- A thread has blocked itself
- A thread with a higher priority has become runnable

Terminating Threads
- Thread terminate when the run method of its Runnable returns
- How to terminate a running method
  - Ex. Have multiple threads to find a solution to a problem. First thread to solve, terminate all the others.
  - DONT STOP WITH stop, dangerous bc resources are shared
  - Instead, thread should terminate itself
    - Use `interrupt` method: `t.interrupt();`
      - Does not terminate, just **flags**
  - Run method should check if there are any interruption...cover `run` with a `try` block

```
public void run()
{
   try
   {
      while (more work to do)
      {
         do work
         Thread.sleep(DELAY);
      }
   }
   catch (InterruptedException exception)
   {
   }
   clean up
}
```

- The run code on the right works because the `sleep` method checked the interrupted flag
  - If the flag is set, `sleep` clears it and throws `InterruptedException`

Check interrupted flag manually: `if(Thread.currentThread().isInterrupted())...`

Thread Synchronization

Corrupting a Shared Data Structure

- Threads can share access to same object
- Ex. (PRODUCER)Two threads each inserts 100 strings into queue, (CONSUMER) third string removes ALL strings

ch9/queue1/ThreadTester.java

```
/**
   This program runs two threads in parallel.
*/
public class ThreadTester
{
   public static void main(String[] args)
   {
      BoundedQueue<String> queue = new BoundedQueue<String>(10);
      queue.setDebug(true);
      final int GREETING_COUNT = 100;
      Runnable run1 = new Producer("Hello, World!",
            queue, GREETING_COUNT);
      Runnable run2 = new Producer("Goodbye, World!",
            queue, GREETING_COUNT);
      Runnable run3 = new Consumer(queue, 2 * GREETING_COUNT);

      Thread thread1 = new Thread(run1);
      Thread thread2 = new Thread(run2);
      Thread thread3 = new Thread(run3);

      thread1.start();
      thread2.start();
      thread3.start();
   }
}
```

ch9/queue1/Producer.java

```
/**
   An action that repeatedly inserts a greeting into a queue.
*/
public class Producer implements Runnable
{
   /**
      Constructs the producer object.
      @param aGreeting the greeting to insert into a queue
      @param aQueue the queue into which to insert greetings
      @param count the number of greetings to produce
   */
   public Producer(String aGreeting, BoundedQueue<String> aQueue,
         int count)
   {
      greeting = aGreeting;
      queue = aQueue;
      greetingCount = count;
   }

   public void run()
   {
      try
      {
         int i = 1;
         while (i <= greetingCount)
         {
            if (!queue.isFull())
            {
               queue.add(i + ": " + greeting);
               i++;
            }
            Thread.sleep((int) (Math.random() * DELAY));
         }
      }
      catch (InterruptedException exception)
      {
      }
   }

   private String greeting;
   private BoundedQueue<String> queue;
   private int greetingCount;

   private static final int DELAY = 10;
}
```

ch9/queue1/Consumer.java

```
/**
   An action that repeatedly removes a greeting from a queue.
*/
public class Consumer implements Runnable
{
   /**
      Constructs the consumer object.
      @param aQueue the queue from which to retrieve greetings
      @param count the number of greetings to consume
   */
   public Consumer(BoundedQueue<String> aQueue, int count)
   {
      queue = aQueue;
      greetingCount = count;
   }

   public void run()
   {
      try
      {
         int i = 1;
         while (i <= greetingCount)
         {
            if (!queue.isEmpty())
            {
               String greeting = queue.remove();
               System.out.println(greeting);
               i++;
            }
            Thread.sleep((int) (Math.random() * DELAY));
         }
      }
      catch (InterruptedException exception)
      {
      }
   }

   private BoundedQueue<String> queue;
   private int greetingCount;

   private static final int DELAY = 10;
}
```

Ch9/queue1/BoundedQueue.java

```
1  /**
2     A first-in, first-out bounded collection of objects.
3  */
4  public class BoundedQueue<E>
5  {
6     /**
7        Constructs an empty queue.
8        @param capacity the maximum capacity of the queue
9     */
10    public BoundedQueue(int capacity)
11    {
12       elements = new Object[capacity];
13       head = 0;
14       tail = 0;
15       size = 0;
16    }
17
18    /**
19       Removes the object at the head.
20       @return the object that has been removed from the queue
21       @precondition !isEmpty()
22    */
23    public E remove()
24    {
25       if (debug) System.out.print("removeFirst");
26       E r = (E) elements[head];
27       if (debug) System.out.print(".");
28       head++;
29       if (debug) System.out.print(".");
30       size--;
31       if (head == elements.length)
32       {
33          if (debug) System.out.print(".");
34          head = 0;
35       }
36       if (debug)
37          System.out.println("head=" + head + ",tail=" + tail
38                + ",size=" + size);
39       return r;
40    }
41
42    /**
43       Appends an object at the tail.
44       @param newValue the object to be appended
45       @precondition !isFull();
46    */
47    public void add(E newValue)
48    {
49       if (debug) System.out.print("add");
50       elements[tail] = newValue;
51       if (debug) System.out.print(".");
52       tail++;
53       if (debug) System.out.print(".");
54       size++;
55       if (tail == elements.length)
56       {
57          if (debug) System.out.print(".");
58          tail = 0;
59       }
60       if (debug)
61          System.out.println("head=" + head + ",tail=" + tail
62                + ",size=" + size);
63    }
64
65    public boolean isFull()
66    {
67       return size == elements.length;
68    }
69
70    public boolean isEmpty()
71    {
72       return size == 0;
73    }
74
75    public void setDebug(boolean newValue)
76    {
77       debug = newValue;
78    }
```

```
1: Hello, World!
1: Goodbye, World!
2: Hello, World!
3: Hello, World!
. . .
99: Goodbye, World!
100: Goodbye, World!
```

>> expected output

```
Object[] elements;
int head;
int tail;
int size;
boolean debug;
```

Race Conditions

When you run the program several time, consumer thread gets stuck and won't complete

- It cant retrieve them all
- Complete but print same greeting repeatedly
- To fix race condition, ensure that only one thread manipulates the queue whenever

Here is one of many scenarios that demonstrates how a problem can occur.

1. The first thread calls the `add` method of the `BoundedQueue` class and executes the following statement:
   ```
   elements[tail] = newValue;
   ```
2. The second thread calls the `add` method on the same `BoundedQueue` object and executes the statements
   ```
   elements[tail] = newValue;
   tail++;
   ```
3. The first thread executes the statement
   ```
   tail++;
   ```

The consequences of this scenario are unfortunate. Step 2 overwrites the object that the first thread stored in the queue. Step 3 increments the tail counter past a storage location without filling it. When its value is removed later, some random value will be returned (see Figure 4).

Locks

A thread can acquire a lock to solve above prob. When another thread tries to acquire the same lock, it is blocked. When the first thread releases the lock, the other threads are unblocked.

Two kinds of locks:

- Objects of the `ReentrantLock` class or another class that implements the `Lock` interface type in the `java.util.concurrent.locks` package.
- Locks that are built into every Java object

```
aLock = new ReentrantLock();
. . .
aLock.lock();
try
{
   protected code
}
finally
{
   aLock.unlock();
}
```

- Finally clause ensure the the lock is unlocked even when an exception is thrown in the protected code
- Assume body of add method is protected by a lock...the error is resolved:

1. The first thread calls the `add` method and acquires the lock. The thread executes the following statement:
   ```
   elements[tail] = newValue;
   ```
2. The second thread also calls the `add` method on the same queue object and wants to acquire the same lock. But it can't—the first thread still owns the lock. Therefore, the second thread is blocked and cannot proceed.
3. The first thread executes the statement
   ```
   tail++;
   ```
4. The first thread completes the `add` method and returns. It releases the lock.
5. The lock release unblocks the second thread. It is again runnable.
6. The second thread proceeds, now successfully acquiring the lock.

Remove should also be locked >>
If one thread calls a method, other threads shouldn't run same method on the same object

Each queue need separate lock obj
Threes can operate on diff `BoundedQueue` objects

Avoiding Deadlocks

- Locks for add and remove methods are not enough to solve problem
- Consider action of the producer:

```
if (!queue.isFull())
{
   queue.add(i + ": " + greeting);
   i++;
}
```

Suppose producer thread queue is not full but time slice has elapsed
Another thread gains control and fill queue
The first thread is reactivated and proceeds where it left off

- Adds msg to full queue >> queue is corrupted

```
public void add(E newValue)
{
   queueLock.lock();
   try
   {
      while (queue is full)
         wait for more space
      . . .
   }
   finally
   {
      queueLock.unlock();
   }
}
```

Test should be moved inside the add method

- Then it'll all test for sufficient space.

<< new add method
But how can you wait for more space?

- Cant call sleep inside try block.
- If sleep after locking queueLock, no other thread can remove elements bc that code block is protected by the same lock.
- The consumer will call remove, but it'll be blocked until add exits
- But the add method doesn't exit until it has space available....**deadlock**

RESOLVE DEADLOCKS: Condition INTERFACE

- Each lock has 1 or more `Condition` objects.
- Create like this:

```
private Lock queueLock = new ReentrantLock();
private Condition spaceAvailableCondition
      = queueLock.newCondition();
private Condition valueAvailableCondition
      = queueLock.newCondition();
```

Therefore for our example, we track 2 conditions:

1. space is available for insertion
2. Values are available for removal

Calling `await` on a condition object temp releases a lock and blocks current thread.

Ex. Modified add method

```
public void add(E newValue)
{
   . . .
   while (size == elements.length)
      spaceAvailableCondition.await();
   . . .
}
```

Current thread is blocked until another thread calls `signalAll` or `signal` on the condition object for which the thread is waiting

```
public E remove()
{
   . . .
   E r = elements[head];
   . . .
   spaceAvailableCondition.signalAll(); // Unblock waiting threads
   return r;
}
```

The `valueAvailableCondition` is maintained in the same way. The `remove` method starts with the loop

```
while (size == 0)
   valueAvailableCondition.await();
```

After the `add` method has added an element to the queue, it calls

```
valueAvailableCondition.signalAll();
```

Note that the test for a condition *must* be contained in a `while` loop, not an `if` statement:

```
while (not ok to proceed)
   aCondition.await();
```

The condition must be retested after the thread returns from the call to `await`.

Ch9/queue2/BoundedQueue.java

```
1  import java.util.concurrent.locks.*;
2
3  /**
4     A first-in, first-out bounded collection of objects.
5  */
6  public class BoundedQueue<E>
7  {
8     /**
9        Constructs an empty queue.
10       @param capacity the maximum capacity of the queue
11    */
12    public BoundedQueue(int capacity)
13    {
14       elements = new Object[capacity];
15       head = 0;
16       tail = 0;
17       size = 0;
18    }
19
20    /**
21       Removes the object at the head.
22       @return the object that has been removed from the queue
23    */
24    public E remove() throws InterruptedException
25    {
26       queueLock.lock();
27       try
28       {
29          while (size == 0)
30             valueAvailableCondition.await();
31          E r = (E) elements[head];
32          head++;
33          size--;
34          if (head == elements.length)
35             head = 0;
36          spaceAvailableCondition.signalAll();
37          return r;
38       }
39       finally
40       {
41          queueLock.unlock();
42       }
43    }
44
45    /**
46       Appends an object at the tail.
47       @param newValue the object to be appended
48    */
49    public void add(E newValue) throws InterruptedException
50    {
51       queueLock.lock();
52       try
53       {
54          while (size == elements.length)
55             spaceAvailableCondition.await();
56          elements[tail] = newValue;
57          tail++;
58          size++;
59          if (tail == elements.length)
60             tail = 0;
61          valueAvailableCondition.signalAll();
62       }
63       finally
64       {
65          queueLock.unlock();
66       }
67    }
68
69    private Object[] elements;
70    private int head;
71    private int tail;
72    private int size;
73
74    private Lock queueLock = new ReentrantLock();
75    private Condition spaceAvailableCondition
76          = queueLock.newCondition();
77    private Condition valueAvailableCondition
78          = queueLock.newCondition();
79 }
```

When the call `anIcon.getIconWidth()` is executed, the Java interpreter first looks up the actual type of the object, then it locates the `getIconWidth` method of that type, and finally invokes that method. For example, suppose you pass a `MarsIcon` to the `showMessageDialog` method:

```
JOptionPane.showMessageDialog(. . ., new MarsIcon(50));
```

Then the `getIconWidth` method of the `MarsIcon` class is invoked. But if you supply an `ImageIcon`, then the `getIconWidth` method of the `ImageIcon` class is called. These two methods have nothing in common beyond their name and return type. The `MarsIcon` version simply returns the `size` instance field, whereas the `ImageIcon` version returns the width of the bitmap image.

Polymorphism refers to the ability to select different methods according to the actual type of an object.

The ability to select the appropriate method for a particular object is called *polymorphism*. (The term "polymorphic" literally means "having multiple shapes".)

An important use of polymorphism is to promote *loose coupling*. Have another look at Figure 4. As you can see, the `JOptionPane` class uses the `Icon` interface, but it is decoupled from the `ImageIcon` class. Thus, the `JOptionPane` class need not know anything about image processing. It is only concerned with those aspects of images that are captured in the `Icon` interface type.

Another important use of polymorphism is *extensibility*. By using the `Icon` interface type, the designers of the `JOptionPane` class don't lock you into the use of bitmap icons. You can supply icons of your own design.

4.3 The `Comparable` Interface Type

The `Collections.sort` method can sort objects of any class that implements the `Comparable` interface type.

For another useful example of code reuse, we turn to the `Collections` class in the Java library. This class has a static `sort` method that can sort an array list:

```
Collections.sort(list);
```

The objects in the array list can belong to any class that implements the `Comparable` interface type. That type has a single method:

```
public interface Comparable<T>
{
   int compareTo(T other);
}
```

This interface is a generic type, similar to the `ArrayList` class. We will discuss generic types in greater detail in Chapter 7, but you can use the `Comparable` type without knowing how to implement generic types. Simply remember to supply a type parameter, such as `Comparable<String>`. The type parameter specifies the parameter type of the `compareTo` method. For example, the `Comparable<Country>` interface defines a `compareTo(Country other)` method.

The call

```
object1.compareTo(object2)
```

is expected to return a negative number if `object1` should come before `object2`, zero if the objects are equal, and a positive number otherwise.

Why does the `sort` method require that the objects that it sorts implement the `Comparable` interface type? The reason is simple. Every sorting algorithm compares objects in various positions in the collection and rearranges them if they are out of order. The code for the `sort` method contains statements such as the following:

```
if (object1.compareTo(object2) > 0)
   rearrange object1 and object2;
```

For example, the `String` class implements the `Comparable<String>` interface type. Therefore, you can use the `Collections.sort` method to sort a list of strings:

```
ArrayList<String> countries = new ArrayList<String>();
countries.add("Uruguay");
countries.add("Thailand");
countries.add("Belgium");
Collections.sort(countries); // Now the array list is sorted
```

If you design a class whose objects need to be compared to each other, your class should implement the `Comparable` interface type.

If you have an array list of objects of your own class, then you need to make sure your class implements the `Comparable` interface type. Otherwise, the `sort` method will throw an exception.

For example, here is a class `Country` that implements the `Comparable<Country>` interface type. The `compareTo` method compares two countries by area. The test program demonstrates sorting by area.

Ch4/sort1/Country.java

```
/**
   A country with a name and area.
*/
public class Country implements Comparable<Country>
{
   /**
      Constructs a country.
      @param aName the name of the country
      @param anArea the area of the country
   */
   public Country(String aName, double anArea)
   {
      name = aName;
      area = anArea;
   }

   /**
      Gets the name of the country.
      @return the name
   */
   public String getName()
   {
```

```
23          return name;
24       }
25
26       /**
27          Gets the area of the country.
28          @return the area
29       */
30       public double getArea()
31       {
32          return area;
33       }
34
35       /**
36          Compares two countries by area.
37          @param otherObject the other country
38          @return a negative number if this country has a smaller
39          area than otherCountry, 0 if the areas are the same,
40          a positive number otherwise
41       */
42       public int compareTo(Country other)
43       {
44          if (area < other.area) return -1;
45          if (area > other.area) return 1;
46          return 0;
47       }
48
49       private String name;
50       private double area;
51    }
```

Ch4/sort1/CountrySortTester.java

```
1  import java.util.*;
2
3  public class CountrySortTester
4  {
5     public static void main(String[] args)
6     {
7        ArrayList<Country> countries = new ArrayList<Country>();
8        countries.add(new Country("Uruguay", 176220));
9        countries.add(new Country("Thailand", 514000));
10       countries.add(new Country("Belgium", 30510));
11
12       Collections.sort(countries);
13       // Now the array list is sorted by area
14       for (Country c : countries)
15          System.out.println(c.getName() + " " + c.getArea());
16    }
17 }
```

4.4 The Comparator Interface Type

Now suppose you want to sort an array of countries by the country name instead of the area. It's not practical to redefine the `compareTo` method every time you want to change the sort order. Instead, there is a second `sort` method that is more flexible. You can use *any* sort order by supplying an object that implements the `Comparator` interface type. The `Comparator<T>` interface type requires one method

```
int compare(T first, T second)
```

that returns a negative number, zero, or a positive number depending on whether `first` is less than, equal to, or greater than `second` in the particular sort order.

Similar to the `Comparable` interface type, the `Comparator` interface type is also generic. The type parameter specifies the type of the `compare` method parameters. For example, to compare `Country` objects, you would use an object of a class that implements the `Comparator<Country>` interface type.

You can sort a collection in any sort order by supplying an object of a class that implements the `Comparator` interface type.

Note the method is called `compare`, not `compareTo`—it compares two explicit parameters rather than comparing the implicit parameter to the explicit parameter.

If `comp` is an object of a class that implements the `Comparator` interface type, then

```
Collections.sort(list, comp)
```

sorts the objects in `list` according to the sort order that `comp` defines. Now `list` can contain any objects—they don't have to belong to classes that implement any particular interface type. For example, to sort the countries by name, define a class `CountryComparatorByName` whose `compare` method compares two `Country` objects.

```
public class CountryComparatorByName implements Comparator<Country>
{
   public int compare(Country country1, Country country2)
   {
      return country1.getName().compareTo(country2.getName());
   }
}
```

Now make an object of this class and pass it to the sort method:

```
Comparator<Country> comp = new CountryComparatorByName();
Collections.sort(countries, comp);
```

An object such as `comp` is often called a *function object* because its sole purpose is to execute the comparison function.

The `CountryComparatorByName` class has no state—all objects of this class behave in exactly the same way. However, it is easy to see that some state might be useful. Here is a comparator class that can sort in either ascending or descending order.

```
public class CountryComparator implements Comparator<Country>
{
   public CountryComparator(boolean ascending)
   {
```

```
      if (ascending) direction = 1; else direction = -1;
   }

   public int compare(Country country1, Country country2)
   {
      return direction * country1.getName().compareTo(country2.getName());
   }

   private int direction;
}
```

Then an object

```
Comparator<Country> reverseComp = new CountryComparator(false);
```

can be used to sort an array of `Country` objects from Z to A.

4.5 Anonymous Classes

Consider again the call to the `sort` method of the preceding section:

```
Comparator<Country> comp = new CountryComparatorByName();
Collections.sort(countries, comp);
```

There is actually no need to explicitly name the `comp` object. You can pass an *anonymous object* to the `sort` method since you only need it once.

```
Collections.sort(countries, new CountryComparatorByName());
```

An anonymous object is an object that is not stored in a variable.

After the call to `sort`, the comparator object is no longer needed. Thus, there is no reason to store it in the `comp` variable.

Is it good style to use anonymous objects? It depends. Sometimes, the variable name gives useful information to the reader. But in our situation, the variable `comp` did not make the code clearer. If you look at your own programs, you will find that you often use anonymous values of type `int` or `String`. For example, which of these two styles do you prefer?

```
countryNames.add("Uruguay");
```

or

```
String countryName1 = "Uruguay";
countryNames.add(countryName1);
```

Most programmers prefer the shorter style, particularly if they have to type the code themselves.

An anonymous class is a class without a name. When defining an anonymous class, you must also construct an object of that class.

An anonymous object is handy if you only need an object once. The same situation can arise with classes. Chances are good that you only need the `CountryComparatorByName` class once as well—it is a "throw-away" class that fulfills a very specialized purpose.

If you only need a class once, you can make the class anonymous by defining it inside a method and using it to make a single object.

```
Comparator<Country> comp = new
   Comparator<Country>() // Make object of anonymous class
   {
      public int compare(Country country1, Country country2)
      {
         return country1.getName().compareTo(country2.getName());
      }
   };
```

The `new` expression:

- Defines a class with no name that implements the `Comparator<Country>` interface type.
- Has only one method, `compare`.
- Constructs one object of the class.

NOTE An anonymous class is a special case of an *inner class*. An inner class is a class that is defined inside another class.

TIP Most programmers find it easier to learn about anonymous classes by rewriting the code and explicitly introducing a class name. For example:

```
class MyComparator implements Comparator<Country> // Give a name to the class
{
   public int compare(Country country1, Country country2)
   {
      return country1.getName().compareTo(country2.getName());
   }
}
Comparator<Country> comp = new MyComparator();
```

After you have gained experience with anonymous classes, they will become quite natural, and you will find that you no longer need to rewrite the code.

Anonymous classes are very useful because they relieve you from the drudgery of having to name and document classes that are merely of a technical nature. Unfortunately, the syntax is rather cryptic. You have to look closely at the call `new` to find out that it constructs an object of an anonymous class.

The opening brace after the constructor parameter

```
new Comparator<Country>() { . . . }
```

shows that a new class is being defined.

Of course, in this situation, you know that `new Comparator<Country>()` couldn't have been a regular constructor call—`Comparator<Country>` is an interface type and you can't construct instances of an interface type.

Note the semicolon after the closing brace of the anonymous class definition. It is part of the statement

```
Comparator<Country> comp = an object;
```

TIP To make anonymous classes easier to read, you should start the anonymous class definition on a new line, like this:

```
Comparator<Country> comp = new // Break line here
   Comparator<Country>() // Indent one tab stop
   {
      . . .
   };
```

The line break after the `new` keyword tips you off that something special is going to happen. Furthermore, the interface type name lines up nicely with the braces surrounding the definitions of the features of the anonymous class.

NOTE Anonymous classes look tricky when first encountered. However, they are a programming idiom that has become extremely popular with professional Java programmers. You will encounter anonymous classes frequently when looking at professional Java code, and it is important that you spend time mastering the idiom. Fortunately, with a little practice, it quickly becomes second nature to most programmers.

In our first example, we made a single short-lived object of the anonymous class, making it truly into a "throwaway" class. But it is easy to create multiple objects of the anonymous class, simply by putting the construction inside a method. For example, the `Country` class can have a static method that returns a comparator object that compares countries by name:

```
public class Country
{
   . . .
   public static Comparator<Country> comparatorByName()
   {
      return new
         Comparator<Country>() // Make object of anonymous class
         {
            public int compare(Country country1, Country country2)
            {
               return country1.getName().compareTo(country2.getName());
            }
         };
   }
   . . .
}
```

You can now sort an array list of countries like this:

```
Collections.sort(countries, Country.comparatorByName());
```

Actually, for a class that doesn't have one natural ordering, this is a very nice setup, much better than implementing the `Comparable` interface type. Rather than defining a `compareTo` method that sorts rather arbitrarily by area or name, the `Country` class can define two methods that return `Comparator` objects.