List of Slides

Case study 1

Elephant joke teller

Introduction

- This handout briefly covers a small case study which is related to graphs.
  - An elephant joke telling program: `Elephant`.

Motivation

- Part of the curious humour of elephant jokes is that they are not funny.
  - For example:

```
Q: Why do elephants wear sandals in the desert?
A: So they don't sink in the sand.
```

- If there is any humour, it comes from the **dependencies** between elephant jokes.
  - For example:

```
Q: Why do osteriches stick their head in the sand?
A: They're looking for elephants
   that were not wearing sandals.
```

- Like all humour, nobody can explain why *some* people find them funny.
- But any *chance* of a laugh is ruined if the jokes are told in the wrong order!

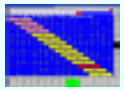
The final program

- The program is available in `/opt/teaching/bin/Elephant`.
- Run:  /opt/teaching/bin/Elephant
  Click here for ‘humour’

The classes and interfaces of the program

Class / Interface	Purpose
Elephant	The main class.
Fold	To fit text into a given width.
GraphNode	The full version of the general directed graph.
GraphNodeNameMap	To find graph nodes given their name.
InstanceCounter	To keep track of how many joke tellers.
Joke	The actual jokes.
JokeTeller	A GUI to tell the jokes.
RandomCounter	Used in producing random topological orders.

Elephant jokes are graph nodes

- A set of elephant jokes is a graph.
- There are typically several chains of jokes about one theme that must be told in the correct order.
- Sometimes the themes join up.
- And then there are several stand-alone jokes, some of which are red herrings, and some of which are (almost) funny in their own right.
- Graph demo:  the-jokes
  - Change the order to random!

Elephant joke implementation

- So an elephant joke is a graph node.
- We can implement it as a sub-class of `GraphNode`.

Joke code

```
public class Joke extends GraphNode
{
  private String question, answer;

  public Joke(String name, String requiredQuestion,
                           String requiredAnswer)
  {
    super(name); // What does this do?
    question = requiredQuestion;
    answer = requiredAnswer;
  } // Joke
```

Methods `getQuestion()` and `getAnswer()`

```
  public String getQuestion()
  {
    return question;
  } // getQuestion

  public String getAnswer()
  {
    return answer;
  } // getAnswer

} // class Joke
```

Class `GraphNodeNameMap`

- We need a way to be able to find a graph node from its name.
- The file containing the joke data has a name for each joke, followed by its question, its answer, and then the names of the jokes upon which it depends.
- To build the graph of jokes when we load the file we need to be able to find the jokes from their name so we can add the dependencies between jokes.
- The class `GraphNodeNameMap` handles this for us, and it is based on a `TreeMap`.

Class GraphNodeNameMap

```
import java.io.*;
import java.util.*;

public class GraphNodeNameMap
{
  // Map from name to GraphNode.
  // TreeMap means the names can be retrieved in alphabetic order.
  private Map nameMap = new TreeMap();

  // Temporary store of dependent / dependency pairs.
  private ArrayList dependents = new ArrayList();
  private ArrayList dependencies = new ArrayList();
```

Constructor

```
public GraphNodeNameMap()
{
} // GraphNodeNameMap
```

Methods addNode() and getNode()

```
public void addNode(String name, GraphNode graphNode)
{
  nameMap.put(name, graphNode);
} // addNode

public GraphNode getNode(String name)
{
  return (GraphNode) nameMap.get(name);
} // getNode
```

Methods nodeNames( ) and nodes( )

```
public Iterator nodeNames()
{
  return nameMap.keySet().iterator();
} // nodeNames

public Iterator nodes()
{
  return nameMap.values().iterator();
} // nodes
```

Method `removeNode()`

```
public void removeNode(String nodeName)
{
  nameMap.remove(nodeName);
} // remove
```

Method recordDependency()

```
// Record a dependency between two named nodes,
// to be added to the graph later.
public void recordDependency(String dependent, String dependency)
{
  dependents.add(dependent);
  dependencies.add(dependency);
} // recordDependency
```

Method `plantDependencies()`

```
  public void plantDependencies()
  {
    for (int i = 0; i < dependents.size(); i++)
    {
      GraphNode dependent
        = (GraphNode) nameMap.get(dependents.get(i));
      GraphNode dependency
        = (GraphNode) nameMap.get(dependencies.get(i));
      if (dependent != null && dependency != null)
        dependent.addDependency(dependency);
    } // for
  } // plantDependencies

} // class GraphNodeNameMap
```

Class `JokeTeller`

- We need a user interface in which the user can ask for the next joke, and then ask for its answer. This is provided by the class `JokeTeller`.
- A `JokeTeller` is given the end joke of the set of jokes. This is the end node of the graph, and will be the last 'joke' told.
- The `JokeTeller` obtains a topological order of the jokes, from the end joke, and then presents the jokes in that order when requested by the user. The topological order processes dependencies of each node in a random order, so the jokes are told in a random topological order.
- A `JokeTeller` has a `Clone` button which causes it to make a copy of itself. Each copy will tell all the jokes, as they each obtain their own topological order from the graph.
- The program exits only when all the `JokeTeller` instances have been quit.

Making the program exit

- Prior to Java 1.4, making the program exit when the last window is closed was non-trivial.
- Since Java 1.4, if each window is disposed on close, then the GUI event thread *should* end when the last one is disposed.
- This will cause the program to terminate if there are no other threads left running at that time.
- However, it is worth looking at what was needed prior to Java 1.4. We needed to manage the program exit ourselves.
- The approach may still be useful in some circumstances (e.g. when there are other threads running too).

Interface `InstanceCounter`

- Something keeps track of the number of instances of `JokeTeller`.
- Here, this is an `InstanceCounter`.
- When a `JokeTeller` is created, it will be passed a reference to the instance counter. It will then cause the instance counter to increment the count.
- When a `JokeTeller` is quit, it will cause the instance counter to decrement the count.
- One way of achieving this instance counting is to have it defined as an **interface**. We make the main class of the program (`Elephant`) `implement` this interface, so it can then act as the instance counter for all the joke tellers.
- The `InstanceCounter` mechanism is also used to make each `JokeTeller` appear at a different place on the screen.

InstanceCounter code

```
public interface InstanceCounter
{
  public void increment();

  public void decrement();

  public int getCurrentCount();

} // interface InstanceCounter
```

JokeTeller code (part)

```
// N.B: This uses AWT directly, rather that Swing.
import java.awt.*;
import java.awt.event.*;
import java.util.*;

public class JokeTeller extends Frame
{
  ...
  private Panel buttonPanel = new Panel();

  private Joke endJoke;
  private InstanceCounter instanceCounter;
  private Button askJoke;
  private Iterator topologicalOrder;
  ...
```

JokeTeller code (part)

```
public JokeTeller(Joke requiredEndJoke,
                  InstanceCounter requiredInstanceCounter)
{
  endJoke = requiredEndJoke;
  instanceCounter = requiredInstanceCounter;
  instanceCounter.increment();

  topologicalOrder
    = endJoke.topologicalOrder(GraphNode.ORDER_RANDOM,
                               false);
  ...
```

JokeTeller code (part)

```
    Button cloneButton = new Button("Clone");
    buttonPanel.add(cloneButton);
    cloneButton
      .addActionListener
         (new ActionListener()
          { public void actionPerformed(ActionEvent e)
              { new JokeTeller(endJoke, instanceCounter); }
          });
    ...

    setLocation(instanceCounter.getCurrentCount() * 40,
                instanceCounter.getCurrentCount() * 30);
    pack();
    show();
  } // JokeTeller
```

JokeTeller code (part)

```
// Called when windowClosing or when Quit button pressed.
private void endThisJokeTeller()
{
  dispose();
  instanceCounter.decrement();
} // endThisJokeTeller
```

JokeTeller code (part)

```
// This alternates between null and the current joke.
private Joke currentJoke = null;

// Called when ask joke / tell answer button pressed.
private void tellJoke()
{
  if (currentJoke == null)
  { if (topologicalOrder.hasNext())
    { currentJoke = (Joke) topologicalOrder.next();
      questionTextArea.setText
        (Fold.fold(0, 30, currentJoke.getQuestion()));
      answerTextArea.setText("");
      askJoke.setLabel(TELL_ANSWER_LABEL);
    } // if
  } // if
```

JokeTeller code (part)

```
    else
    {
      answerTextArea.setText
        (Fold.fold(0, 30, currentJoke.getAnswer()));
      currentJoke = null;
      askJoke.setLabel(ASK_JOKE_LABEL);
    } // else
  } // tellJoke

} // class JokeTeller
```

Elephant code (part)

```
public class Elephant implements InstanceCounter
{
  public static void main(String [] args)
  { GraphNodeNameMap jokeNameTable
      = new GraphNodeNameMap();
    Joke lastJoke
      = new Joke("LAST_JOKE",
                 "How do you know when you have heard"
                 + " all the elephant jokes?",
                 "There are no more.");
```

... Code here to load the jokes from a file, and build the graph, with `lastJoke` as the end node.

```
    InstanceCounter ic = new Elephant();
    new JokeTeller(lastJoke, ic);
  } // main
```

Elephant code (part)

```
private static int jokeTellerCount = 0;

public void increment()
{
  jokeTellerCount++;
} // increment

public void decrement()
{
  jokeTellerCount--;
  if (jokeTellerCount == 0)
    System.exit(0);
} // decrement
```

Elephant code (part)

```
  public int getCurrentCount()
  {
    return jokeTellerCount;
  } // getCurrentCount

} // class Elephant
```

Class `Fold`

- When displaying the text of a joke question or answer, we need to make it fit in the space available in the text boxes of the `JokeTeller`.
- The class `Fold` provides a single static method `fold()` which takes a `String` and integer margin and width arguments, and produces a `String` consisting of several lines which will fit in the given width and margin (i.e. the result has new line characters in it).

Fold code

```
import java.util.StringTokenizer;

public class Fold
{
  public static String fold(int margin, int width,
                            String text)
  {
    StringBuffer marginStringBuffer = new StringBuffer();
    while (marginStringBuffer.length() < margin)
      marginStringBuffer.append(" ");
    String marginString = marginStringBuffer.toString();
```

Fold code

```
StringBuffer result = new StringBuffer();
int currentLineLength = 0;
StringTokenizer linesTokenizer
  = new StringTokenizer(text, "\n", false);
```

Fold code

```
while (linesTokenizer.hasMoreTokens())
{
  StringTokenizer wordsTokenizer
    = new StringTokenizer(linesTokenizer.nextToken());
  while (wordsTokenizer.hasMoreTokens())
  {
    String word = wordsTokenizer.nextToken();
    if (currentLineLength == 0)
    {
      result.append(marginString);
      currentLineLength = margin;
    } // if
    else
```

Fold code

```
{
  if (currentLineLength + 1 + word.length() > width)
  {
    result.append("\n");
    result.append(marginString);
    currentLineLength = margin;
  } // if
  else
  {
    result.append(" ");
    currentLineLength++;
  } // else
} // else
```

Fold code

```
        result.append(word);
        currentLineLength += word.length();
      } // while
    } // while
    result.append("\n");
    return result.toString();
  } // fold

} // class Fold
```

Class `RandomCounter`

- The `RandomCounter` class is used to help make random topological orders. When creating the topological order, we traverse the dependencies of a node in a random order.
- A `RandomCounter` object is given a positive number, `size` when it is created. Then, each time its method `next()` is called, it will return a number between 0 and `size - 1`, without repetition, but in a random order. If it is called more than `size` times, it returns `-1` from then on.
- To scan the list of dependencies of a graph node in a random order, we simply create a `RandomCounter`, giving the size of the dependencies list as the argument, and then call `next()` that many times, using the result as the index of the next dependency to consider.

RandomCounter code

```
public class RandomCounter
{
  private int [] countList;
  private int remainingCountListLength;

  // Obtains count from 0 to size -1, but in random order.
  public RandomCounter(int size)
  {
    remainingCountListLength = size;
    countList = new int[size];
    for (int i = 0; i < size; i++)
      countList[i] = i;
  } // RandomCounter
```

RandomCounter code

```
  // Returns -1 when all the list is exhausted.
  public int next()
  { if (remainingCountListLength == 0)
      return -1;
    else
    { int randomInt
        = (int) (Math.random() * remainingCountListLength);
      int result = countList[randomInt];
      remainingCountListLength--;
      countList[randomInt]
        = countList[remainingCountListLength];
      return result;
    } // else
  } // next

} // class RandomCounter
```