Software Development 2

State Diagrams

F27SB

Have your say

https://doodle.com/poll/pkv4w5kxwya9d89r

Quick Question

Room: F27SB

Do you do the tutorials?

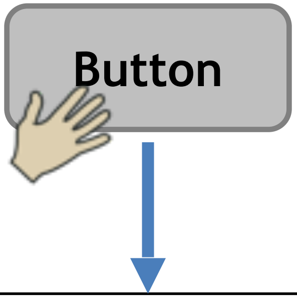
Please, be honest.
There are no names and
no negative implications
if you say no.

Today’s Lecture

- State transition diagrams
- Interactive system design

State and Interaction

- An interactive system can be seen as a sequence of *events* and associated *actions*
  - ***Event***: something that indicates the need for change, e.g. selecting a button
  - ***Action***: something that causes change, e.g. a listener method is invoked to respond to event



```
public void
    actionPerformed
            (ActionEvent e){
            // do action
    }
}
```

State and Interaction

- It is usually possible to identify distinct ***states***
- Where a state can be characterised as:
  - **a configuration**: the status of things that may change, such as variables and components
  - **valid events/actions** for current configuration
    - i.e. how change is indicated/what may be changed and how

State and Interaction

It is useful to think of the entire system as a set of states. System execution can then be seen as a series of ***state transitions***:

- Starting from a particular state
  1. event occurs
  2. action is triggered
  3. configuration is changed
  4. new state is entered

State and Interaction

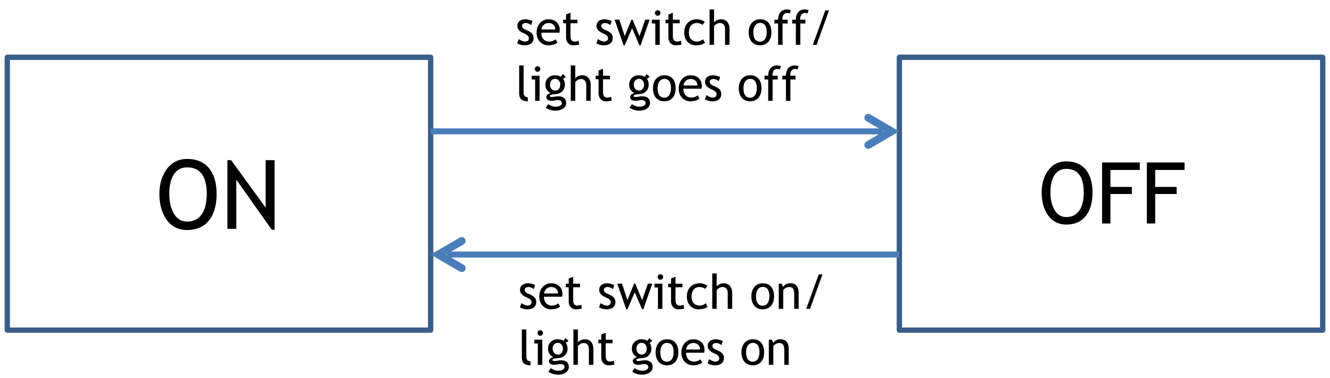
A simple example: a light circuit

- ON STATE
  - configuration – light is on
  - event – switch set to off
    - action – light goes off
    - now in OFF STATE

- OFF STATE
  - configuration – light is off
  - event – switch set to on
    - action – light goes on
    - now in ON STATE

State Transition Diagrams

A simple example: a light circuit



ON

OFF

set switch off/
light goes off

set switch on/
light goes on

State Transition Diagrams

Depict the states and transitions in a system

- **state**:
  - box with its name in it
- **transition**:
  - *arc* from a state to a state
  - labelled with "***event*** / ***action***"

- also known as *state charts*
- closely related to *finite state machines*

State and Interaction

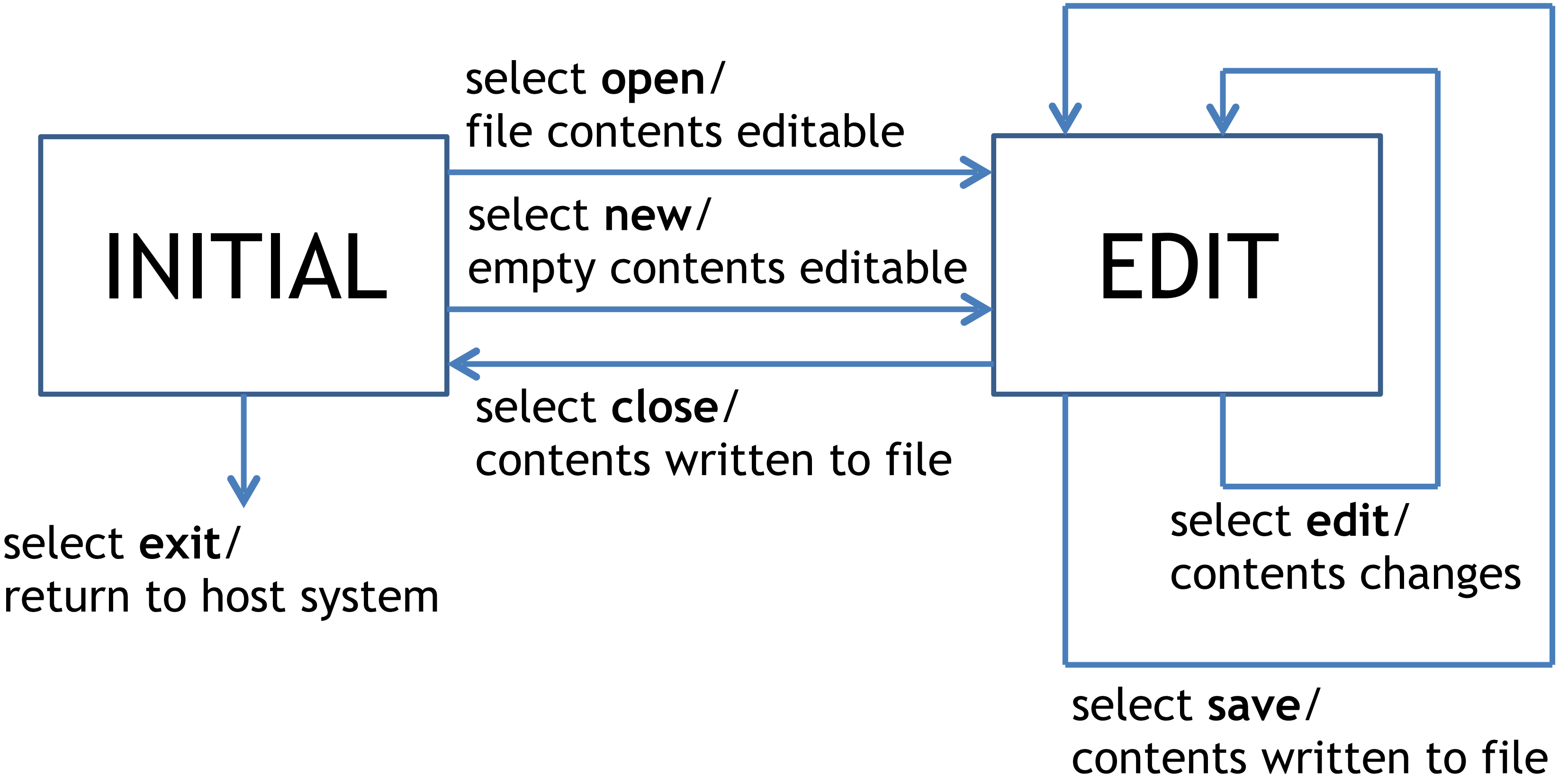
A more complex example: a text editor

- INITIAL STATE
  - configuration — nothing available to edit
  - event — select *open file*
    - action — contents from file available for editing
    - now in EDIT STATE
  - event — select *open new*
    - action — empty contents available for editing
    - now in EDIT STATE
  - event — select *exit*

State and Interaction

- EDIT STATE
  - configuration — file open for edit
  - event — select *edit operation*
    - action — file contents changed
    - now in EDIT STATE
  - event — select *save file*
    - action — contents copied back to file
    - now in EDIT STATE
  - event — select *close file*
    - action — contents copied back to file
    - now in INITIAL STATE

State Transition Diagrams



INITIAL

EDIT

select **open**/
file contents editable

select **new**/
empty contents editable

select **close**/
contents written to file

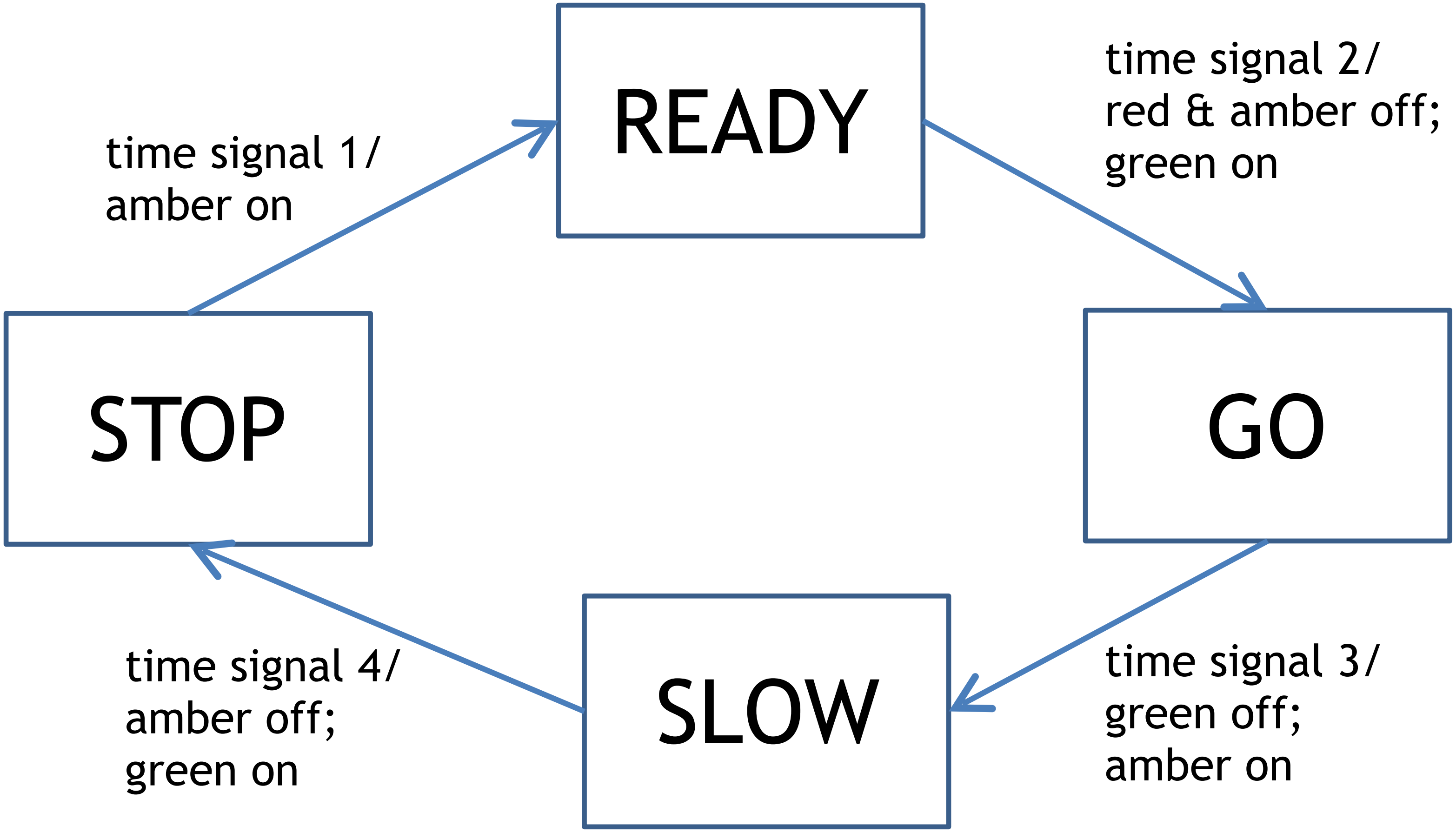
select **exit**/
return to host system

select **edit**/
contents changes

select **save**/
contents written to file

State Transition Diagrams

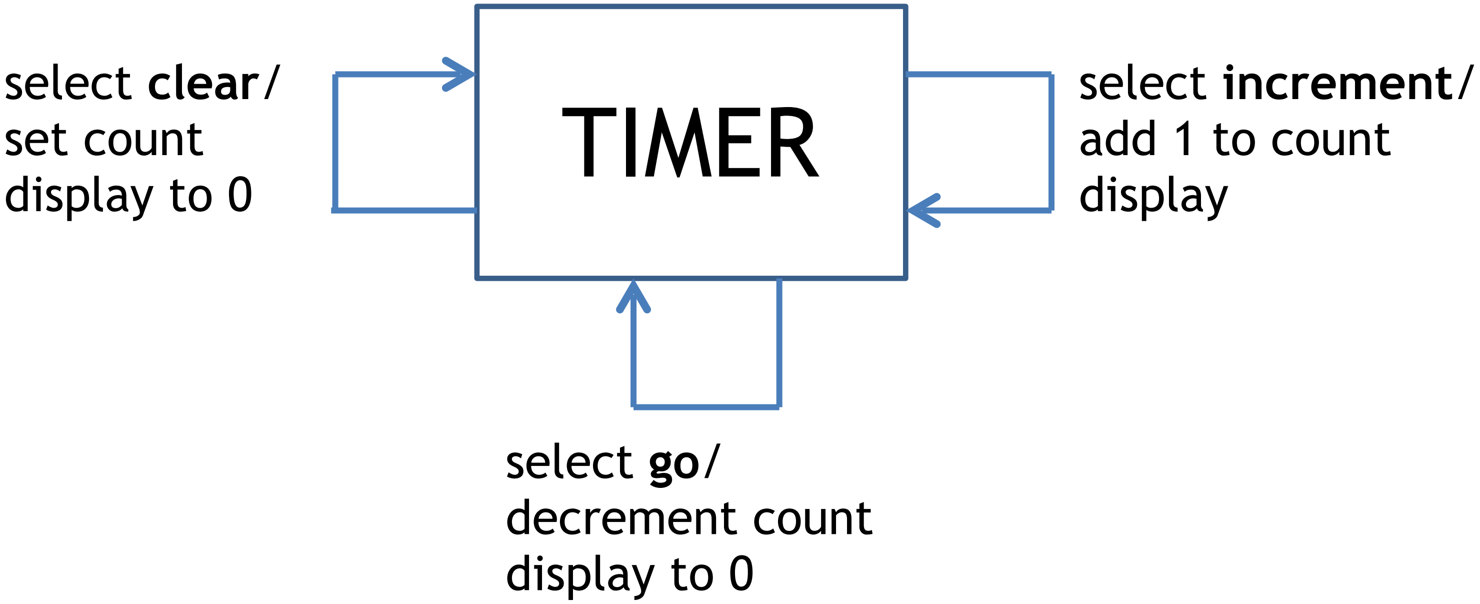
e.g. traffic lights



State Transition Diagrams

A system may have only one state

- e.g. a timer



TIMER

select **clear**/
set count
display to 0

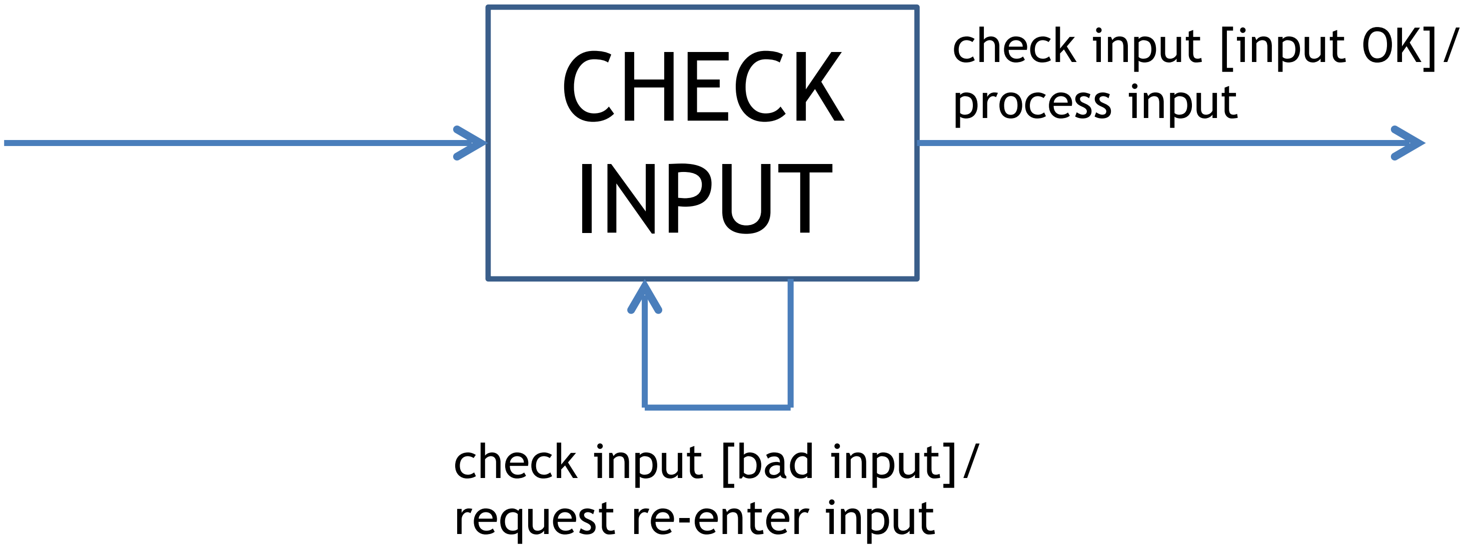
select **increment**/
add 1 to count
display

select **go**/
decrement count
display to 0

State Transition Diagrams

Can attach a guard to a state: [*guard*]

- *guard* must be true for transition to occur



CHECK INPUT

check input [input OK]/
process input

check input [bad input]/
request re-enter input

Things To Consider: Perspectives

The user's view of a system's states is not necessarily the same as a developer's view:

- the user interprets system behaviour in terms of their conceptual model of what system is for
- the developer knows about, and thinks about, underlying programming constructs
- e.g. to use editor, user doesn't need to know how interface or file system are implemented

Things To Consider: Perspectives

A user characterises state by properties of domain entities depicted on the display:

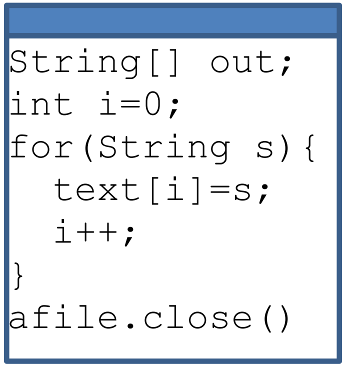
- current configuration: entities in problem domain represented by constructs in display
- events: interact with entities in problem domain by selecting appropriate constructs
- actions: changes to entities in problem domain reflected by changes in display

- e.g. editor: user thinks about documents in files represented by scrollable text on the screen

Things To Consider: Perspectives

A developer typically thinks of state in terms of programming constructs:

- current configuration: variable values; open files; `JFrame Component`s
- events: Java `Event`s
- actions: methods



```
String[] out;
int i=0;
for(String s){
   text[i]=s;
   i++;
}
afile.close()
```

- e.g. editor: developer thinks about character positions in arrays of `String`s, file I/O etc.

Things To Consider: Perspectives

The developer should construct the system to:

- reflect the user’s conceptual model
- and thereby enable effective system use

Things To Consider: Valid Behaviour

In any system, in any given state, only some of all possible events and actions are appropriate

- e.g. if light off, can turn it on but not off
- e.g. in editor, if no file is open can’t save

Often, when the state changes the appropriate events and actions change

- e.g. after turning light on can turn it off but not on
- e.g. in editor, after closing file, can’t close it again

Things To Consider: Valid Behaviour

It is important to constrain available events and actions to prevent action sequences which might damage the system (and user)

- e.g. can’t open washing machine door during wash cycle
- e.g. can’t exit editor without saving file

Also important to constrain available actions to prevent user confusion

- e.g. can’t close file which is already closed

Form Follows Function

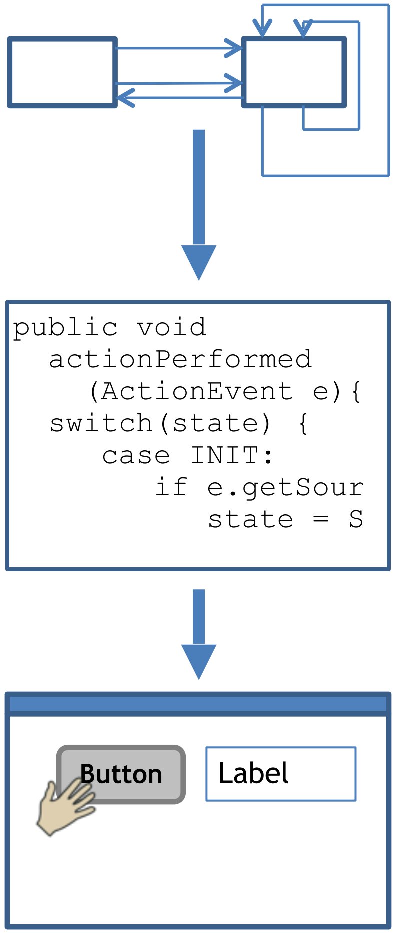
For effective system use, the user always needs to know what state the system is in

- *hidden mode*: in some state but no way to tell which one — avoid hidden modes!
- ensure user always knows current system state
  - unambiguous display content
  - explicit statement of mode
  - e.g. MS Word always indicates current style, font, etc.

From State Transition Diagram to Java

These are the idealised stages:

- draw state transition diagram
- will realise:
  - events as `Event`s with `Component` sources
  - guards as `if` statements
  - actions as methods
- design and implement interface
- implement actions as methods

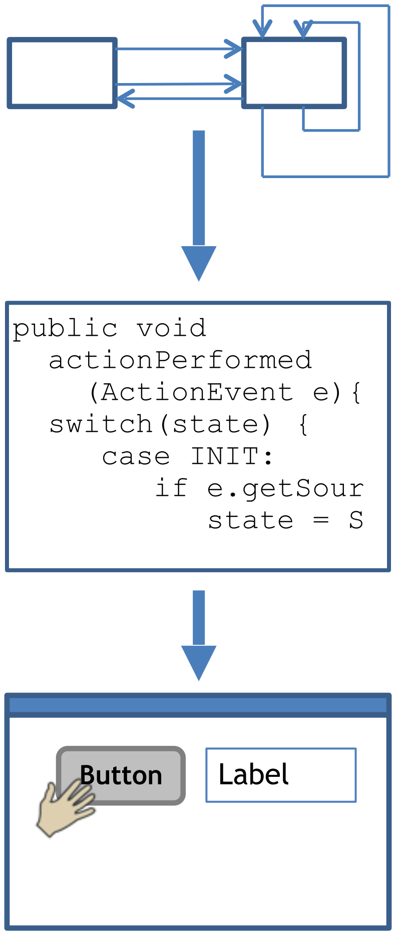


```
public void
  actionPerformed
    (ActionEvent e){
  switch(state) {
     case INIT:
        if e.getSour
           state = S
```

From State Transition Diagram to Java

State transitions are implemented in `actionPerformed`

- if only one state then:
  - no transitions
  - only need to identify events
- if more than one state then:
  - need to keep track of current state when reacting to event
  - to ensure that event is valid for state



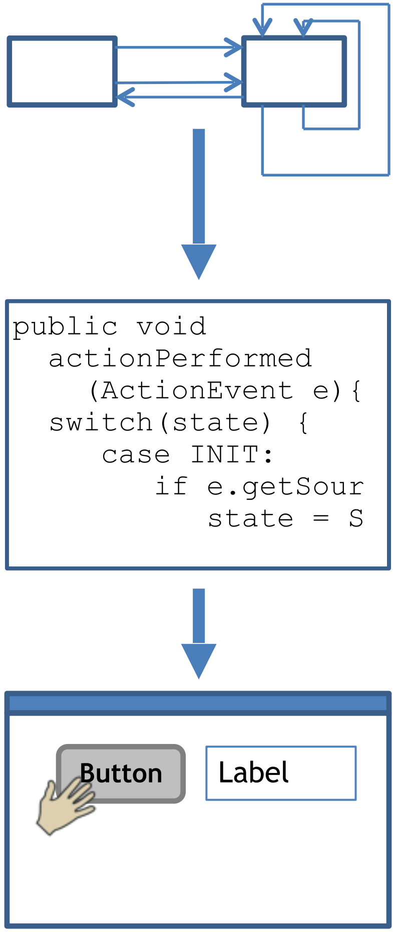
```
public void
  actionPerformed
    (ActionEvent e){
  switch(state) {
     case INIT:
        if e.getSour
           state = S
```

From State Transition Diagram to Java

Maintain a *state variable* to keep track of state

  - value indicates current state
  - for simplicity, represent states as integer constants

- `actionPerformed` **must:**
  - have separate cases for each state
  - within each state, identify and react to appropriate events



```
public void
  actionPerformed
    (ActionEvent e){
  switch(state) {
     case INIT:
        if e.getSour
           state = S
```

From State Transition Diagram to Java

```
final static int STATE1=1, STATE2=2, ...;  // define states
int state_variable = initial state;
...
public void actionPerformed(ActionEvent e) {
   switch(state_variable) { // for each state
      case STATE1:
          if(e.getSource()==$Component_{11}$) { // event
              state 1 action for this event
              state_variable = next state; // transition
          }
          else if(e.getSource()==$Component_{12}$) {
              ...
          }
          else ...
          break;
```

From State Transition Diagram to Java

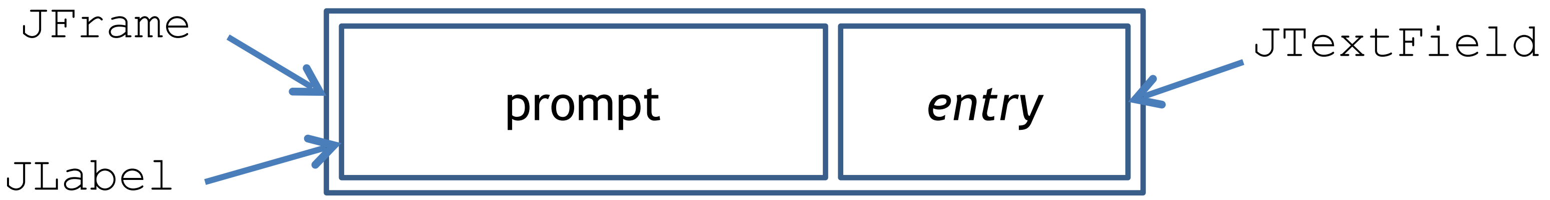
```
       case STATE2:
            if(e.getSource()==Component$_{21}$) { // event
                state 2 action for this event
                state_variable = next state;
            }
            else ...
            break;

       case STATE3: ...
    }
}
```

EXAMPLE

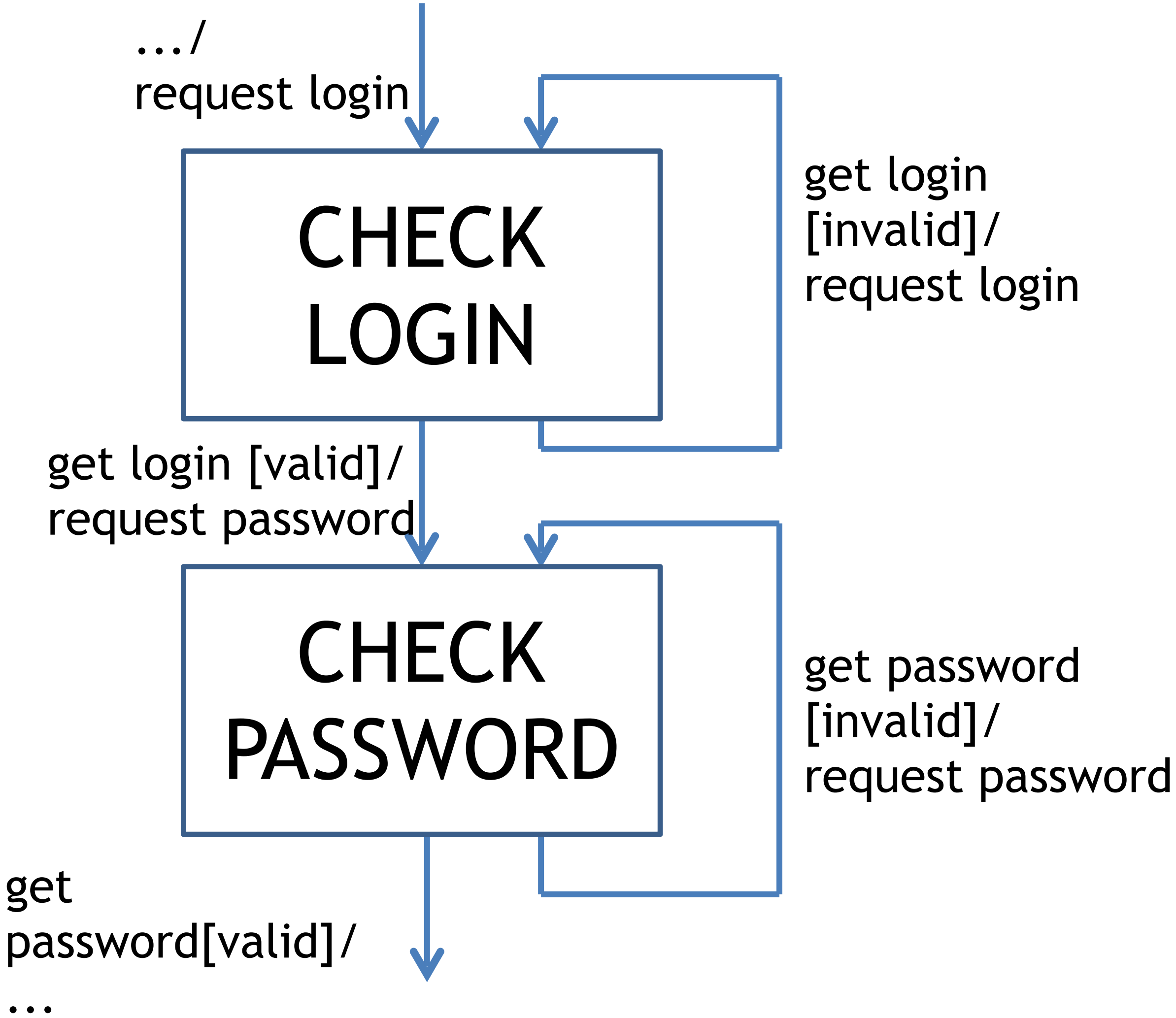
Example: password control

System access is controlled by a password

- file of user names and passwords
- user must enter name and password
  - verify name and password from file
- don't worry about encryption, hiding password etc.



Example: password control



.../
request login

CHECK LOGIN

get login
[invalid]/
request login

get login [valid]/
request password

CHECK PASSWORD

get password
[invalid]/
request password

get
password[valid]/
...

Example: password control

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

// An ID consists of a login name and a password
class ID
{  String login, password;

   ID(String l,String p)
   {  login = l;
      password = p;
   }
}
```

Example: password control

```
class Login extends JFrame implements ActionListener
{  JLabel prompt;          // for messages to user
   JTextField entry;       // for user to enter login details
   ID [] details;          // names and passwords
   int n = 0;              // number of IDs read in
   int idno;               // user's ID (once entered)

   // possible states of the system
   final static int LCHECK = 0;   // login name check
   final static int PCHECK = 1;   // password check
   final static int SUCCESS = 2;  // once password correct

   // current state of the system
   int state = LCHECK;            // set initial state
```

Example: password control

```
final static int MAXID = 100; // max. number of users

// method to read login name/password info from file
void getDetails(String f) throws IOException
{  String l,p;
   BufferedReader file =
      new BufferedReader(new FileReader(f));
   details = new ID[MAXID];
   l = file.readLine();
   while(l!=null)
   {  p = file.readLine();
      details[n] = new ID(l,p);
      n++;
      l = file.readLine();
   }
}
```

Example: password control

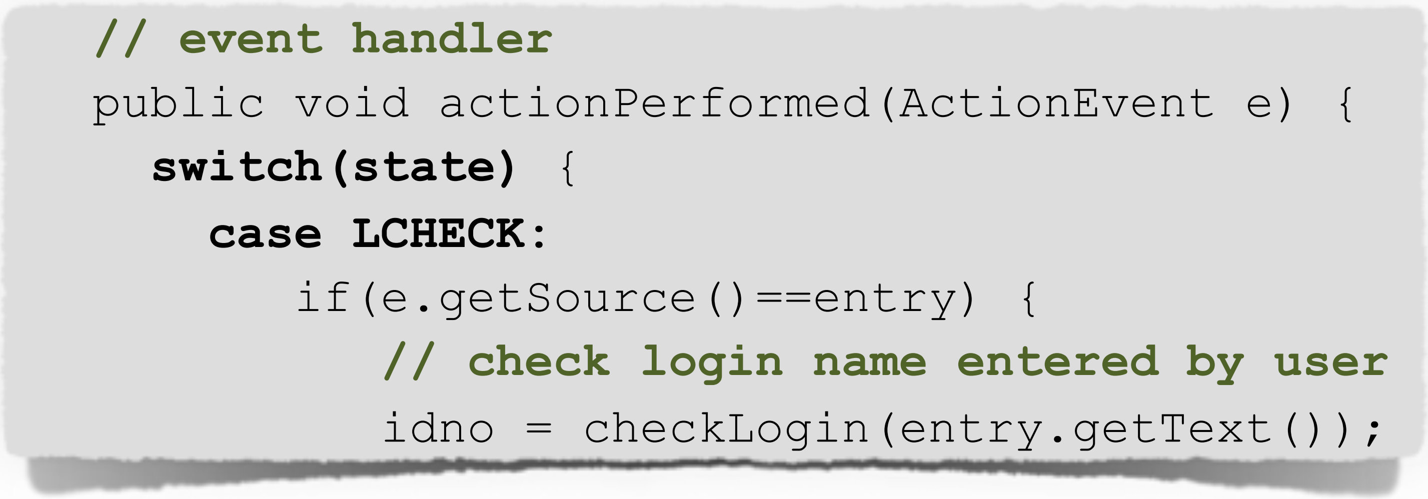
```
public Login(String filename) throws IOException
{   setLayout(new FlowLayout());

    // label for messages to user
    prompt = new JLabel("Please enter login:     ");
    prompt.setFont(new Font("Sansserif",Font.BOLD,18));
    add(prompt);

    // text field for user input
    entry = new JTextField(12);
    entry.setFont(new Font("Sansserif",Font.BOLD,18));
    add(entry);
    entry.addActionListener(this);

    // load logins from file
    getDetails(filename);
}
```

Example: password control

```
// checks whether login name exists
int checkLogin(String l)
{  for(int i=0;i<n;i++)
    if(details[i].login.equals(l))
     return i;
   return -1;
}

// checks whether password is correct for current login
boolean checkPassword(String p) {
   return details[idno].password.equals(p);
}
```

Example: password control

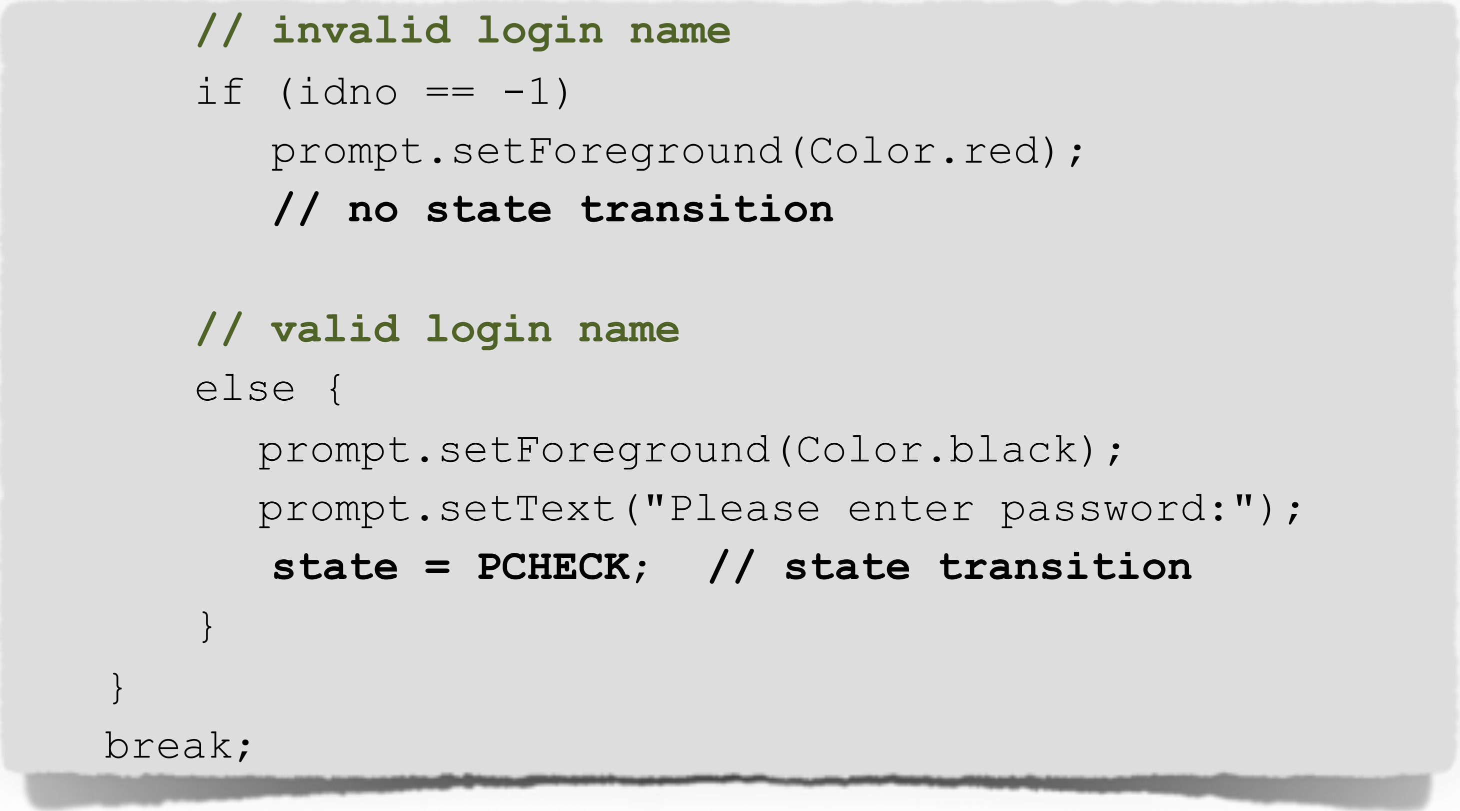
Implement the login check state:



```
// event handler
public void actionPerformed(ActionEvent e) {
  switch(state) {
    case LCHECK:
       if(e.getSource()==entry) {
          // check login name entered by user
          idno = checkLogin(entry.getText());
```

Example: password control

- different actions and transitions depending on whether login is valid or invalid:



```
    // invalid login name
    if (idno == -1)
       prompt.setForeground(Color.red);
       // no state transition

    // valid login name
    else {
      prompt.setForeground(Color.black);
      prompt.setText("Please enter password:");
      state = PCHECK;  // state transition
    }
}
break;
```

Example: password control

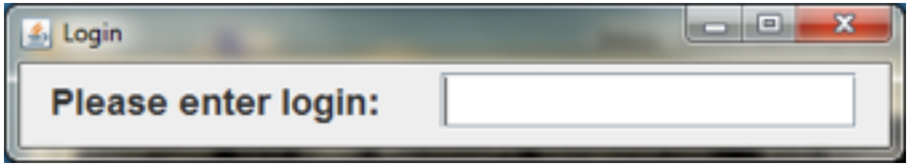
Implement the password check state:

```
case PCHECK:
    if (e.getSource() == entry) {
        // password invalid
        if (!checkPassword(entry.getText()))
            prompt.setForeground(Color.red);
        // password valid
        else {
            prompt.setForeground(Color.black);
            prompt.setText("Login successful");
            state = SUCCESS;  // state transition
        }
    }
    entry.setText("");
    break; // note break required after each case
}
```

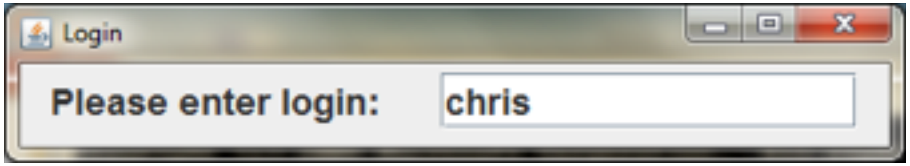
Example: password control

```
class TestLogin
{  public static void main(String [] args) throws IOException
   {  Login l = new Login("users.txt");
      l.setTitle("Login");
      l.setSize(450,80);
      l.setVisible(true);
      l.addWindowListener(...)
   }
}
```

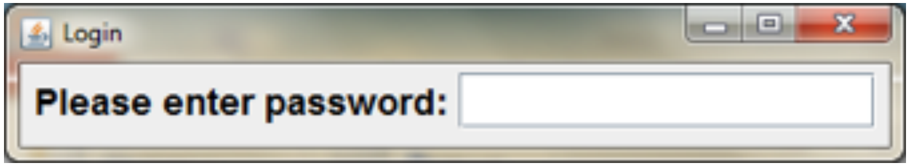
Example: password control



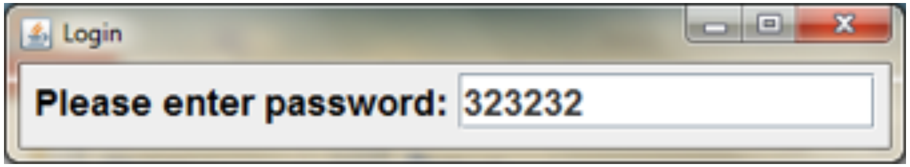
Please enter login:



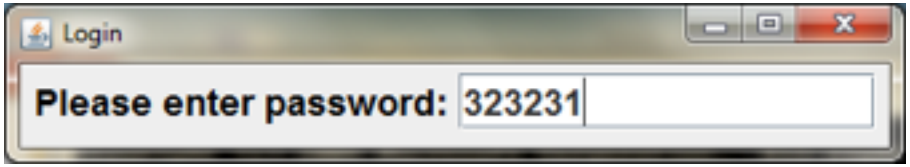
Please enter login: chris



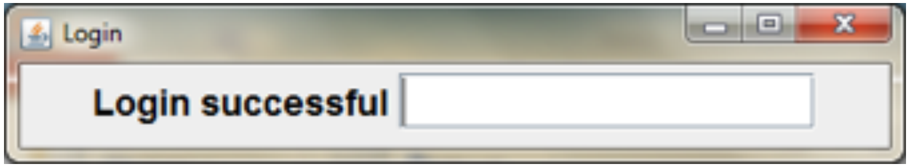
Please enter password:



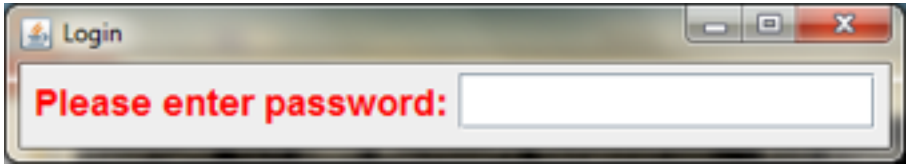
Please enter password: 323232



Please enter password: 323231



Login successful



Please enter password:

Example: password control

- Swing provides specialised `JPasswordField`
  - like `JTextField` but:
    - can set to not show password as typed
    - returns array of `char` not `String`

THAT’S IT!

Next Lecture

- A more involved example
  - State diagram
  - GUI design
  - Implementation