# Software Development 2

Further Swing

F27SB

Awesome GUI stuff

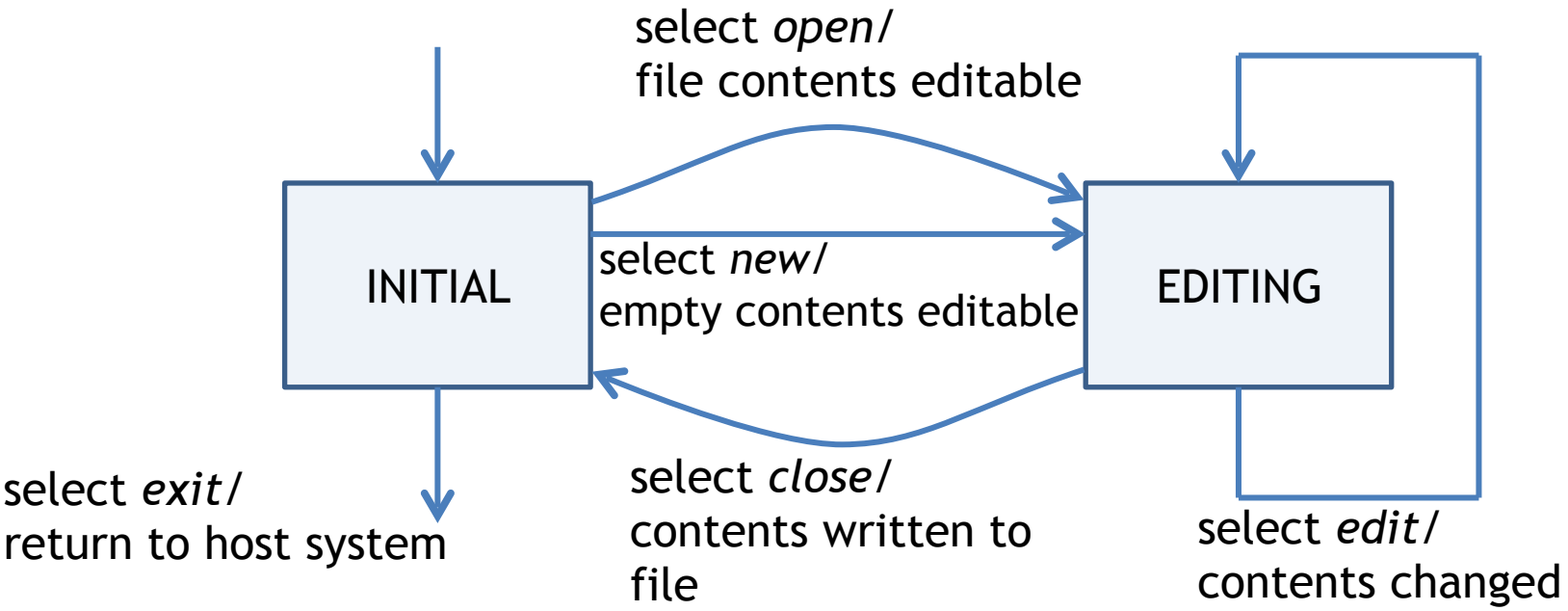
# TODAY'S LECTURE

# Today's Lecture

- Other GUI components
  - `JTextArea`
  - `JScrollPane`
  - `JMenu`
  - `JFileChooser`
  - `JOptionPane`

# Overview

- Rolling example of a text editor:



# Overview

- Will focus on methods to:
  - display/manipulate scrollable text
  - select options from menu bars
  - open/save files from directory
  - conduct simple standard dialogs

# Editable text

`JTextArea extends javax.swing.text.JTextComponent`

- multiple lines of plain, unformatted text
- user can edit text

`JTextArea(String` *`text`*`, int` *`rows`*`, int` *`columns`*`)`

- *`text`*: text to be edited
- *`rows`*: number of displayed rows
- *`columns`*: number of displayed columns

# Editable text

`public String setText(String text)`

- replaces all text with *`text`*

`public String append(String text)`

- adds *`text`* at end

`public String getText()`

- returns text as single `String`

`JTextArea` supports drag and drop

- but no right button menu

# Editable text

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

// Define class: a window that contains a text area
class Edit11 extends JFrame {

   JTextArea ta; // text display area

   Edit() {
      ta = new JTextArea("",24,80); // create

      add(BorderLayout.CENTER,ta);  // and add to window
   }
}
```

# Editable text

```
class TestEdit {
   public static void main(String [] args)
         throws IOException {
      Edit11 e = new Edit11(); // create window with text area

      // setup window
      e.setTitle("edit1");
      e.setSize(400,200);
      e.setVisible(true);
      e.addWindowListener
        (new WindowAdapter()
         {  public void
             windowClosing(WindowEvent e)
            {  System.exit(0); }
            });
```

# Editable text

```
        // read text from file a line at a time
        // note: in practice, this wouldn't be in main!
        BufferedReader fin;
        fin = new BufferedReader
                (new FileReader("test.txt"));
        String next = fin.readLine();
        while(next!=null) {
           e.ta.append(next);
           next = fin.readLine();
        }
    }
}
```

# Editable text

test.txt:

Once upon a time there were three little computers called Boris, Doris and Horace. One day Boris was having difficulty opening a file.

"I just can't open this file!" said Boris sadly. "Maybe there's something wrong with my H: drive."

"Maybe it's a network problem," said Doris. "I can't access the printer."

"And I can't get onto the Internet," said Horace.

Just then, along came the Nice Computer Manager.

# Editable text

"Hello computers!" said the Nice Computer Manager.

"Hello Nice Computer Manager!" said the computers.

"How are you all today?" asked the Nice Computer Manager.

"I can't open a file," said Boris, sadly.

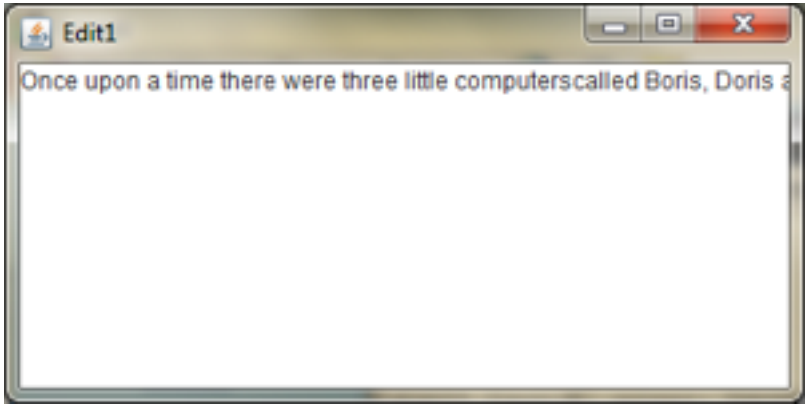
"And I can't access the printer," said Doris, glumly.

"And I can't get onto the Internet," said Horace, gloomily.

"Don't worry," said the Nice Computer Manager. "You're all being replaced by a shiny new laptop!"

The End.

# Line wrap

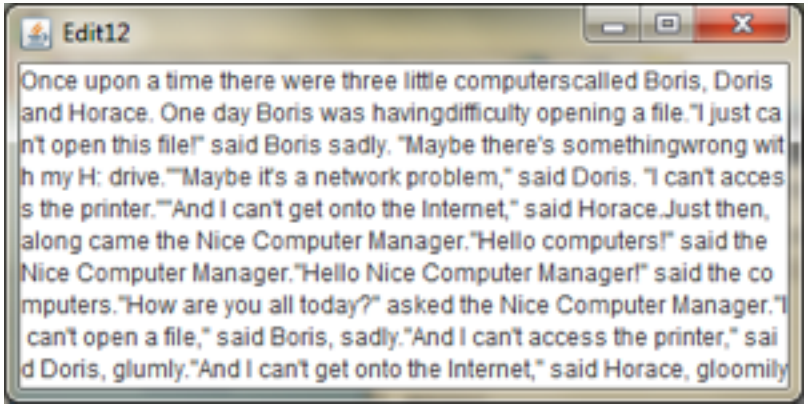



Currently all the text is on one line

- need to turn on line wrapping
- `public void setLineWrap(Boolean wrap)`

# Line wrap

```
Edit() {
    ta = new JTextArea("",24,80);
    ta.setLineWrap(true);

    add(BorderLayout.CENTER,ta);
}
```

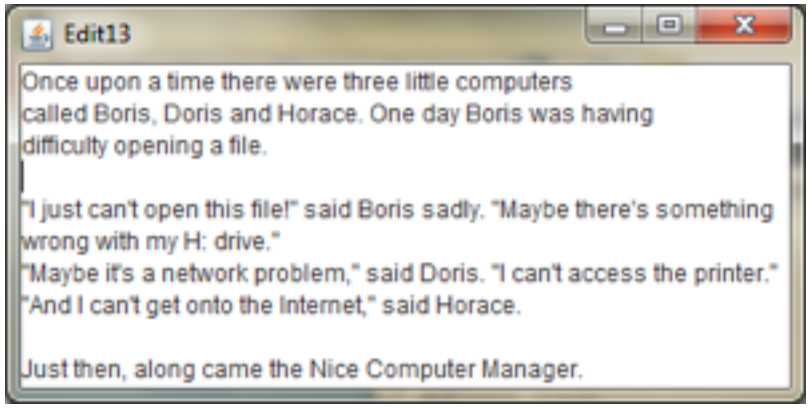


# Line termination

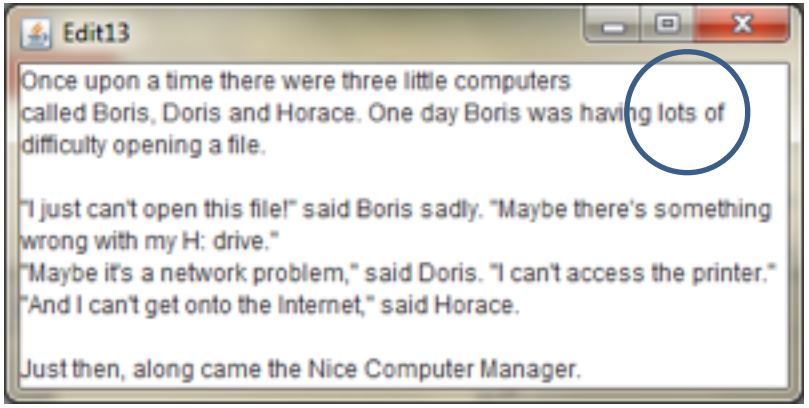
- also need to add line termination

```
 public static void main(String [] args)
       throws IOException {
   Edit13 e = new Edit13();
   ...
   String next = fin.readLine();
   while(next!=null)
   {
      e.ta.append(next+"\n");
      next = fin.readLine();
   }
}
```

# Line termination




- can add/delete text using the mouse & keyboard



# SCROLL PANE

# Scrolling

However, text doesn't scroll, so also need a…

```
JScrollPane extends JComponent
   implements Accessible, ScrollPaneConstants

public JScrollPane()
public JScrollPane(Component view)
```

- constructs a `JScrollPane` container for `Component` *`view`*
- allows horizontal & vertical scrolling

# Scrolling

Can change presence of scroll bars with:

`setHorizontalScrollBarPolicy(int` *`policy`*`)`

`setVerticalScrollBarPolicy(int` *`policy`*`)`

- where *policy* is a `ScrollPaneConstants`:
  - `HORIZONTAL_SCROLLBAR_` or
  - `VERTICAL_SCROLLBAR_` followed by:
  - `ALWAYS` `AS_NEEDED` or `NEVER`
- e.g. `setHorizontalScrollBarPolicy(`
  `ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER)`

# Scrolling

Add a `Component` to a scroll pane using:

- `public void setViewportView(Component view)`
- Don't use `add()`
  - This will compile, but won't work as expected!

# Scrolling

Change `Edit` to provide vertical scrolling

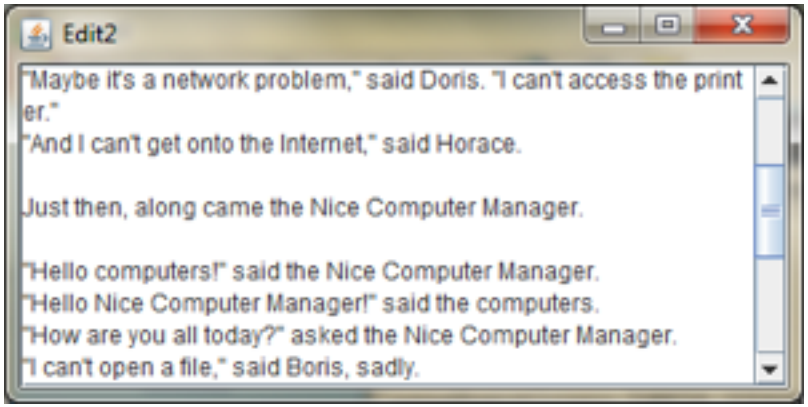
- have wrap round so don't need horizontal scrolling
- horizontal scrolling is poor for text manipulation

# Scrolling

```
...
class Edit extends JFrame {
   JTextArea ta;   // the text area
   JScrollPane sp; // the text area's scroll pane

   Edit2() {
      sp = new JScrollPane();
      // turn off the horizontal scroll bar
      // use default policy for vertical: appears as needed
      sp.setHorizontalScrollBarPolicy
         (ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER);
      ta = new JTextArea("",24,80);
      ta.setLineWrap(true);
      sp.setViewportView(ta); // set view to be the scroll pane
      add(BorderLayout.CENTER,sp);
   }
}
```

# Scrolling

```
class TestEdit {
   static BufferedReader fin;

   public static void main(String [] args) throws IOException
   {
        Edit e = new Edit();
        ...
   }
}
```



# Advanced [not examinable]

Java also supports formatted text areas

- `JEditorPane` supports HTML
- `JTextPane` is more like a word processor
- http://docs.oracle.com/javase/tutorial/uiswing/components/editorpane.html

# MENUS

# Menus

Windows interfaces provide menu bars with:

- selectable pop-up/pull-down menus
- of selectable menu items

Advantages over buttons

- more space efficient
- show available functionality in one place
- standardised menu names and menu items
  - e.g. File, Edit, Copy, Paste, …

# Menus

There's a Swing class for menu bars:

```
JMenuBar extends JComponent
      implements MenuElement
```

`JMenuBar()`

- creates a new `JMenuBar`

`setJMenuBar(JMenuBar` *`menubar`*`)`

- places *`menubar`* at the top of a `JFrame`

# Menus

Also a Swing class for menus:

```
JMenu extends JMenuItem
      implements MenuElement
```

`JMenu(String s)`

- creates a new `JMenu` identified by `s`

*`jmenubar`*`.add(`*`jmenu`*`)`

- adds *`jmenu`* to *`jmenubar`*

# Menus

And a Swing class for menu items:

```
JMenuItem extends AbstractButton
      implements MenuElement
```

`JMenuItem(String s)`

- creates a new `JMenu` identified by `s`

*`jmenu`*`.add(`*`jmenuitem`*`)`

- Adds menu item *`jmenuitem`* to *`jmenu`*

*`jmenu`*`.add(`*`submenu`*`)`

- Adds menu *`submenu`* to *`jmenu`*

# Menus

Add menu bar to editor with a menu for *File* with options for *Open*, *New*, *Close* and *Exit*

```
...
class Edit extends JFrame {
   JScrollPane sp;
   JTextArea ta;

   JMenuBar jb; // the menu bar
   JMenu file; // the file menu
   JMenuItem MNew, MOpen, MClose, MExit; // the menu items
```

# Menus

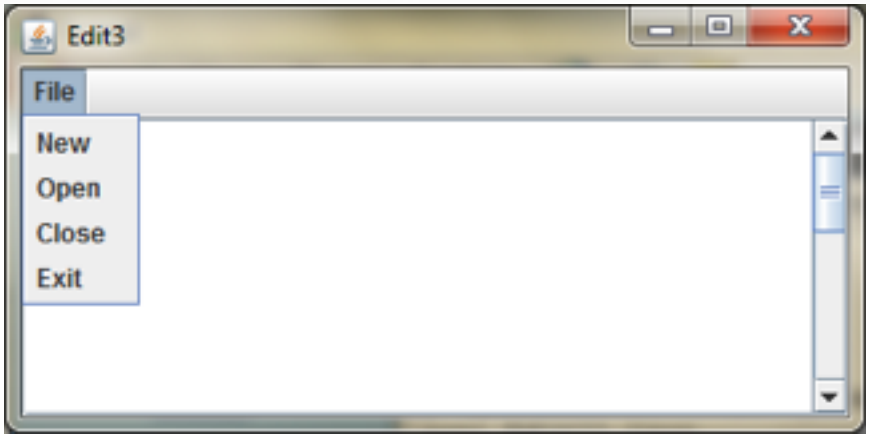
```
Edit() {
    sp = new JScrollPane();
    sp.setHorizontalScrollBarPolicy
     (ScrollPaneConstants.
      HORIZONTAL_SCROLLBAR_NEVER);

    ta = new JTextArea("",24,80);
    ta.setLineWrap(true);

    sp.setViewportView(ta);
    add(BorderLayout.CENTER,sp);

    jb = new JMenuBar(); // create menu bar
    file = new JMenu("File"); // create named menu
```

# Menus

```
        // create menu items, specifying names
        MNew = new JMenuItem("New");
        MOpen = new JMenuItem("Open");
        MClose = new JMenuItem("Close");
        MExit = new JMenuItem("Exit");

        // add menu items to file menu
        file.add(MNew);
        file.add(MOpen);
        file.add(MClose);
        file.add(MExit);

        jb.add(file); // add file menu to menu bar

        setJMenuBar(jb); // display menu bar (in current window)
    }
}
```

# Menus

```
class TestEdit {
   static BufferedReader fin;

   public static void main(String [] args) throws IOException {
      Edit e = new Edit();
      ...
   }
}
```



# Menu

Now need to associate actions with the options

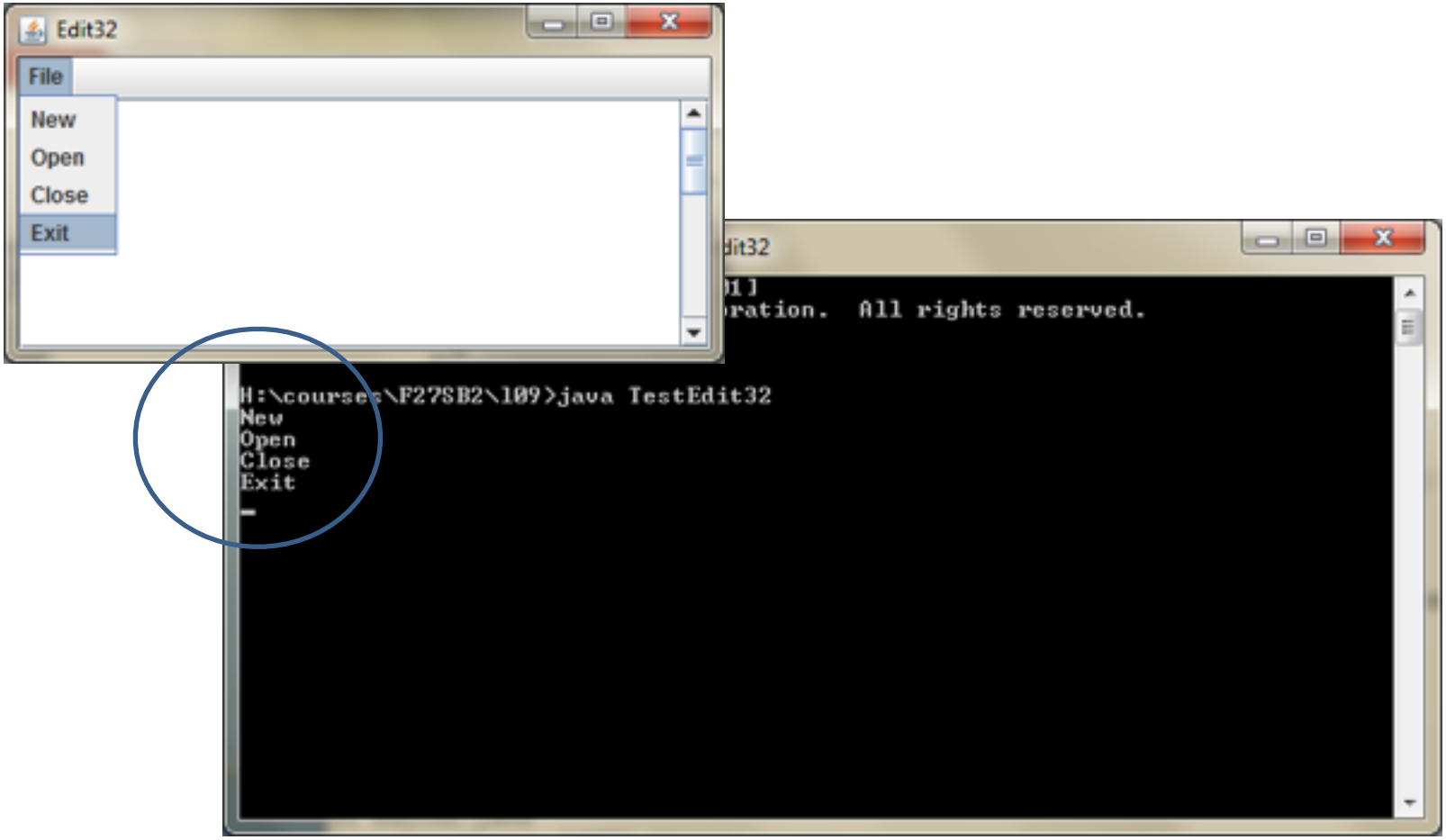
- `JMenuItem` **causes** `ActionEvent`

```
class Edit extends JFrame implements ActionListener {
   ...
   Edit() {
      ...
      // add action listener to menu items, as with buttons
      MNew.addActionListener(this);
      MOpen.addActionListener(this);
      MClose.addActionListener(this);
      MExit.addActionListener(this);
      ...
   }
```

# Menu

```
   // process action events in the same way as buttons
   // just print out selected menu item's name for now
   public void actionPerformed(ActionEvent e)
   {
      if(e.getSource()==MNew)
         System.out.println("New");
      else if(e.getSource()==MOpen)
         System.out.println("Open");
      else if(e.getSource()==MClose)
         System.out.println("Close");
      else if(e.getSource()==MExit)
         System.out.println("Exit");
   }
}

class TestEdit{  ...  }
```

# Menu


Edit32
File
New
Open
Close
Exit
ration. All rights reserved.
H:\course\F27SB2\109>java TestEdit32
New
Open
Close
Exit

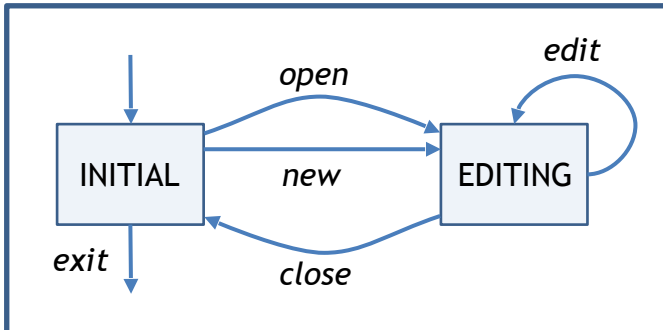
# State machine

Next, develop state machine in `actionPerformed`

- two states, so can use a boolean state variable:

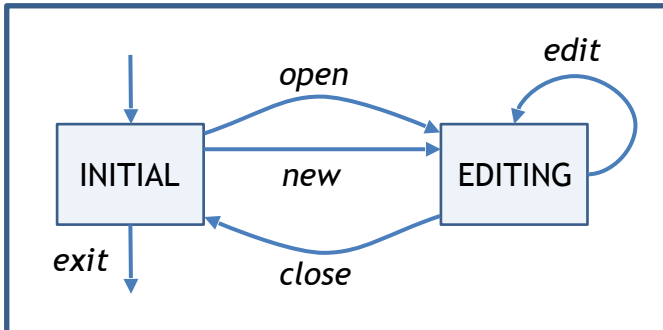
Also need to use `setEditable(boolean e)`

- changes `Component` editable status
- initially, not `editing`
  and text area is not editable



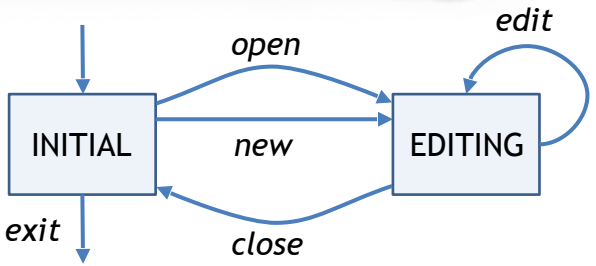
# State machine

```
class Edit extends JFrame implements ActionListener {
   ...
   boolean editing; // state variable: editing or not editing?
   ...

   Edit() {
      ...
      editing = false; // initially not in the editing state
      ta.setEditable(false); // so disable the text area
      ...
   }
```



# State machine

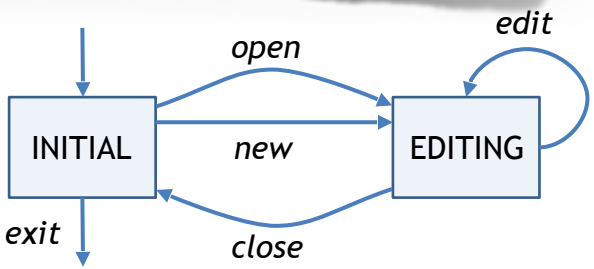
```
public void actionPerformed(ActionEvent e) {
   if(!editing) // check state before processing events
   {
      if(e.getSource() == MNew) {
         System.out.println("New");
         editing = true; // transition to editing state
         ta.setEditable(true);
      }
      else if(e.getSource() == MOpen) {
         System.out.println("Open");
         editing = true; // transition to editing state
         ta.setEditable(true);
      }
```



# State machine

```
        else if(e.getSource()==MExit)
             System.out.println("Exit");

      }
      else {// if in editing state
          if(e.getSource()==MClose)
          {
              System.out.println("Close");
              editing = false; // transition to non-editing state
              ta.setEditable(false);
          }
      }
   }
}
...
```

- next implement the actions…



# Advanced [not examinable]

- Menu separators

```
jmenu.addSeparator()
```

- Accelerators (i.e. keyboard shortcuts)

```
jmenuitem.setAccelerator(KeyStroke.getKeyStr
oke( KeyEvent.VK_1, ActionEvent.ALT_MASK));
```

- For further info, e.g. popup menus, see:http://docs.oracle.com/javase/tutorial/uiswing/components/menu.html

# Advanced [not examinable]

Mac OS handles menus a bit differently!

- first, check if this is a Mac

```
if(System.getProperty("os.name")
    .toLowerCase().startsWith("mac os x")) {
```

- if so, move the menu bar to the top of the screen

```
System.setProperty(
    "apple.laf.useScreenMenuBar", "true");
```

- and give the ‘application menu’ a proper name

```
System.setProperty("com.apple.mrj.applicatio
n.apple.menu.about.name", "Editor");
```

Never ever will I hard code a filename again!

# FILE CHOOSERS

# File chooser

We don't want to hard code the file paths, so use:

```
JFileChooser extends JComponent
             implements Accessible
```

- a Swing component providing a standard GUI for file system navigation

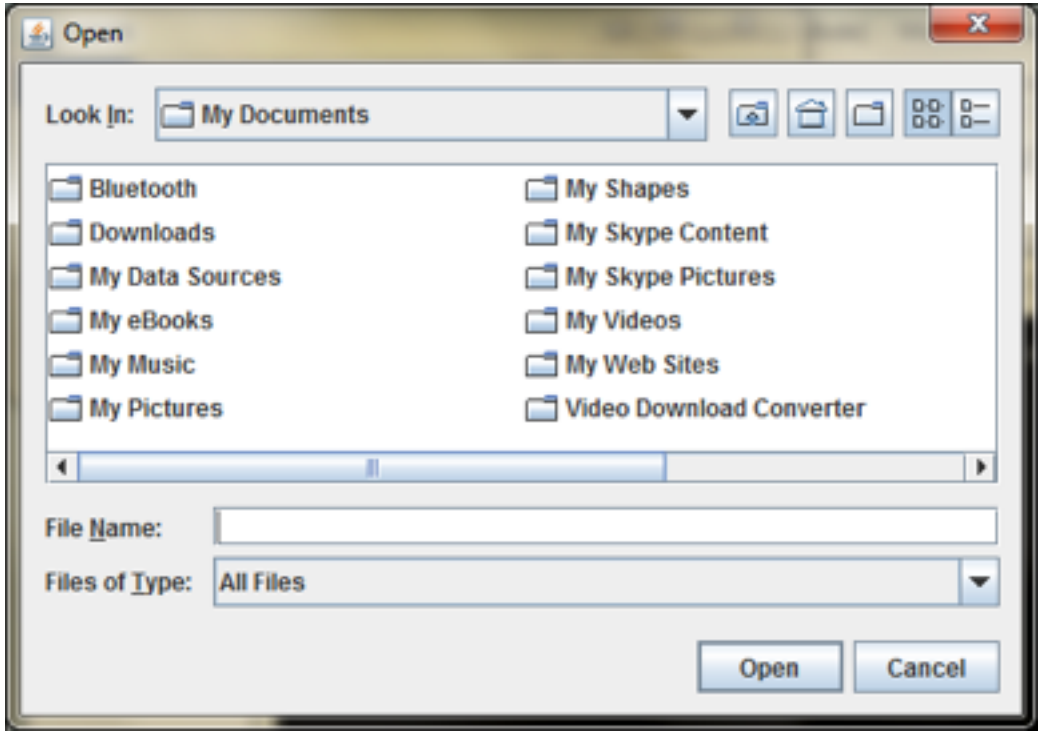
`JFileChooser()`

- create `JFileChooser` component for current directory
- Does **not** support `WindowListener`

# File chooser

`int showOpenDialog(Component parent)`

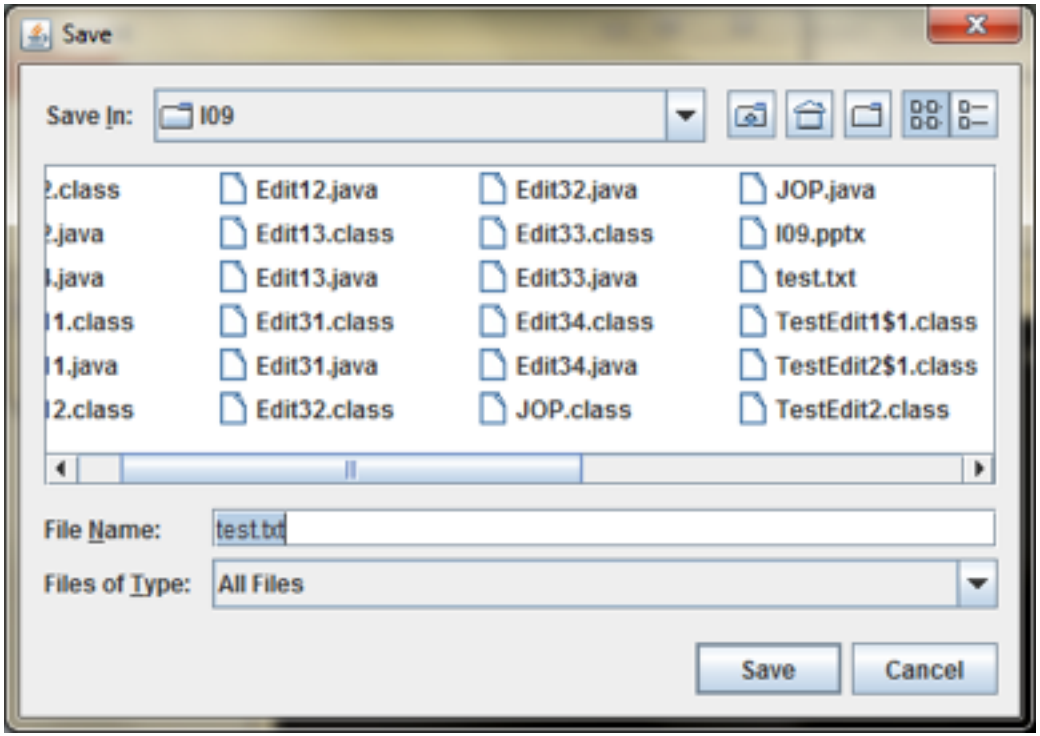
- shows the standard open file dialog box



# File chooser

`int showSaveDialog(Component parent)`

- shows the standard save file dialog box



# File chooser

- both return constants:
  - `APPROVE_OPTION` for *Open/Save* buttons
  - `CANCEL_OPTION` for *Cancel* button

```
File getSelectedFile()
```

- returns selected file from `JFileChooser`

# File chooser

```
// process action events from menu items
public void actionPerformed(ActionEvent e) {
   if(!editing) {
      if(e.getSource()==MNew)
         doNew();
      else if(e.getSource()==MOpen)
         doOpen();
      else if(e.getSource()==MExit)
         doExit();
   }
   else {
      if(e.getSource()==MClose)
         doClose();
   }
}
```

# File chooser

```
class Edit extends JFrame implements ActionListener {
   ...
   JFileChooser files;
   files = new JFileChooser(); // create the file chooser
   ...
   // respond to user selecting 'open' menu item
   public void doOpen() {
      try {
         // ask file chooser to show file open dialog
         int response = files.showOpenDialog(this);
         // check whether user pressed okay or cancel
         if(response==JFileChooser.APPROVE_OPTION) {
            // if they pressed okay, identify selected file
            File f = files.getSelectedFile();
```

# File chooser

```
        // then read file as usual...
        BufferedReader fin =
           new BufferedReader(new FileReader(f));

        String next = fin.readLine();
        while(next!=null)
        {  ta.append(next+"\n");
           next = fin.readLine();
        }
        fin.close();
        editing = true;
        ta.setEditable(true);
      }
    }
    catch(IOException e){};
}
```

# File chooser

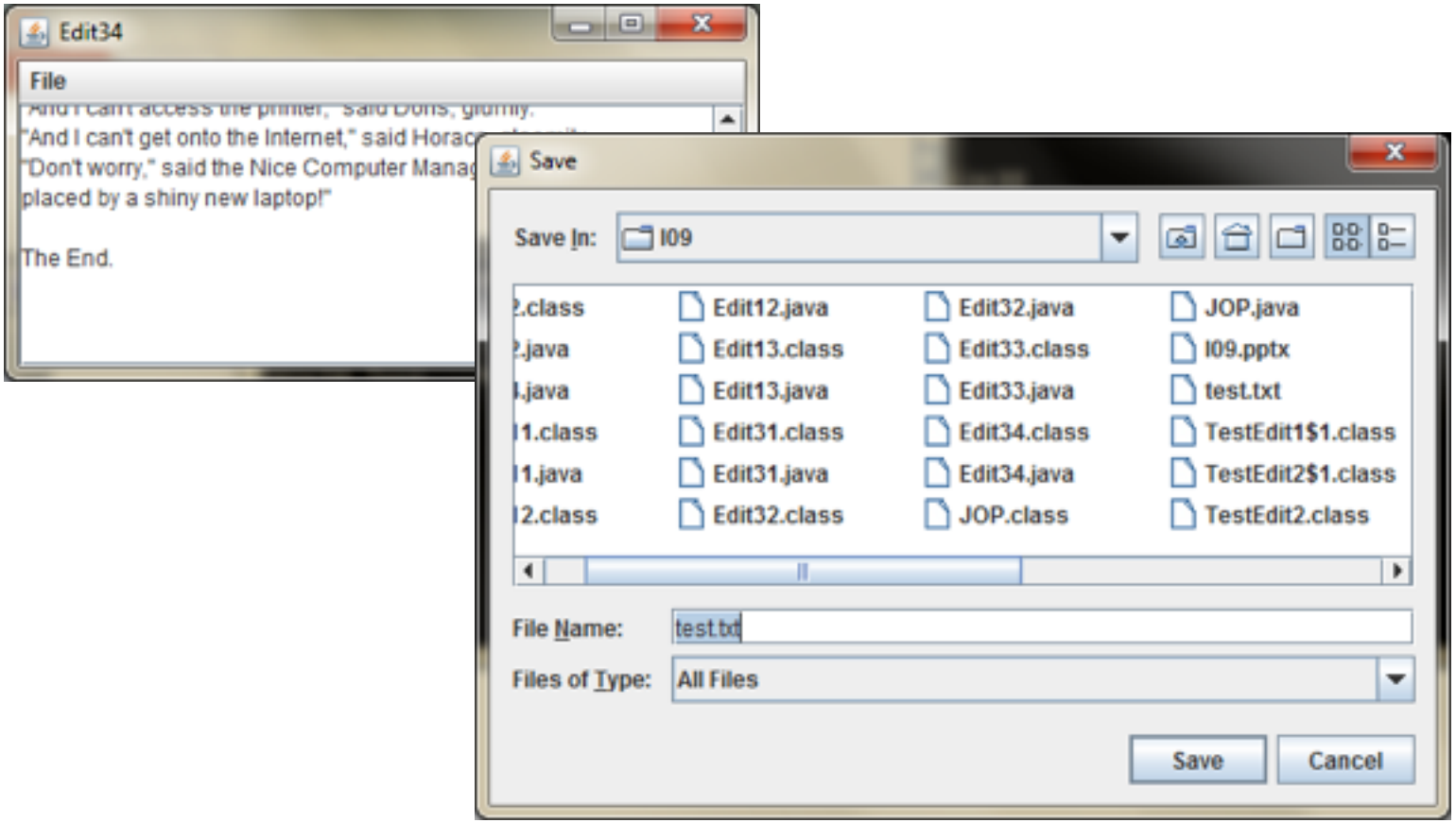
```
// respond to user selecting 'close' menu item
public void doClose() {
   try {
       // ask file chooser to show save file dialog
       int response = files.showSaveDialog(this);
       // if user pressed okay, identify chosen file
       if(response==JFileChooser.APPROVE_OPTION) {
          File f = files.getSelectedFile();
          BufferedWriter fout =
          new BufferedWriter(new FileWriter(f));
          fout.write(ta.getText());
          fout.close();
          editing = false;
          ta.setEditable(false);
       }
    }
    catch(IOException e){};
 }
```

# File chooser

```
// respond to user selecting 'exit' menu item
public void doExit() {
   System.exit(0);
}

// respond to user selecting 'new' menu item
public void doNew() {
   editing = true; // state transition
   ta.setEditable(true);
   ta.setText("");
}
```

# File chooser

```
class TestEdit34{  ...  }
```



‘Murican for dialogues

# DIALOGS

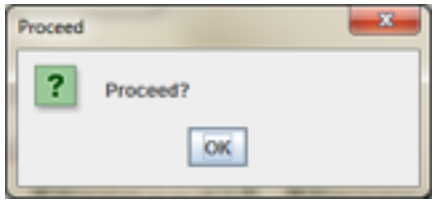
# Dialogs

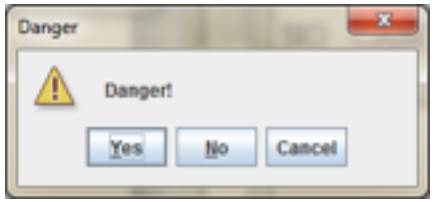
`JFileChooser` is an example of a *dialog*

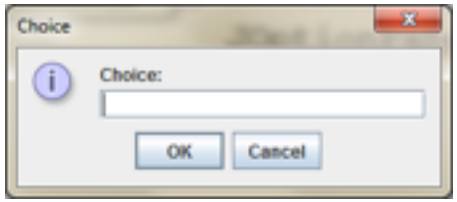
- A common restricted specialised frame

Some simpler examples are:

- **Message dialog**: a message with an *OK* button
- **Confirm dialog**: a message with a *YES*/*NO* option
- **Input dialog**: a request for simple text entry



# Dialogs

These dialogs can be created using:

```
JOptionPane extends JComponent
            implements Accessible
```

`JOptionPane()`

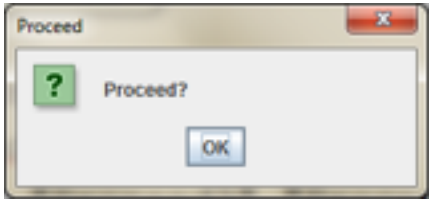
- constructor creates new empty dialog

However `new JOptionPane()` is rarely used

- instead, use static methods
- *`showXxxDialog()`*

# Dialogs

*Message* dialog:

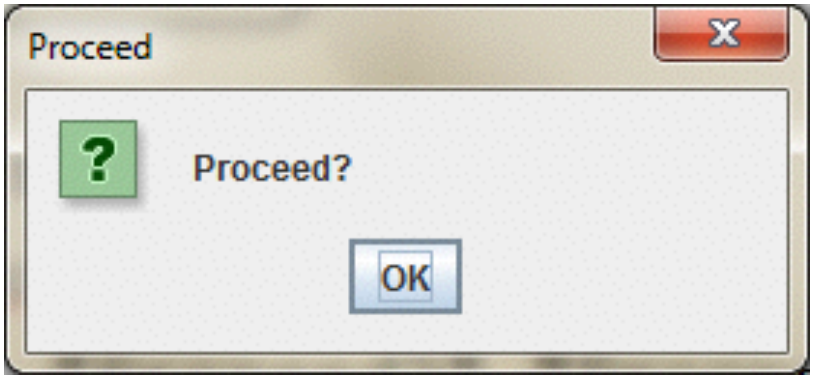



```
public static void showMessageDialog
(Component parent, Object message,
 [String title, int messagetype, Icon icon])
```

- *parent*: usually a frame, can be `null`
- *message*: string to display
- other arguments are optional
  - *messagetype*: `WARNING_MESSAGE, QUESTION_MESSAGE, INFORMATION_MESSAGE, ERROR_MESSAGE, PLAIN_MESSAGE`
  - *icon*: icon to display with message
- Supports `WindowListener`

# Dialogs

```
class JOptionPaneDemo extends JFrame {
   JOptionPaneDemo () {
      ...
      JOptionPane.showMessageDialog
       (this,"Proceed?","Proceed",
        JOptionPane.QUESTION_MESSAGE);
   }
```



# Dialogs

- Confirm dialog:


```
public static int showConfirmDialog
( Component parent, Object message,
  [String title, int optiontype,
  int messagetype, Icon icon])
```

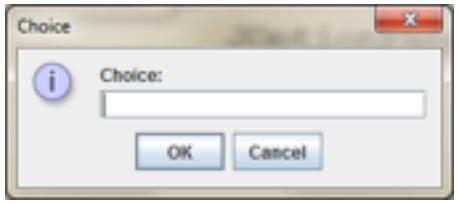
  - *optiontype*: `DEFAULT_OPTION (Okay), YES_NO_OPTION, YES_NO_CANCEL_OPTION, OK_CANCEL_OPTION`
  - returns: `OK_OPTION, CANCEL_OPTION, YES_OPTION, NO_OPTION, CLOSED_OPTION` (window closed without selection)

# Dialogs

```
...
JOptionPane.showConfirmDialog
 (this,"Danger!","Danger",
  JOptionPane.YES_NO_CANCEL_OPTION,
  JOptionPane.WARNING_MESSAGE);
...
```

# Dialogs

- Input dialog:

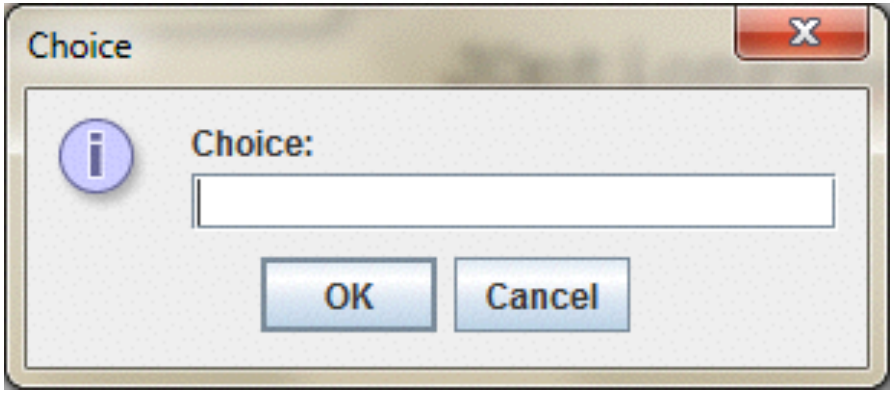



```
public static String showInputDialog
  (Component parent, Object message,
    [String title, int messagetype, ...])
```

- returns entered `String`

# Dialogs

```
...
String input =
 JOptionPane.showInputDialog(this,"Choice:","Choice",
                        JOptionPane.INFORMATION_MESSAGE);
...
```




- also a version with a pull down menu of options

# Dialogs

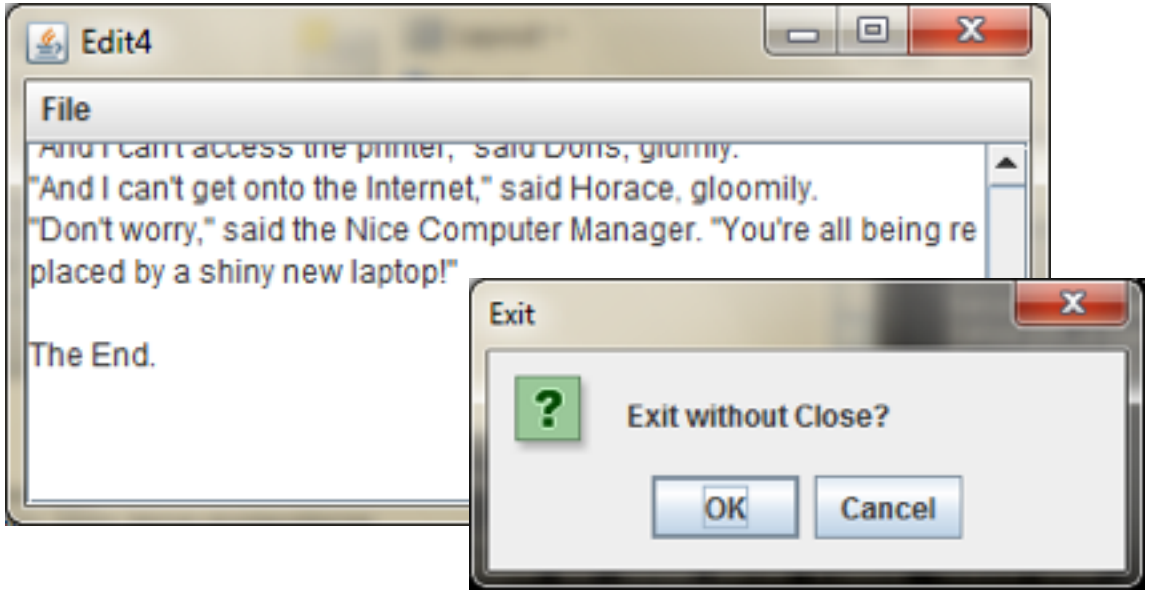
Let's modify the editor to allow exit without save

- but first ask the user if they want to save

```
...
class Edit extends JFrame implements ActionListener {
   ...
   public void doExit() {
      // show confirm dialog
      int response = JOptionPane.showConfirmDialog
                        (this,"Exit without Close?","Exit",
                         JOptionPane.OK_CANCEL_OPTION,
                         JOptionPane.QUESTION_MESSAGE);
      // only proceed if the user selected 'okay'
      if(response==JOptionPane.OK_OPTION)
         System.exit(0);
   } ...
}
```

# Dialogs

```
class TestEdit {
   public static void main(String [] args)
   {  Edit e = new Edit(); ... }
}
```



# THAT'S IT!

# Next Lecture

- Revision!