Graphical User Interfaces (GUIs)

- Before we get on to the case study proper, we discuss the 'architecture' of programs that have GUIs
  - we see that GUIs are *event-driven programs*
  - we see we need three kinds of objects: *components*, *events* and *listeners*
  - we see different ways of organising these three
  - we discuss some related design patterns (Observer, Model-View, Model-View-Controller, etc.)
- Then we work through a simple design process resulting in a working system that has a GUI

Java versions/GUI Toolkit versions

- To write GUIs, need a GUI Toolkit (i.e. library classes):
  - AWT (Abstract Windowing Toolkit): for low functionality, inefficient GUIs with native look-and-feel
  - Swing: for high functionality, more efficient GUIs with pluggable look-and-feel

Java	Development kit	GUI Toolkit
Java 1	JDK 1.0 (1.0.1-1.0.2) library of about 200 classes/interfaces	AWT (with inheritance-based event model)
	JDK 1.1 (1.1.1-1.1.8) library of about 500 classes/interfaces	AWT (with delegation-based event model) or Swing (downloaded separately)
Java 2	SDK 1.2 (1.2.1-1.2.2) library of about 1800 classes/interfaces	Swing

Event-driven programming

- GUIs are examples of *event-driven programs*: events cause methods to be executed
- *Components* (widgets, window gadgets, controls): Objects we create which have a screen presence and with which the user interacts (e.g. buttons). When a user interacts with a component, that component is the *event source*
- *Events:* Java automatically creates an event object when the user interacts with a component. The event object has an instance variable recording the event source
- *Listeners:* Objects we create and to which we delegate the task of responding to events. A listener object will have an instance method, called a *handler*, that is invoked by Java to take appropriate action to respond to the event

Components

- Components include buttons, textfields, scrollbars, windows, pop-up menus, menu items, etc.
- E.g. `javax.swing.JButton`
  - Constructor, e.g.:

    ```
    JButton btn = new JButton("Press me");
    ```

  - Instance methods, e.g.:

    ```
    String str = btn.getText();
    btn.setText("Press me harder!");
    ```

    Also `addActionListener` and `removeActionListener` — see later
  - If you have, e.g., a window, then the window object will offer methods that allow you to add components (such as buttons) to the window — see the example program

Event objects

- There are different classes of event to represent different user actions (some quite low-level, others quite high-level)
- E.g. `java.awt.event.ActionEvent`
  - Unlikely we would call the constructors
  - Created by Java when, e.g., a user clicks on a button
  - Instance methods for an `ActionEvent ae`:
    * `ae.getSource();` — returns a reference to the event source, i.e. the button/textfield/etc. that generated this event
    * `ae.getActionCommand();` — returns a `String`; for buttons, the button's text

Listener objects

- Listener objects are delegated the task of responding to certain events for certain components
- Listener objects contain *handler methods* that run in response to events. Unlikely that we would invoke the handlers ourselves: Java does it for us when events occur
- Listeners can be any class of object we like but must implement certain interfaces to guarantee inclusion of appropriately named handlers
- E.g. we could write a listener for button clicks
  - We would state: `implements ActionListener` (`java.awt.event.ActionListener`)
  - So we must include definition of `actionPerformed` method

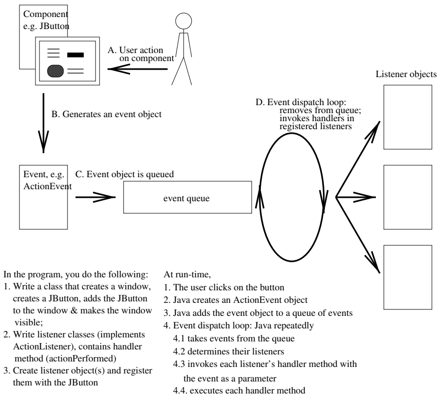
Registering listeners

- How does Java know which listener object will respond to a component's events?
- If a component's events are to be responded to, we must register one or more listeners with the component
- E.g. if `btn` is a `JButton` and `lstnr1` and `lstnr2` are listeners (`implements ActionListener`):

```
btn.addActionListener(lstnr1);
btn.addActionListener(lstnr2);
...
btn.removeActionListener(lstnr2);
```

- When the user interacts with the button, Java invokes the handler methods in each registered listener

A button-click pic



Exercise: Colour this picture in using wax crayons.

Our first GUI

```
import javax.swing.JFrame;
import java.awt.Container;
import javax.swing.JButton;
import java.awt.event.ActionListener;
public class FrameWithButton
   extends JFrame
{  public FrameWithButton()
   {



   }

   private Button btn;

   public static void main(String[] args)
   {  JFrame fwb = new FrameWithButton();
      fwb.setBounds(100, 300, 400, 200);
      fwb.setVisible(true);
   }
}
```


Our first GUI's listener class

```
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;

public class SimpleButtonListener
   implements ActionListener
{
  public SimpleButtonListener()
  {
  }

  public void actionPerformed(ActionEvent ae
  {


  }
}
```


Observations

- We didn't have to code up any of the low-level details (detection of user interaction, creation of event object, enqueuing of event object, the event dispatch loop)
- A component may have zero, one or more listener objects registered with it
- Our code doesn't send a message to invoke the handler, `actionPerformed`: Java invokes it for us
- Handlers shouldn't involve a lot of computation. Why?


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

public class TempConversionGUI
   extends JFrame
{
   public TempConversionGUI()
   {  fahrLabel =
         new JLabel(DEGREES_FAHRENHEIT);
      fahrField =
         new JTextField(INIT_FAHR, TFCOLS);
      centLabel =
         new JLabel(DEGREES_CENTIGRADE);
      centField =
         new JTextField(INIT_CENT, TFCOLS);
      compFahrBtn =
         new JButton(COMPUTE_FAHRENHEIT);
      compCentBtn =
         new JButton(COMPUTE_CENTIGRADE);
```

```
   // Add components to container










  // Register listeners
      compFahrBtn.addActionListener(
         new CompFahrBtnLstnr(this));
      compCentBtn.addActionListener(
         new CompCentBtnLstnr(this));
      addWindowListener(
         new GenericWindowLstnr());
}
```



```
   public void setFahrField(int theFahr)
   {
   }

   public int getFahrField()
   {
   }

   // setCentField and getCentField similarl

   private JLabel fahrLabel;
   private JTextField fahrField;
   private JLabel centLabel;
   private JTextField centField;
   private JButton compFahrBtn;
   private JButton compCentBtn;
```



```
   public static final String
    DEGREES_FAHRENHEIT = "Degrees Fahrenheit
   public static final String
    DEGREES_CENTIGRADE = "Degrees Centigrade
   public static final String
    COMPUTE_FAHRENHEIT = "Compute Fahrenheit
   public static final String
    COMPUTE_CENTIGRADE = "Compute Centigrade

   private static final String
      INIT_FAHR = "212";
   private static final String
      INIT_CENT = "100";
   private static final int TFCOLS = 6;

   public static void main(String[] args)
   {  JFrame tcg = new TempConversionGUI();
      tcg.setBounds(50, 75, 250, 150);
      tcg.setVisible(true);
   }
}
```



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

public class CompFahrBtnLstnr
   implements ActionListener
{
  public CompFahrBtnLstnr(
        TempConversionGUI theGUI)
  {
  }

  public void actionPerformed(ActionEvent ae
  {



  }

  private TempConversionGUI gui;
}
```

`CompCentLstnr` will be very similar.


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

public class GenericWindowLstnr
   implements WindowListener
{
  public GenericWindowLstnr()
  {
  }

  public void windowOpened(WindowEvent we)
  {
  }

  public void windowIconified(WindowEvent we
  {
  }

  // Likewise windowActivated, windowClosed
  // windowDeactivated, windowDeiconified &

  public void windowClosing(WindowEvent we)
  {  System.exit(0);
  }
}
```


Components in our GUI

- Need a subclass of `JFrame`, 2 `JButton`s &...
- `javax.swing.JLabel`
  - allows display of user-uneditable text/images
  - constructor can specify initial text
  - methods `setText`, `getText`, etc.
- `javax.swing.JTextField`
  - allows display, entry & edit of a single line of text (`String`s!)
  - constructor can specify initial value & no. of columns
  - methods `setText`, `getText`, etc.

GUI containers

- In GUI toolkits, such as AWT & Swing, some components are *containers*
  - can have other components added to them
  - e.g. a `JFrame` (window) is a container: it can have, e.g., `JButton`s added to it (to its content pane)
  - a container can even have other containers added to it
  - e.g. a `JFrame` can have, e.g., two `JPanel`s added to it and the `JPanel`s are containers that could contain `JButton`s
  - (in fact, in Swing, *all* components are containers — a major increase in flexibility!)

Layout managers

- You can position components in containers using co-ordinates (`setLocation`) but this is insensitive to different platforms and window resizing
- Each container has an instance variable which holds a *layout manager* object, which you can get and set (`get/setLayout`)
- When you add objects to a container, the layout manager positions them and sizes them for you
- `FlowLayout` places components sequentially, filling each line from left-to-right and filling the lines from top-to-bottom
  (Very simple, but not usually the best approach)
- There is also `BorderLayout`, `CardLayout`, `GridLayout`, `GridBagLayout` & `BoxLayout` and you could define your own (hard!)

The listeners

- In this program, only the buttons have listeners registered with them (we could have registered listeners with the textfields so that pressing 'Enter' invokes a handler method)
- There's a clean separation of concerns between the GUI class (handles the I/O) and the classes that handle 'real' processing
  — this is achieved by giving the listeners a reference to the GUI class so they can send messages to the GUI class
- We've not done anything about handling illegal user input
- Windows, such as `JFrame`, generate window events (`WindowEvent`); they need a window listener (`addWindowListener`) which implements `WindowListener` (7 methods)

Some design decisions in GUIs

- Variations in the listeners of the temperature conversion GUI
- Convenient listener definitions by subclassing
- Revisions that change the program from four class definitions to one class definition
  - listeners as anonymous classes
  - combining listeners
  - making the GUI act as its own listener

I: Convenient listener class definitions

The Java library contains abstract class `WindowAdapter`.
It looks something like:

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

public abstract class WindowAdapter
   implements WindowListener
{
  public void windowOpened(WindowEvent we)
  {
  }

  // Likewise windowActivated, windowClosed,
  // windowDeactivated, windowDeiconified,
  // windowIconified & windowClosing
}
```


I: Convenient listener class definitions, cont'd

Our definition of `GenericWindowLstnr` is now more concise:

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

public class GenericWindowLstnr
   extends WindowAdapter
{
  public void windowClosing(WindowEvent we)
  {  System.exit(0);
  }
}
```

II: Anonymous classes

- Can avoid separately defining very simple class definitions by using *anonymous classes*
- Use for 'one-off' classes
  - the class is defined at the same point in the code where an instance is being created
  - the class has no name (an anonymous subclass of some superclass or an anonymous implementer of some interface) & no real constructor of its own
  - much used for listeners

II: Anonymous classes, cont'd

Delete `GenericWindowLstnr`.

In TempConversionGUI, change registration statements from

```
addWindowListener(new GenericWindowLstnr());
```

to

```
addWindowListener(new WindowAdapter()
{ public void windowClosing(WindowEvent we)
  {  System.exit(0);
  }
});
```


III: Combining listeners

- One listener object could respond to events from more than one component
- E.g. replace `CompFahrBtnLstnr` and `CompCentBtnLstnr` by `CompTempBtnLstnr` (next slide)
  - Registration changes:

```
compFahrBtn.addActionListener(
      new CompFahrBtnLstnr(this));
compCentBtn.addActionListener(
      new CompCentBtnLstnr(this));

CompTempBtnLstnr lstnr =
   new CompTempBtnLstnr(this);
compFahrBtn.addActionListener(lstnr);
compCentBtn.addActionListener(lstnr);
```

  - Fewer classes, but `actionPerformed` is more complex, less efficient & less cohesive


```
import java.awt.event.*;
public class CompTempBtnLstnr
   implements ActionListener
{
  public CompTempBtnLstnr(
     TempConversionGUI theGUI)
  {  gui = theGUI;
  }

  public void actionPerformed(ActionEvent ae
  {
     if
     {  int cent = gui.getCentField();
        int fahr = (int) Math.round( etc.
        gui.setFahrField(fahr);
     }
     else
     {  int fahr = gui.getFahrField();
        int cent = (int) Math.round( etc.
        gui.setCentField(cent);
     }
  }
  private TempConversionGUI gui;
}
```


IV: The GUI class as a listener

- Could do away with separate listener classes
- The GUI class could act as the listener for the components
- i.e. when an event occurs, the object that responds is the GUI itself
- Quite common but can be a poor design (high coupling)
- Suppose `TempConversionGUI` will act as a listener for its own buttons
  - `TempConversionGUI` will declare `implements ActionListener`
  - therefore it would have to include `actionPerformed` method

IV: The GUI as a listener, cont'd

- The registration statements change from

```
CompTempBtnLstnr lstnr =
   new CompTempBtnLstnr(this);
compFahrBtn.addActionListener(lstnr);
compCentBtn.addActionListener(lstnr);
```

  to

```
compFahrBtn.addActionListener(this);
compCentBtn.addActionListener(this);
```

- So when the buttons are pressed, it's the `TempConversionGUI` object (i.e. `this`) whose `actionPerformed` method is executed
- A small advantage is that the `actionPerfomred` method can be simpler: it's in the same class so methods/variables are now visible

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class TempConversionGUI
   extends JFrame
   implements ActionListener
{
   public TempConversionGUI()
   {  // create labels, textfields & buttons
      // and add to window, as before

      // The revised listener registration
      compFahrBtn.addActionListener(this);
      compCentBtn.addActionListener(this);

      addWindowListener(new WindowAdapter()
      {  public void windowClosing(WindowEve
         {  System.exit(0);
         }
      });
   }

   // four methods, get/setFahr/CentField,
   // as before
```



```
 public void actionPerformed(ActionEvent ae)
 {  if
    {  int cent =
       int fahr = (int) Math.round( etc.


    }
    else
    {  int fahr =
       int cent = (int) Math.round( etc.


    }
 }

 // variables as before

 // main method as before
}
```


More UML Diagrams: for behaviour

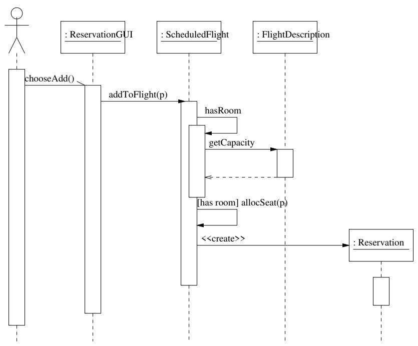
- Interaction diagrams: show how objects collaborate to perform a task
  - Sequence diagrams: focus on message sequence
  - Collaboration diagrams: focus on object connections
- State diagrams:
  - Statechart diagrams: focus on an object's states and state changes
  - Activity diagrams: focus on workflow

Sequence diagrams

- Across the top, boxes denoting objects (see class diagrams) or stick figures representing the user
- Beneath each box, a lifeline (dashed line)
- Blocks on lifelines represent a method executing
- Lines with arrows represent messages and dashed lines represent method returns

Interaction type	Symbol	Meaning
Synchronous		Sender loses control until return from receiver
Return		Return from receiver to sender
Flat		Sender loses control; return is not expected
Asynchronous		Sender stays active; receiver becomes active

Example Sequence Diagram



Notes

- For an object calling its own methods (including recursion) or a callback, optionally show nested activation blocks
- Can show conditionals or iteration, but keep this simple
- Returns can be shown but are optional
- Object creation messages point to the new object, and can optionally be labelled ⟨⟨create⟩⟩
- Object deletion is shown by a 'X' on a lifeline; an object deletion message can optionally be labelled ⟨⟨destroy⟩⟩
- Other options, e.g. timing constraints

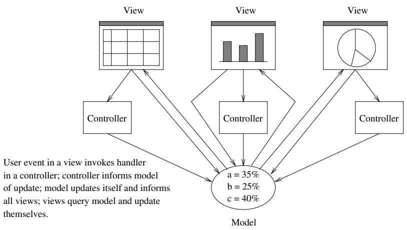
A program with two views

- View 1: A `JFrame` containing a slider
  - constructor: `JSlider(min, max, initial)`
  - generates `ChangeEvent`s
  - listener implements `ChangeListener`, has a `stateChanged` method
- View 2: A `JFrame` containing a `JTextField`, with a separate listener object
- In `TempConversionGUI`, the listener causes the view to be updated.
- Problem: in a program with multiple views, how will changes made in one window propagate to the other window without coupling each listener to each view?

Model-View-Controller

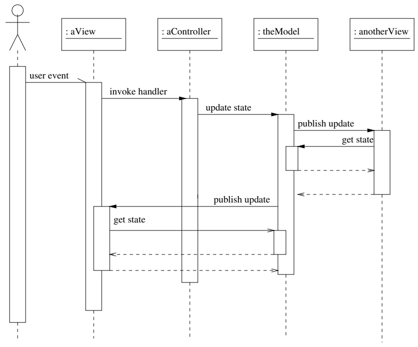
- Gives a good architecture for programs with GUIs
  - *Model:* Defines & manages application data; keeps list of associated views; responds to messages from controllers; informs views of changes to the data
  - *Views:* Create & arrange components; accurately reflect the model visually to users
  - *Controllers:* handle events; inform model of events
- Our previous programs showed architectural options for view and controllers; now see advantages of separating out model

Multiple views



- The model maintains a list of views that have subscribed
- The model publishes notification of change of state to all subscribed views

MVC Sequence Diagram



Controllers respond to user-events by informing the model; the model broadcasts notification of change to all views. Allows multiple views of the same model to be kept 'in step', without tightly coupling the views

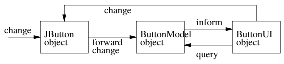
Using MVC

- Base the architecture of your GUI on MVC (even if only one view)
- Still common to collapse controllers (listeners) into views, resulting in a Model-View architecture
- We look at a program for a shop till that has the MVC architecture
- Then we look at a program for a burger bar that has the architecture
- But first. . .

MVC in Swing

- GUI toolkits, such as Swing, may use MVC (or MV) for the implementation of the components
- E.g. `ColorChooser` has obvious model and views/ controllers
- But even, e.g., `JButton` has model and view/controller
- A programmer who wishes to use a button (and many of the other simple components) in his/her GUI can remain (largely) unaware of MVC
- But a programmer who wishes to use a list (or any of the advanced Swing components) in his/her GUI has to understand the MVC architecture
- Using it for all components is how Swing achieves pluggable look-and-feel

‘Behind the scenes’ with `JButton` (simplified)



- The button model is where data about the button is kept. (A class that implements `ButtonModel`, e.g. `DefaultButtonModel`)
- The UI delegate knows how to display the button (view) and how to respond to button clicks (controller). (A class that implements `ButtonUI`, e.g. `BasicButtonUI, MotifButtonUI`)
- The model can change due to user actions (clicks) or messages sent to the `JButton` (e.g. `setText`), which are forwarded to the model
- The UI delegate is informed of any change to the model

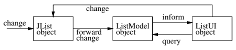
Unchangeable lists

If we're not updating the contents of a list, then the programmer doesn't need to know about `JList`'s MVC architecture.

```
list = new JList(v); // Vector or Obj array
list.setSelectionMode(
      ListSelectionModel.SINGLE_SELECTION);
list.setSelectedIndex(0);

list.addListSelectionListener(
         new ListSelectionListener()
{  public void valueChanged(
      ListSelectionEvent lse)
   {  if (lse.getValueIsAdjusting())
      {  // do nothing
      }
      else
      {  int i = list.getSelectedIndex();
         // do something with v.elementAt(i)
      }
   }
});
```


Changeable lists

- Looking after the data that is in the list is the responsibility of a list model object
- List models implement `ListModel` (specifies 4 methods: `addListDataListener`, `removeListDataListener`, `getSize` & `getElementAt`)
- Swing provides `DefaultListModel` (a subclass of `AbstractListModel`), offering numerous methods in addition to the 4 above (e.g. `addElement`, `elementAt`, `insertElementAt`, `remove`)
- When the model is updated using these methods, the view also gets updated
- The programmer must deal with list & model



Changeable list example

```
model = new DefaultListModel();
list = new JList(model);

btn = new JButton("Insert");

btn.addActionListener(new ActionListener()
{ public void actionPerformed(ActionEvent ae
  {  int index = list.getSelectedIndex();
     int size = model.getSize();
     if (index == -1 || index + 1 == size)
     {  model.addElement("item");
        list.setSelectedIndex(size);
     }
     else
     {  model.insertElementAt("item");
        list.setSelectedIndex(index + 1);
     }
  }
});
```


Customised responses to list model changes

- When the model is changed, the view is updated. But you can register your own listeners *with the model* so, when the *model* is changed, your own handlers are run as well

```
model.addListDataListener(
               new ListDataListener()
{  public void contentsChanged(
               ListDataEvent lde)
   {  // what to do if elements are modified
   }
   public void intervalAdded(
               ListDataEvent lde)
   {  // what to do if something is inserted
   }
   public void intervalRemoved(
               ListDataEvent lde)
   {  // what to do if something is deleted
   }
});
```


Changeable lists: even more customisation

- You can write your own list model (containing at least the methods in the `ListModel` interface) based on any of the data structures you know
- E.g. you could define a list model offering `push`, `pop`, etc., and then create `JList`s from this
- `JList`s can display strings & images; if you want to display other components (e.g. `JLabel`s) you can define your own list cell renderer

The `Till` class definition: guess the rest!

```
public Till(int theTillFloat)
public int getTillFloat()
public int getTakings()
public int getTillTotal() // float + takings
public int getTransactionCount()
public int getTotalOwed()

public void ringUpItem(int theItemPrice)
{  amountOwedByThisCust += theItemPrice;
   if (gui != null)
   {  gui.update();
   }
}
public void submitTransaction()
{  transactionCount++;
   takings += amountOwedByThisCust;
   amountOwedByThisCust = 0;
   if (gui != null)
   {  gui.update();
   }
}
```



```
public void cancelTransaction()
{  amountOwedByThisCust = 0;
   if (gui != null)
   {  gui.update();
   }
}

public void setGUI(ITillGUI theGUI)
{  gui = theGUI;
}

private int tillFloat;
private int takings;
private int transactionCount;
private int amountOwedByThisCust;

private ITillGUI gui;
```


Highlights of the `TillGUI` class definition

```
public class TillGUI extends JPanel
   implements ITillGUI
{
  public TillGUI(Till theTill)
  { till = theTill;
    tc = new TillController(till, this);

    // Create the four main parts of the GUI
    JPanel custP = createCustPanel();
    JPanel numPadP = createNumPadPanel();
    JPanel actionP = createActionPanel();
    JPanel tillTotP = createTillTotPanel();

    // Layout the 4 panels one beneath the
    // other using GridBagLayout

    // Initialise the fields in the GUI
    update();
  }
```



```
public int getPriceEntered()
{ return Integer.parseInt(priceTF.getText())
}

public void setPriceEntered(
  int thePriceEntered)
{ priceTF.setText("" + thePriceEntered);
}

public void update()
{ priceTF.setText("" + 0);
  owedTF.setText("" + till.getTotalOwed());
  floatTF.setText("" + till.getTillFloat());
  tillTotalTF.setText(
   "" + till.getTillTotal());
  transTF.setText(
   "" + till.getTransactionCount());
}

// Registering Enter/Submit/Cancel listener

enterB.addActionListener(tc);
submitB.addActionListener(tc);
cancelB.addActionListener(tc);
```



```
// Registering numeric buttons' listener

ActionListener lstnr = new ActionListener()
{ public void actionPerformed(ActionEvent ae
  { String btnText = ae.getActionCommand();
    int digit = Integer.parseInt(btnText);
    int priceSoFar = getPriceEntered();
    int newPrice = priceSoFar * 10 + digit;
    setPriceEntered(newPrice);
  }
};
for (int i = 0; i < 10; i++)
{ digitB[i].addActionListener(lstnr);
}

// Variable declarations include:

private Till till;
private TillController tc;
```


The `TillController` class: listener for 3 buttons

```
public class TillController
  implements ActionListener
{ public TillController(
     Till theTill, ITillGUI theGUI)

  public void actionPerformed(ActionEvent ae
  { String str = ae.getActionCommand();
    if (str.equals(ITillGUI.ENTER))
    { till.ringUpItem(gui.getPriceEntered())
    }
    else if (str.equals(ITillGUI.SUBMIT))
    { till.ringUpItem(gui.getPriceEntered())
      till.submitTransaction();
    }
    else // i.e. cancel
    { till.cancelTransaction();
    }
  }

  private Till till;
  private ITillGUI gui;
}
```

`ITillGUI`: a Java interface

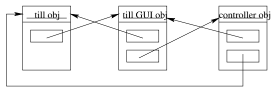
```
public interface IGUI
{
   public int getPriceEntered();
   public void update();

   public static final String
      ENTER = "Enter", SUBMIT = "Submit";
      CANCEL = "Cancel";
}
```

The `main` method

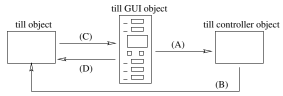
```
public static void main(String[] args)
{  Till till = new Till(1000);
   ITillGUI gui = new TillGUI(till);
   till.setGUI(gui);

   // Add the gui to a frame; size the frame
   // make the frame visible
}
```


The architecture of the program

The till object has a reference to the GUI object; the GUI object has references to the till object and the controller object; the controller object has references to the GUI and the till object:



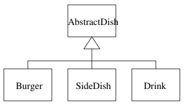
When a user presses Enter/Submit/Done, the controller invokes setters in the till object; the till object invokes a method in the GUI object:



(A) User clicks button; handler in controller is invoked
(B) Handler invokes setters in till object to update till object
(C) Till object invokes method in GUI object, so that GUI gets updated
(D) GUI invokes getters in till object to find out what to update itself with

Derek's Burger Bar: MVC at several levels

- A `BurgerBar` object has instance variables for
  - its bill-of-fare: this is a `Vector` of foodstuffs that are available; the foodstuffs are created from this class hierarchy:



  - a `DefaultListModel` object (to keep track of the current customer's order); and
  - a `Till` object
- MVC architecture: the `BurgerBar` and its `BurgerBarGUI` object, the `Till` and its GUI; the current order (list)

The `BurgerBar` class

```
public class BurgerBar
{
   public BurgerBar()
   {  till = new Till(TILL_FLOAT);
      currentOrder = new DefaultListModel();
      billOfFare = new Vector();
      // Now put stuff into this Vector,
      // representing the dishes available
      // in this burger bar.
   }

   public Till getTill()
   public Vector getBillOfFare()
   public ListModel getCurrentOrder()
```



```
public void addItemToOrder(
  AbstractDish theItem)
{ currentOrder.insertElementAt(theItem, 0);
  till.ringUpItem(theItem.getPrice());
}

public void removeItemFromOrder(
  int theItemIndex)
{ AbstractDish item = (AbstractDish)
      currentOrder.elementAt(theItemIndex);
  currentOrder.removeElementAt(theItemIndex)
  till.ringUpItem( - item.getPrice());
}

public void concludeOrder()
{ till.submitTransaction();
  currentOrder.removeAllElements();
}
```


Listeners

```
// Bill-of-fare buttons
public void actionPerformed(ActionEvent ae)
{  burgerBar.addItemToOrder(item);
}

// Delete button
public void actionPerformed(ActionEvent ae)
{  int index = list.getSelectedIndex();
   if (index != -1)
   {  burgerBar.removeItemFromOrder(index);
   }
}

// Next customer button
public void actionPerformed(ActionEvent ae)
{  burgerBar.concludeOrder();
}
```

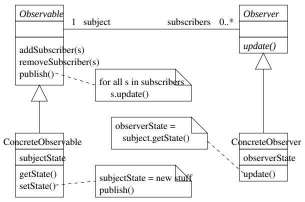
MVC conclusions

- You can now understand any Swing component
  - Many are simple: `JButton`, `JRadioButton`, `JCheckBox`, etc.
  - Others are simple if used in a limited way, but to use properly you have to understand MVC: `JList`, `JComboBox`, `JTable`, `JTree`, various text components
- You've seen that we also use MVC so that our whole program has an architecture that exhibits high cohesion & low coupling

Observer Design Pattern

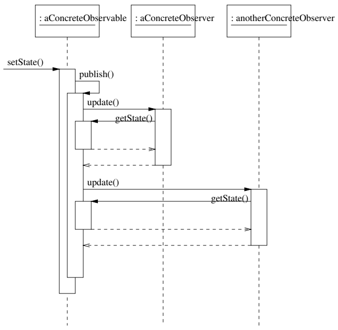
- *Name:* Observer (also known as Observable-Observer, Subject-Observer, Dependents, Publish-Subscribe, Source-Listener, Source-Support-Listener, Producer-Bus-Consumer, Model-View, Model-View-Controller, Document-View, Source-Distributed Listener)
- *Problem:* A one-to-many dependency between objects; when one changes state, all its dependents are notified and updated so that consistency is restored — without tight coupling.
- *Solution:*
  - Observers subscribe to observable
  - When observable changes, it notifies its observers
  - Each observer queries the observable and uses this to update itself

Observer Design Pattern, cont'd



(Alternative: use Java interfaces instead of abstract superclasses)

Observer Design Pattern, cont'd



Observer Design Pattern, cont'd

- *Consequences:*
  - can vary an observable's observers easily
  - low coupling: only indirect coupling between observers; observable doesn't need to know much about each observer
  - changes can be broadcast to any number of observers (message traffic is lower than it would be in a tightly coupled system!)
  - But, one small change can launch a torrent of other changes