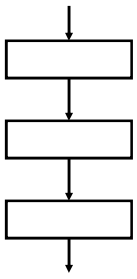
Decisions, buttons, booleans (JFS Chap 7)

- In this section:
  - The need for *alternative* actions within Java code based upon *decisions* taken at run-time
  - The `if` statement: various forms and example programs
  - Expressing complex tests (decisions)
  - Handling alternative events
  - The `boolean` data type: values, operators and variables

Alternative Paths of Execution

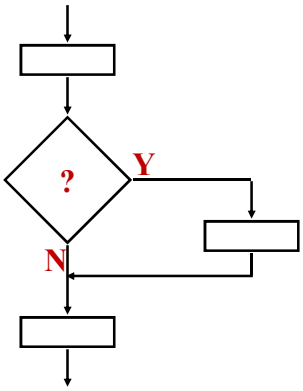
- Each of the methods we have seen so far does exactly the same thing whenever called
  - Even though different calls may involve different data
- Their method *bodies* have the same flow diagram form:



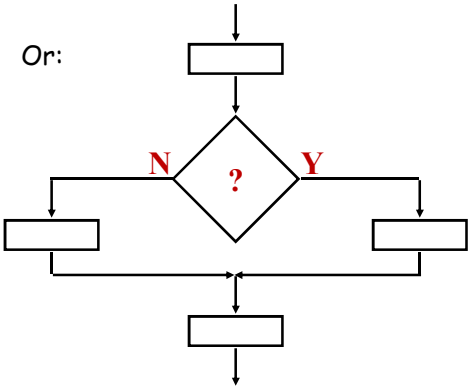
- However, in general, methods need to be able to do *different things*, depending on the circumstances...

Alternative Paths

- In flow diagram form we may have:



Or:



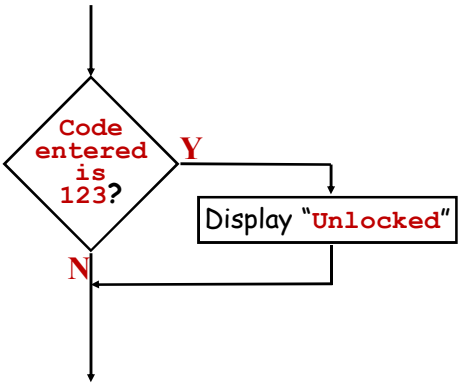
Example: the `Safe` program (JFS, p114)

- Appearance:
  - A text field for input, a button and a text field for output

Demo
Safe

- Behaviour:
  - Lower text field is initially clear
  - It remains clear when button is pressed if the wrong code has been given
  - It is changed to "Unlocked" the first time that the correct code is entered
- On the whole, this is like previous examples, BUT the `actionPerformed` method has a choice of action, depending on the code entered...

- The `actionPerformed` method body works like this:



- There are two distinct *paths of execution* through `actionPerformed`

The `Safe` program

```
public class Safe extends JFrame
                  implements ActionListener {

    private JLabel greetingLabel;
    private JTextField codeField;
    private JButton button;
    private JTextField outcomeTextField;

    public static void main(String[] args) {
        Safe demo = new Safe();
        demo.setSize(100,150);
        demo.createGUI();
        demo.setVisible(true);
    }
```

```
private void createGUI() {
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        Container window = getContentPane();
        window.setLayout(new FlowLayout());
        ------------------------------------------
        greetingLabel = new JLabel("Enter code");
        window.add(greetingLabel);
        ------------------------------------------
        codeField = new JTextField(5);
        window.add(codeField);
        ------------------------------------------
        button = new JButton("Unlock");
        window.add(button);
        button.addActionListener(this);
        ------------------------------------------
        outcomeTextField = new JTextField(5);
        window.add(outcomeTextField);
    }
```



```
public void actionPerformed(ActionEvent event) {
1.      String codeString = codeField.getText();
2.      int code = Integer.parseInt(codeString);
        if (code == 123) {
3.          outcomeTextField.setText("Unlocked");
        }
    }
}
```

Note the layout convention: next line and indented

- The body of `actionPerformed` comprises *three steps*:
  - Steps 1 and 2: Fetch and convert the user's input
  - Step 3: An `if` statement – contains an embedded statement
- These are executed *in sequence*, as usual
  - This may include or exclude the `setText` statement

The "one-branch" `if` statement

- The `if` statement has the form:

```
if (condition) {
    action
}
```

- The *condition* is evaluated, then:
  - If it is `true`, then *action* is carried out
  - If it is `false`, then *action* is *not* carried out
  - And in either case execution continues *after* the `if`
- The entire `if` statement (3 lines here) counts as a single statement, a single step in the overall method body

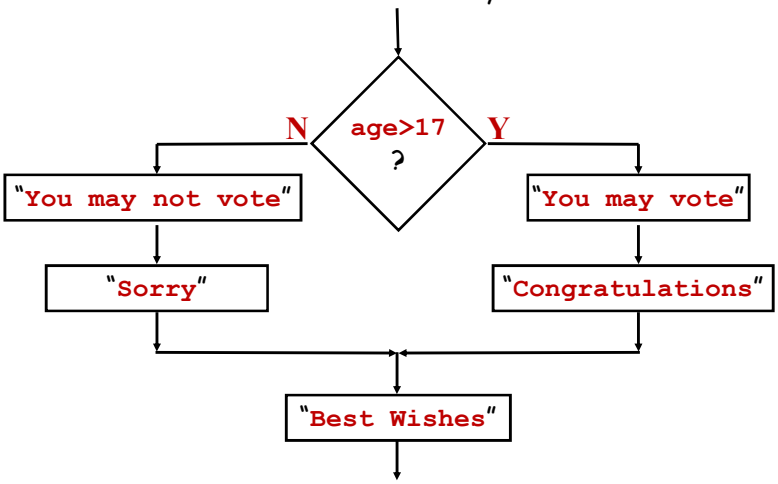
The `Voting` program (JFS, p117)

- The user enters their age and clicks a button
- The program chooses *one of two* actions
- Appearances:

Demo Voting

- Most details of the program are similar to previous examples
- But this time `actionPerformed` has a slightly different form...

- The `actionPerformed` method body has the form:



- Again, there are two distinct *paths of execution*
  - But now they *both* do something!

The `Voting actionPerformed` method

```
public void actionPerformed(ActionEvent event) {
Step 1  int age = Integer.parseInt(ageField.getText());
        if (age > 17) {
            decisionField.setText("You may vote");
            commentaryField.setText("Congratulations");
Step 2  }                                          "true" branch
        else {
            decisionField.setText("You may not vote");
            commentaryField.setText("Sorry");      "false" branch
        }
Step 3  signOffField.setText("Best Wishes");
    }
```

Key points

- Note the two-branch structure:

```
if (condition) {
    action
}
else {
    action
}
```

- In each case `action` may be:
  - Either: `statement;` (and the `{ }` are *optional*)
  - Or:
```
statement;
statement; ...
```
- A `{ ... }` structure is called a "block"
  - Java either executes the block, or *skips* the *whole* of the block
- This applies to both the one- and two-branch forms of the `if`

Expressing the test conditions

- The test conditions are *expressions* formed from:
  - Values, variables, operators
- The *comparison* operators: (for building simple tests)

```
<   >   ==   !=   <=   >=
```

  - NOTE: `==` (adjacent = signs)
  - Examples:

```
age > 17    age >= 18
```

(the same result)

- *Logical* operators:
  - Building complex tests from simple ones

```
&&    "and"
||    "or"
!     "not"
```

- Use `( ... )` if appropriate to group things

- Examples:

```
(age > 6) && (age < 16)
(age == 1) || (age == 13) || (age == 12)
```

- Note: The operator precedence is:

```
Highest:    !
            *  /  %
            +   -
            <  <=  >  >=
            == !=
            &&
Lowest:     ||
```

  - So we may often omit parentheses. E.g.

```
age >= mid-2 && age <= mid+2 || age == 16
```

  makes sense, and if `mid` has value 6 then this whole expression is true when `age` is 4, 5, 6, 7, 8 or 16

JButtons – review of details

- The program must "implement `ActionListener`", eg:

```
public class CarCount extends JFrame
                implements ActionListener {
```

  - If we have sliders *and* buttons, then:

```
                implements ActionListener,
                           ChangeListener {
```

- We must declare one variable to hold each button's details, eg:

```
private JButton doIt;
```

- In `createGUI` we must create the button, add it to the display, and register the program as listening for clicks on it, e.g.

```
doIt = new JButton("Click me");
window.add(doIt);
doIt.addActionListener(this);
```

The constructor has one parameter: the button's label

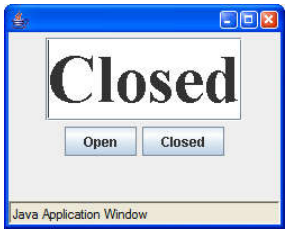
Using two or more buttons (JFS, p127)

- If we have several sliders, then `stateChanged` is called *whichever slider is adjusted*
  - The current values of *all sliders* can be fetched and used
- Similarly, if we have several buttons, then `actionPerformed` is called *whichever button is pressed*
  - **But** in this case the program will usually need to check *which button was the "source of the event" ...*
  - ... and take *appropriate action* - `if`s are essential for this
- The parameter of `actionPerformed` (the `event` of type `ActionEvent`) contains the identity of the pressed button
  - We can extract it using: `event.getSource()`
  - And we can use it like this:

```
if (event.getSource() == button variable name)
    action
```


Using two buttons: The `SimpleShopSign` program (not in JFS)

- Appearance/behaviour:
  - One text field and two buttons
  - The message changes *depending on which button is pressed*



Demo SimpleShop Sign

- Most details of the program are similar to previous examples
- The text field has additional formatting in `createGUI`:

```
textField = new JTextField("Closed");
textField.setFont(new Font("Times New Roman",
                           Font.BOLD, 60));
window.add(textField);
```

- Again `actionPerformed` has a slightly different form...

```
public void actionPerformed(ActionEvent event) {

    if (event.getSource() == openButton) {
        textField.setText("Open");
    }

    if (event.getSource() == closedButton) {
        textField.setText("Closed");
    }
}
```

- Note that *both* `if` statements are executed ...
  - But only *one* of the two conditions can be true
  - We could have an `else` between the `if`s, but it doesn't matter here

Boolean values: the `boolean` type

- We have seen 'tests' in the context of `if` statements
  - We think of their "result" as "yes/no" or "true/false"
- In fact Java represents the result of every 'test' as one of two special *values*
  - These are represented in the language as `true` and `false`
  - These are the (only) two values in the `boolean` data type
  - (Named after George Boole, a 19th century logician)
- Examples: Assuming that `age` has value 15
  - `age > 17` has value `false`
  - `(age > 6) && (age < 16)` has value `true`

    because `age > 6` is `true`, `age < 16` is `true` and `true && true` is `true`

Boolean variables

- Values of the type `boolean` are dealt with just like any other type of data:
  - They may be stored in variables, eg:

```
boolean finished, canVote;
finished = true;
canVote = age > 17;
```

  - And these variables may be referred to in tests, eg:

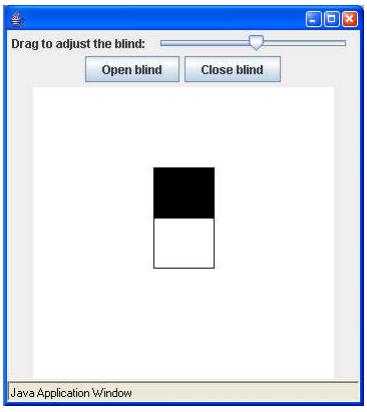
```
if (finished)
    action
if (canVote && finished)
    action
```

- This gives us a very powerful facility for controlling programs:
  - An event handler can record its decisions in global `boolean` variables…
  - …which can be tested later by other methods
  - See the `ShopSign` program in JFS (p138)

Multiple events: buttons *and* sliders
QuickBlind (not in JFS)

Demo
QuickBlind

- Another version of the WindowBlind example
  - This time with two additional buttons for fully opening and closing the blind
- The program must handle both `stateChanged` and `actionPerformed` events
- Design problem:
  - Would like to avoid having all the drawing code in *both* the `stateChanged` and `actionPerformed` event handlers
  - Solution: Draw in a separate method, and call from both



QuickBlind – key components

First, a method to "paint" the screen with up-to-date information:

```
private void paintScreen(Graphics g) {
    g.setColor(Color.white);  // Erase
    g.fillRect(0, 0, 300, 300);
    int sliderValue = slider.getValue();
    g.setColor(Color.black);  // Draw
    g.drawRect(120, 80, 60, 100);
    g.fillRect(120, 80, 60, sliderValue);
}
```

Note: Up-to-date slider setting

Next, the event handlers:

```
public void stateChanged(ChangeEvent e) {
    Graphics paper = panel.getGraphics();
    paintScreen(paper);
}

public void actionPerformed(ActionEvent e) {
    if (e.getSource()==open) {
        slider.setValue(0);
    }
    if (e.getSource()==close) {
        slider.setValue(100);
    }
    Graphics paper = panel.getGraphics();
    paintScreen(paper);
}
```

Note: Force slider to change its setting

End of section