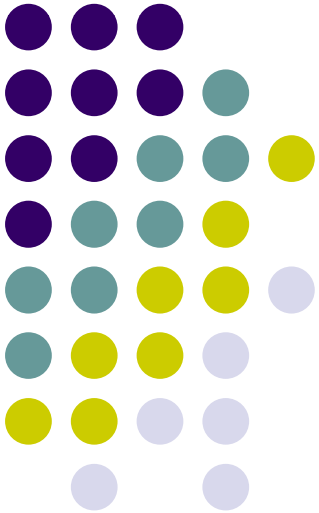
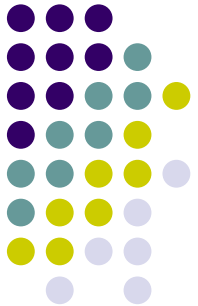
CSI2441: Applications Development

Lecture 7

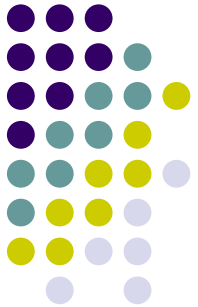
Event-Driven Programming with Graphical User Interfaces





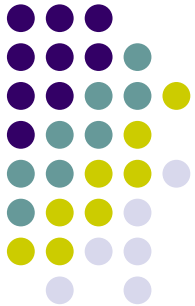
Objectives

- Understand the principles of event-driven programming
- Describe user-initiated actions and GUI components
- Design graphical user interfaces
- Modify the attributes of GUI components
- List the steps to building an event-driven application



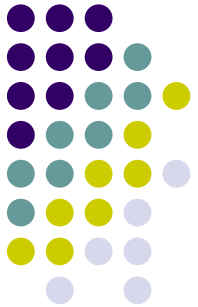
Objectives (continued)

- Understand the disadvantages of traditional error-handling techniques
- Understand the advantages of the object-oriented technique of throwing exceptions



Understanding Event-Driven Programming

- **Command line**: screen location where you type entries – very rare in the web dev world we find ourselves in
- **Icons**: pictures representing objects or actions
- **Graphical user interface** (**GUI**): allows users to interact with an application by clicking icons to select options

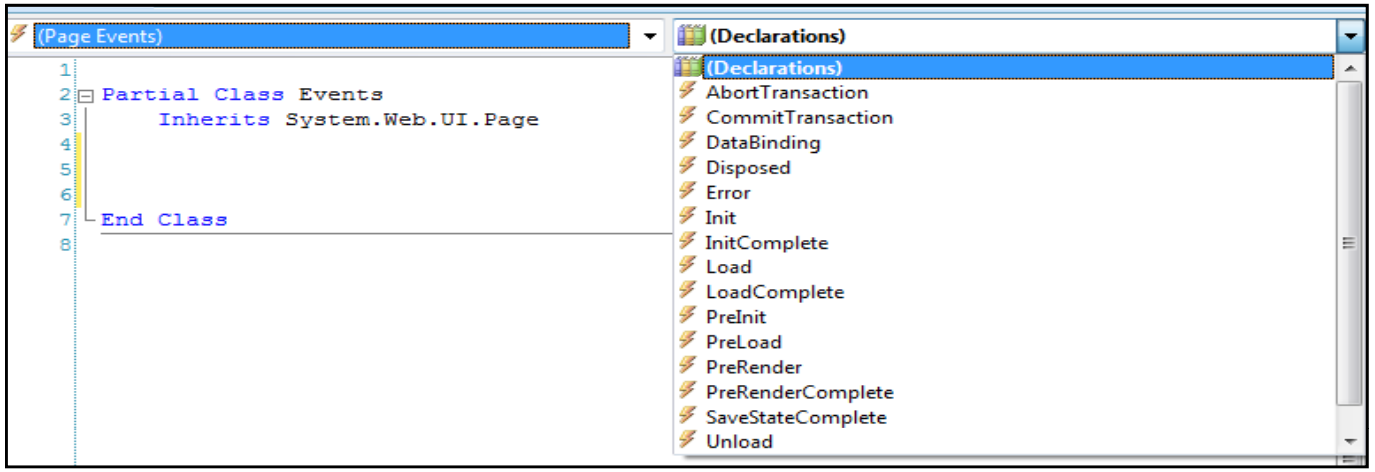


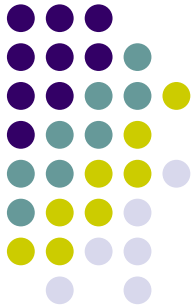
Understanding Event-Driven Programming (continued)

- **Event**: an occurrence that generates a message to an object
- **Event-based** (or **event-driven**): actions occur in response to user-initiated events
- When programming with event-driven languages, the emphasis is on:
  - GUI objects the user can manipulate
  - Events the user can initiate with those objects
- In web dev environments we use any number of events to drive our applications and interfaces
  - Hyperlinks of course
  - JavaScript and Flash events with on-screen objects
  - Form based post-backs in ASP.Net and other web programming environments

Understanding Event-Driven Programming (continued)

- Procedural programming controls the sequence of events
- In event-driven programming, the sequence is controlled by the user's actions – particularly in web apps
- **Source of the event**: a component from which an event is generated
- **Listener**: an object that will respond to the event
- Programmer writes the code that defines what actions are taken when the event occurs



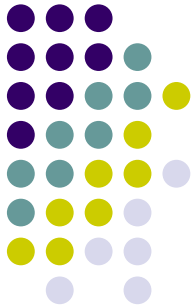


User-Initiated Actions and GUI Components

- Common user-initiated events:

TABLE 14-1: COMMON USER-INITIATED EVENTS

Event	Description
Key press	Pressing a key on the keyboard
Mouse point	Placing the mouse pointer over an area on the screen
Mouse click or left mouse click	Pressing the left mouse button
Right mouse click	Pressing the right mouse button
Mouse double-click	Pressing the left mouse button two times in rapid sequence
Mouse drag	Holding the left mouse button down while moving the mouse over the desk surface



User-Initiated Actions and GUI Components (continued)

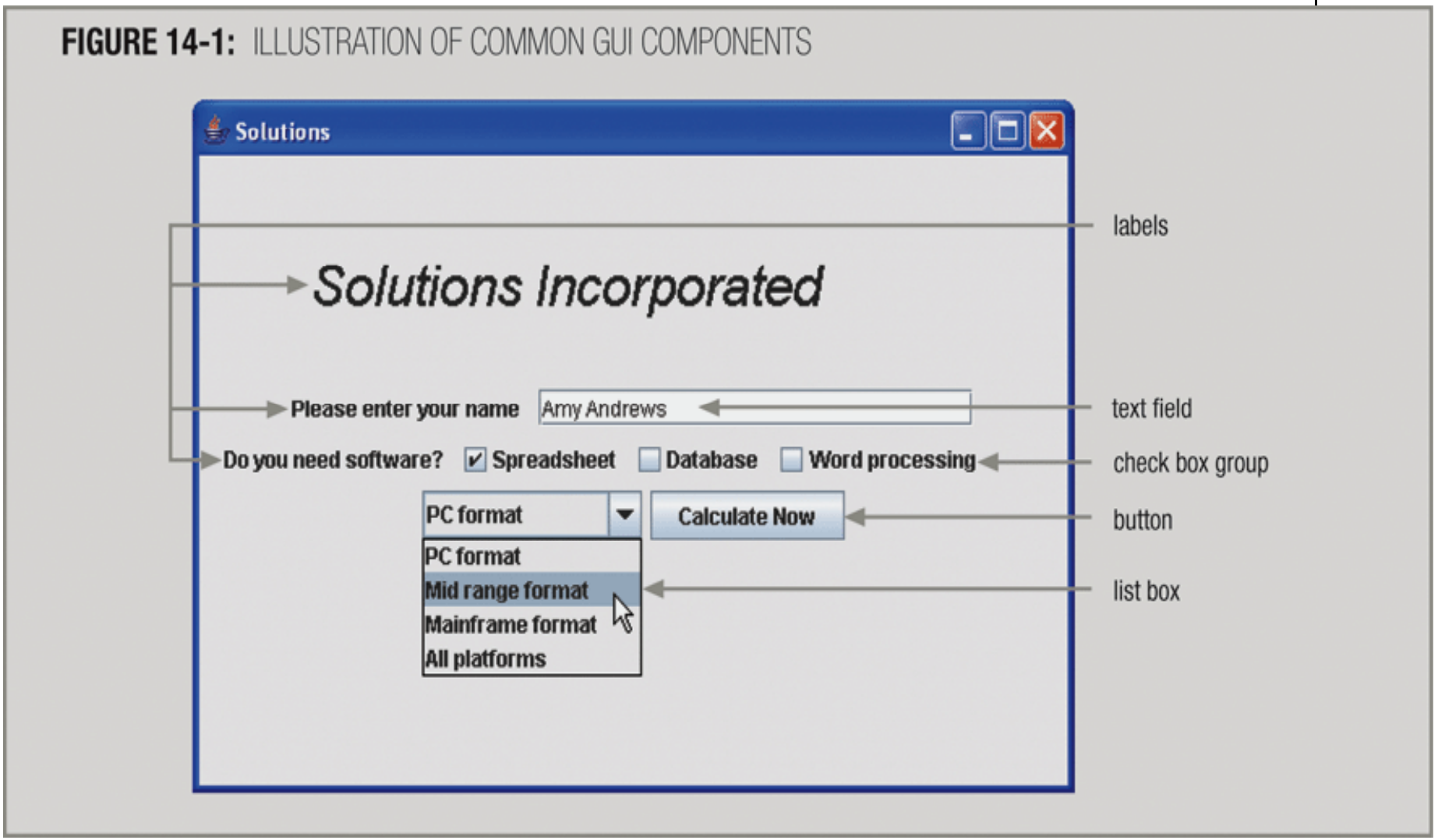
- Common GUI components:

TABLE 14-2: COMMON GUI COMPONENTS

GUI components	Description
Label	A rectangular area that displays text
Text field	A rectangular area into which the user can type a line of text
Button	A rectangular object you can click; usually it appears to press inward like a push button
Check box	A label positioned beside a square; you can click the square to display or remove a check mark—allows the user to turn an option on or off
Option buttons	A group of check-box-type objects in which the options are mutually exclusive; when the user selects any one option, the others are turned off—when the objects are square, they are often called a check box group, whereas when they are round, they are often called a set of radio buttons
List box	A menu of options that appears when the user clicks a list arrow; when the user selects an option from the list, the selected item replaces the original item in the display—all other items are unselected (with some list boxes, the user can make multiple selections)
Toolbar	A strip of icons that activate menu items

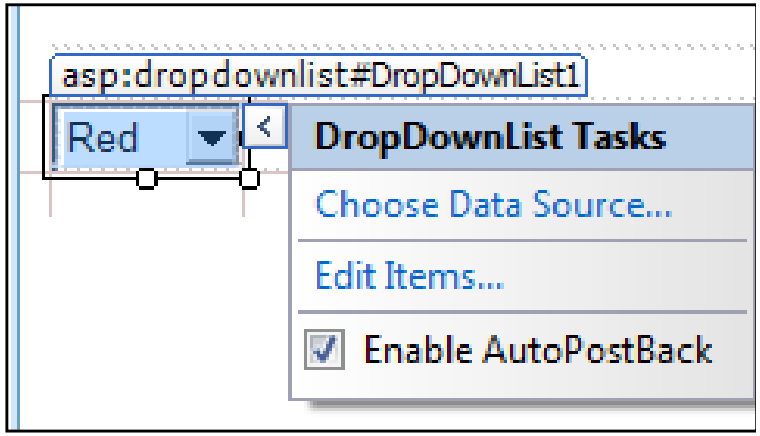
User-Initiated Actions and GUI Components (continued)

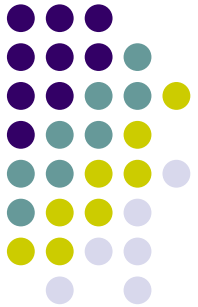
FIGURE 14-1: ILLUSTRATION OF COMMON GUI COMPONENTS



User-Initiated Actions and GUI Components (continued)

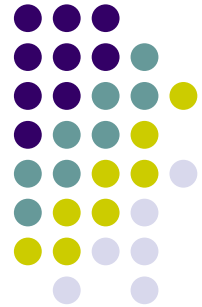
- GUI components are usually predefined classes that act like black boxes
- Programmer must write statements to manipulate GUI objects and take actions when events occur
- The ASP.Net drop-down list Control shown here has a setting which automatically posts the selected item back into the form as soon as the user selects a value
- This value can then be trapped in a isPostBack event and then dealt with accordingly





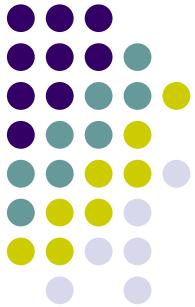
Designing Graphical User Interfaces

- Design principles for creating a GUI program:
  - Interface should be natural and predictable (consistent)
  - Screen design should be attractive and user-friendly
  - Allow the user to customize the application if possible
    - In many applications this is available to a certain degree, but from a programming perspective it means lots of extra work
    - For example, what if you allowed users to edit their background and foreground colours, and they selected the same colour for both?
  - Program should be forgiving
    - See above – ie you should never let users ‘paint’ themselves into a corner
  - GUI is only a means to an end



The Interface Should Be Natural and Predictable

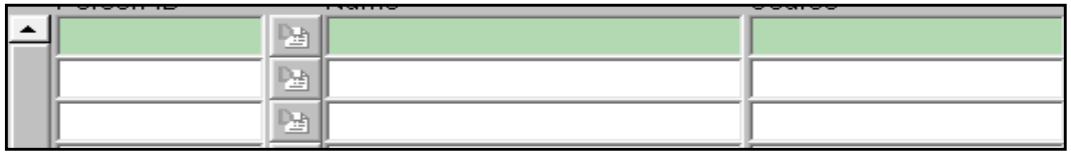
- Natural: icons that look like their real-world counterparts
- Predictable: use standard icons for the same purpose
- Layout of the screen should be predictable
  - As discussed in a previous lecture, menus menu position and menu options should be consistent across all pages
  - Use the same icons on each page – for example do not have a ← arrow on one page and ***Back*** on another to do the same job
  - If you change wording or iconography most users will assume it is for a reason and become confused (followed closely by aggravated)

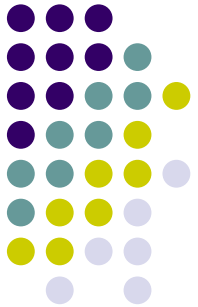


The Screen Design Should be Attractive and User-Friendly

- Attractive interfaces are more likely to be used
  - Though this is of course highly subjective
- Fancy fonts and weird colour combinations are signs of amateur designers!
- Unavailable or inappropriate screen or menu options should be disabled, invisible or “grayed out”
- Screen design should not be distracting
  - No flashing, blinking, noise or high contrast events
  - Interfaces should not rely on 3rd party plug-ins – most users will not install them

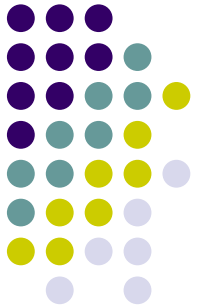
Scroll bar on right





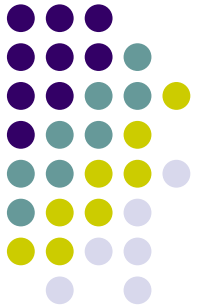
It's Helpful if the User Can Customize Your Applications

- If possible, allow users to:
  - Arrange components in the order they want (such as menu options)
  - Change colour schemes
  - Change their profile image
  - Change what information is publically viewable and which is not



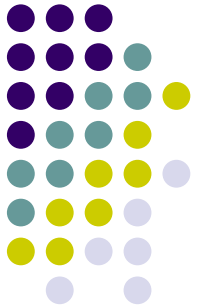
The Program Should be Forgiving

- Good programming design:
  - Always provides an escape route from bad choices
  - Allow user to back out of “dead end” choices
  - As stated previously, if a user has to use the Back button in their web browser, then something has been done seriously wrong in terms of interface design
- Do not expose users to record id’s if not necessary
  - Record id’s are there for the database and application to use in the background, for the most part they should not be near the user interface
  - Too many applications in the past have asked users to search, edit or delete by record id number which is the mark of appalling application design
  - Users cannot be expected to know the underlying workings of a system – they are there to use it to get a job done



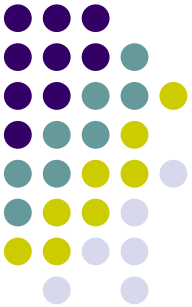
The GUI is Only a Means to an End

- GUI is only an interface, not the purpose of the program
- GUI should help make users more productive
- The GUI design should:
  - Help users see what options are available
  - Allow components to be used as normal
  - Not force the user to figure out how to interact and how the underlying logic of the application works
- The real work of a GUI program is done *after* the user makes an action
  - Good interface design is hard to achieve and is time consuming
  - Bad interface design is quick and easy but will always come back to haunt the developer (in terms of user complaints)



Modifying the Attributes of GUI Components

- Appearance of GUI components can be changed by modifying the attributes
- Common types of changes:
  - Size
  - Color
  - Screen location
  - Font for text in the component
  - Visible or invisible
  - Enabled or disabled (dimmed or undimmed)



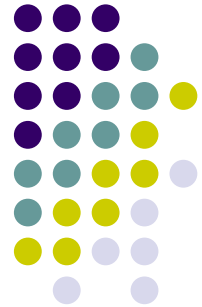
GridView Example

asp:GridView#GridView1

	ProductID	ProductName	Supp
Edit Delete Select	0	abc	0
Edit Delete Select	1	abc	1
Edit Delete Select	2	abc	2
Edit Delete Select	3	abc	3
Edit Delete Select	4	abc	4
Edit Delete Select	5	abc	5
Edit Delete Select	6	abc	6
Edit Delete Select	7	abc	7
Edit Delete Select	8	abc	8
Edit Delete Select	9	abc	9

Design | Split | Source | <asp:GridView#Grid

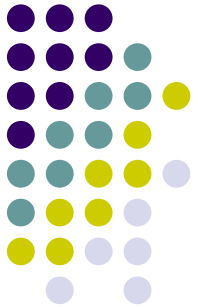
Appearance	
BackColor	
BackImageUrl	
BorderColor	
BorderStyle	NotSet
BorderWidth	
CssClass	
EmptyDataText	
Font	
ForeColor	#333333
GridLines	**None**
ShowFooter	False
ShowHeader	True
Behavior	
AllowSorting	**True**
AutoGenerateColumns	**False**
AutoGenerateDeleteButton	False
AutoGenerateEditButton	False
AutoGenerateSelectButton	False
Enabled	True
EnableModelValidation	False
EnableSortingAndPagingCallbacks	False
EnableTheming	True
EnableViewState	True
SkinID	
ToolTip	
Visible	True



Modifying the Attributes of GUI Components (continued)

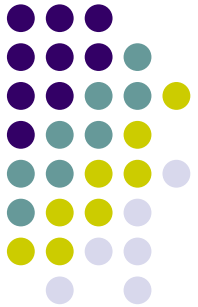
- Different languages have different ways to change attribute values:
  - With coding statements
  - By calling a method and passing arguments
  - Accessing a properties list or table (as in previous slide)
  - All of the above
- The code below shows a Page_LoadComplete event in ASP.Net which in turn contains instructions which programmatically make the previously shown GridView Control invisible be default

```
Protected Sub Page_LoadComplete(ByVal sender As Object, ByVal e As System.EventArgs) Handles Me.LoadComplete

    GridView1.Visible = False

End Sub
```



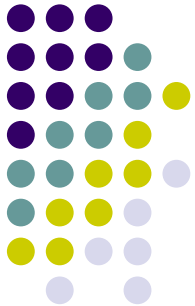
The Steps to Developing an Event-Driven Application

- Steps to developing any program:
  - Understand the problem
  - Plan the logic
  - Code the program
  - Translate the program into machine language
  - Test the program
  - Put the program into production



The Steps to Developing an Event-Driven Application (continued)

- Three steps added for event-driven programs:
  - Understand the problem
  - Create storyboards
  - Define the objects
  - Define the connections between screens
  - Plan the logic
  - Code the program
  - Translate the program into machine language
  - Test the program
  - Put the program into production



Understanding the Problem

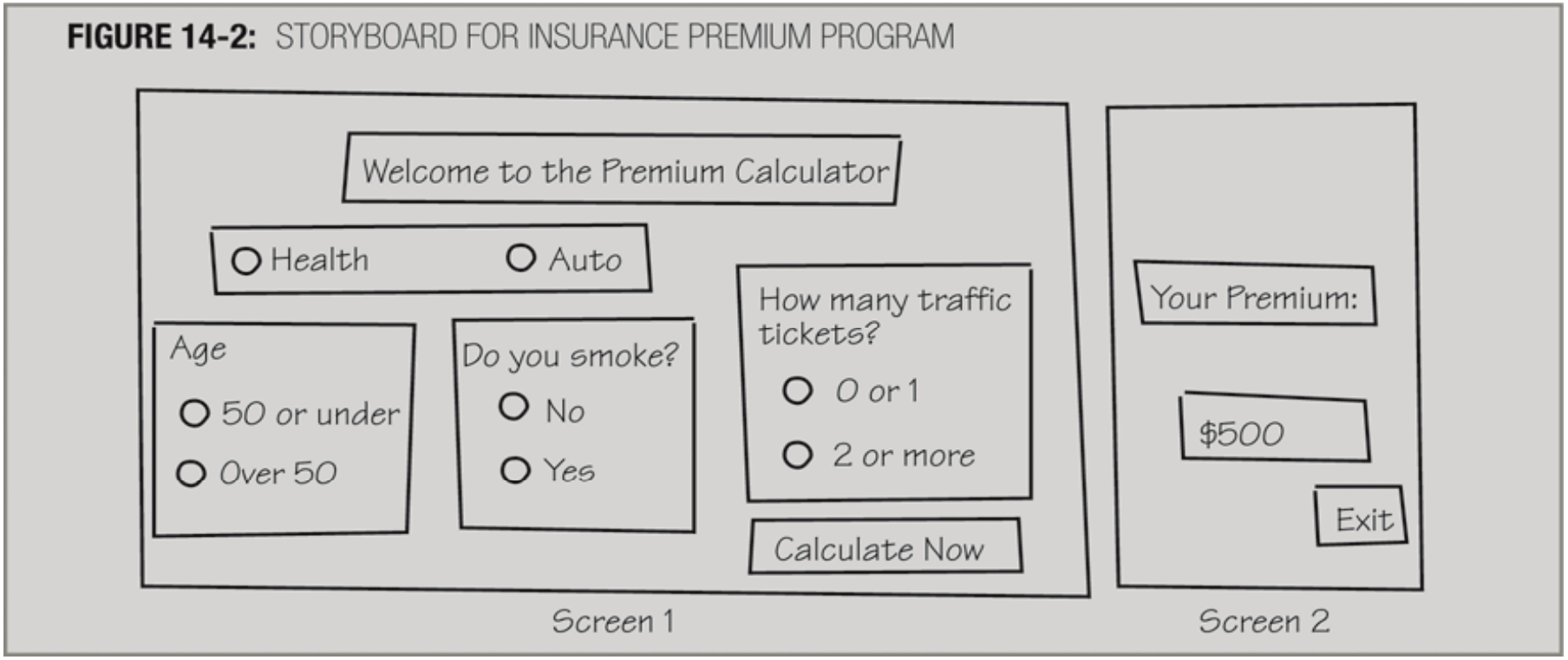
- Developing the application: (try this one in the lab ☺)

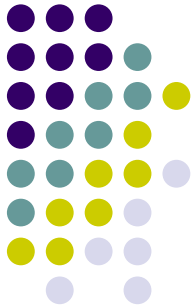
TABLE 14-3: INSURANCE PREMIUMS BASED ON CUSTOMER CHARACTERISTICS

Health policy premiums	Auto policy premiums
Base rate: $500	Base rate: $750
Add $100 if over age 50	Add $400 if more than 2 tickets
Add $250 if smoker	Subtract $200 if over age 50

Creating Storyboards

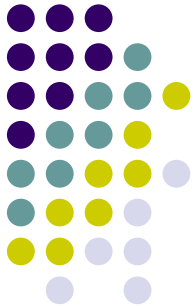
- **Storyboard**: a picture or sketch of the user screen
- Draw one storyboard cell/frame for each user screen





Defining the Objects in an Object Dictionary

- Object dictionary:
  - List of objects used in a program
  - Indicates which screens the objects are used in
  - Indicates if code or script is associated with the object
  - May show which variables are affected by an action on the object
- This is the kind of thing which would be delivered as part of a technical manual (not user manual) and by the application coders during development



Defining the Objects in an Object Dictionary (continued)

FIGURE 14-3: OBJECT DICTIONARY FOR INSURANCE PREMIUM PROGRAM

Object type	Object name	Screen number	Variables affected	Script?
Label	welcomeLabel	1	none	none
Choice	healthOrAuto	1	policyType	none
Choice	age	1	ageOfInsured	none
Choice	smoker	1	insuredIsSmoker	none
Choice	tickets	1	numTickets	none
Button	calculateButton	1	premiumAmount	calcRoutine()
Label	yourPremium	2	none	none
Text field	premAmtField	2	none	none
Button	exitButton	2	none	exitRoutine()

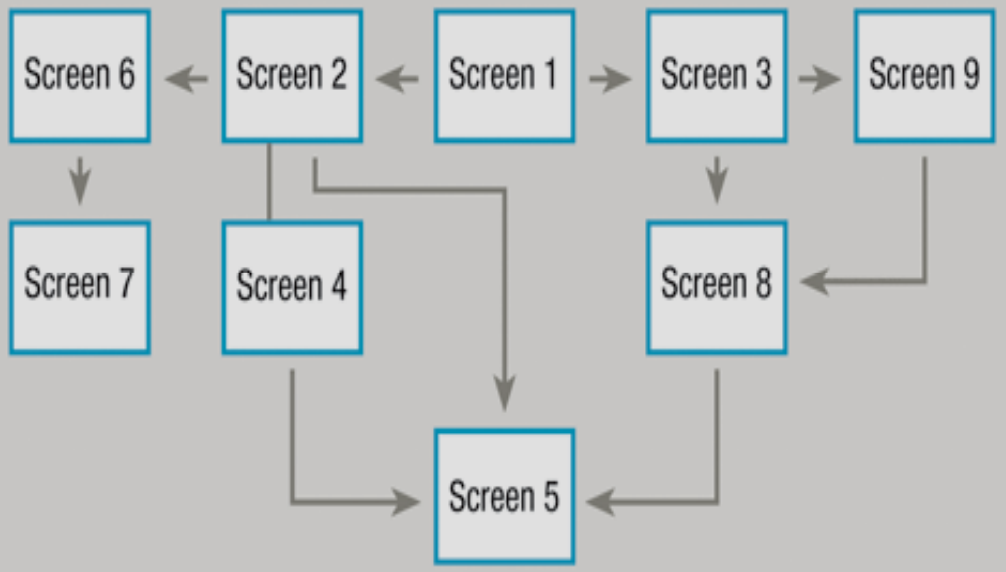
Defining the Connections Between the User Screens

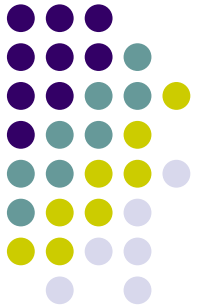
- **Interactivity diagram**: shows the relationship between screens in an interactive GUI program

FIGURE 14-4: DIAGRAM OF INTERACTION FOR INSURANCE PREMIUM PROGRAM

Defining the Connections Between the User Screens (continued)

FIGURE 14-5: DIAGRAM OF INTERACTION FOR A HYPOTHETICAL COMPLICATED PROGRAM





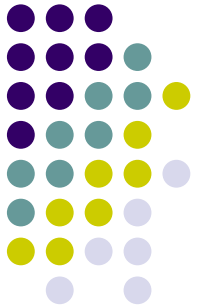
Planning the Logic

- After you have:
  - Designed the screens
  - Defined the objects
  - Defined how screens will connect
- Then you can plan the logic

Planning the Logic (continued)

FIGURE 14-6: PSEUDOCODE FOR `calcRoutine()`

```
calcRoutine()
   const char HEALTH = "H"
   const char YES = "Y"
   const num BASE_PREMIUM_HEALTH = 500
   const num AGE_CUTOFF = 50
   const num AGE_HEALTH_EXTRA = 100
   const num SMOKER_EXTRA = 250
   const num BASE_PREMIUM_AUTO = 750
   const num TICKET_CUTOFF = 2
   const num TICKET_EXTRA = 400
   const num AGE_AUTO_DISCOUNT = -200
   if policyType = HEALTH then
      premiumAmount = BASE_PREMIUM_HEALTH
      if ageOfInsured > AGE_CUTOFF then
          premiumAmount = premiumAmount + AGE_HEALTH_EXTRA
      endif
      if insuredIsSmoker = YES then
          premiumAmount = premiumAmount + SMOKER_EXTRA
      endif
   else
      premiumAmount = BASE_PREMIUM_AUTO
      if numTickets > TICKET_CUTOFF then
          premiumAmount = premiumAmount + TICKET_EXTRA
      endif
      if ageOfInsured > AGE_CUTOFF then
          premiumAmount = premiumAmount + AGE_AUTO_DISCOUNT
      endif
   endif
return
```



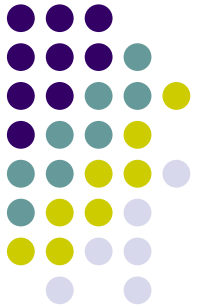
Understanding the Disadvantages of Traditional Error-Handling Techniques

- Validating user input is a large part of any program
- GUI data entry objects may help control what a user can enter
- One error-handling technique was to terminate the program – very unforgiving!

Understanding the Disadvantages of Traditional Error-Handling Techniques (continued)

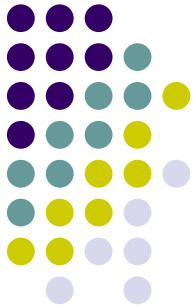
FIGURE 14-7: UNFORGIVING, UNSTRUCTURED METHOD OF ERROR HANDLING

```
calcRoutine()
   const char HEALTH = "H"
   const char AUTO = "A"
   const char YES = "Y"
   const num BASE_PREMIUM_HEALTH = 500
   const num AGE_CUTOFF = 50
   const num AGE_HEALTH_EXTRA = 100
   const num SMOKER_EXTRA = 250
   const num BASE_PREMIUM_AUTO = 750
   const num TICKET_CUTOFF = 2
   const num TICKET_EXTRA = 400
   const num AGE_AUTO_DISCOUNT = -200
   if policyType not = HEALTH AND policyType not = AUTO then
      return
   else
      if policyType = HEALTH then
         premiumAmount = BASE_PREMIUM_HEALTH
         if ageOfInsured > AGE_CUTOFF then
             premiumAmount = premiumAmount + AGE_HEALTH_EXTRA
         endif
         if insuredIsSmoker = YES then
             premiumAmount = premiumAmount + SMOKER_EXTRA
         endif
      else
         premiumAmount = BASE_PREMIUM_AUTO
         if numTickets > TICKET_CUTOFF then
             premiumAmount = premiumAmount + TICKET_EXTRA
         endif
         if ageOfInsured > AGE_CUTOFF then
             premiumAmount = premiumAmount + AGE_AUTO_DISCOUNT
         endif
      endif
   endif
return
```



Understanding the Disadvantages of Traditional Error-Handling Techniques (continued)

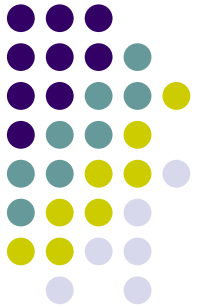
- Can create an error flag:
  - Prompt the user to re-enter a valid value
  - Loop until the data item becomes valid
- Disadvantages:
  - Makes module less reusable
  - Not as flexible as it might be
  - Only works with interactive programs



Understanding the Disadvantages of Traditional Error-Handling Techniques (continued)

FIGURE 14-8: USING A LOOP TO HANDLE INTERACTIVE ERRORS

```
calcRoutine()
   const char HEALTH = "H"
   const char AUTO = "A"
   const char YES = "Y"
   const num BASE_PREMIUM_HEALTH = 500
   const num AGE_CUTOFF = 50
   const num AGE_HEALTH_EXTRA = 100
   const num SMOKER_EXTRA = 250
   const num BASE_PREMIUM_AUTO = 750
   const num TICKET_CUTOFF = 2
   const num TICKET_EXTRA = 400
   const num AGE_AUTO_DISCOUNT = -200
   num errorFlag = 1
   while errorFlag = 1
      errorFlag = 0
      if policyType = HEALTH then
         premiumAmount = BASE_PREMIUM_HEALTH
         if ageOfInsured > AGE_CUTOFF then
             premiumAmount = premiumAmount + AGE_HEALTH_EXTRA
         endif
         if insuredIsSmoker = YES then
             premiumAmount = premiumAmount + SMOKER_EXTRA
         endif
      else
         if policyType = AUTO then
            premiumAmount = BASE_PREMIUM_AUTO
            if numTickets > TICKET_CUTOFF then
               premiumAmount = premiumAmount + TICKET_EXTRA
            endif
            if ageOfInsured > AGE_CUTOFF then
               premiumAmount = premiumAmount + AGE_AUTO_DISCOUNT
            endif
         else
            print "Invalid policy type. Please reenter"
            read policyType
            errorFlag = 1
         endif
      endif
   endwhile
return
```



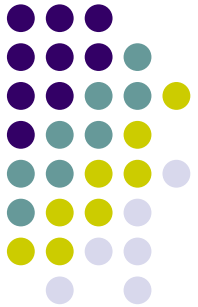
Understanding the Advantages of Object-Oriented Exception Handling

- Exception-handling methods:
  - Used in object-oriented, event-driven programs
  - Check for and manage errors
- Exception: an object that represents an error
- **Try**: to attempt to execute a module that might throw an error
- **Throw**: pass an exception from called module back to the caller
- **Catch**: receive an exception from a called module
- Exception object that is thrown can be any data type

Understanding the Advantages of Object-Oriented Exception Handling (continued)

FIGURE 14-9: THROWING AN EXCEPTION

```
calcRoutine()
   const char HEALTH = "H"
   const char AUTO = "A"
   const char YES = "Y"
   const num BASE_PREMIUM_HEALTH = 500
   const num AGE_CUTOFF = 50
   const num AGE_HEALTH_EXTRA = 100
   const num SMOKER_EXTRA = 250
   const num BASE_PREMIUM_AUTO = 750
   const num TICKET_CUTOFF = 2
   const num TICKET_EXTRA = 400
   const num AGE_AUTO_DISCOUNT = -200
   num errorFlag = 1
   if policyType = HEALTH then
       premiumAmount = BASE_PREMIUM_HEALTH
       if ageOfInsured > AGE_CUTOFF then
           premiumAmount = premiumAmount + AGE_HEALTH_EXTRA
       endif
       if insuredIsSmoker = YES then
           premiumAmount = premiumAmount + SMOKER_EXTRA
       endif
   else
      if policyType = AUTO then
         premiumAmount = BASE_PREMIUM_AUTO
         if numTickets > TICKET_CUTOFF then
             premiumAmount = premiumAmount + TICKET_EXTRA
         endif
         if ageOfInsured > AGE_CUTOFF then
             premiumAmount = premiumAmount + AGE_AUTO_DISCOUNT
         endif
      else
         throw errorFlag
      endif
   endif
return
```

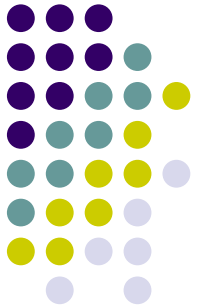


Understanding the Advantages of Object-Oriented Exception Handling (continued)

- **`try` block**: contains code that may throw an exception
- **`catch` block**: contains code that executes when an exception is thrown from within a **`try`** block

FIGURE 14-10: PROGRAM SEGMENT USING `calcRoutine()`

```
try
   perform calcRoutine()
endTry
catch(num thrownCode)
    policyType = "H"
    try
       perform calcRoutine()
    endTry
endCatch
// Program continues
```



Understanding the Advantages of Object-Oriented Exception Handling (continued)

- **`catch`** block
  - Specifies the data type of the exception it will handle
  - Executes only when an exception of that data type is thrown in the corresponding **`try`** block

FIGURE 14-11: ALTERNATE PROGRAM SEGMENT USING `calcRoutine()`

```
num thrownCode = 1
while thrownCode = 1
   try
      perform calcRoutine()
      thrownCode = 0
   endTry
   catch(num thrownCode)
      print "Reenter the policy type"
      read policyType
   endCatch
endwhile
// Program continues
```

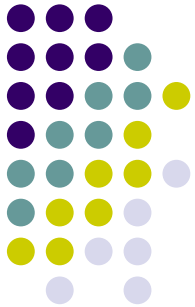
Understanding the Advantages of Object-Oriented Exception Handling (continued)

FIGURE 14-12: PROGRAM SEGMENT THAT DISPLAYS THROWN ERROR CODE

```
try
  perform calcRoutine()
endTry
catch(num thrownCode)
      print "Error #", thrownCode," has occurred"
endCatch
// Program continues
```

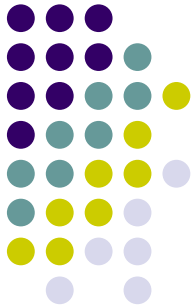
FIGURE 14-13: PROGRAM SEGMENT THAT SETS `premiumAmount` TO 0 WHEN EXCEPTION IS THROWN

```
try
  perform calcRoutine()
endTry
catch(num thrownCode)
      premiumAmount = 0
endCatch
// Program continues
```



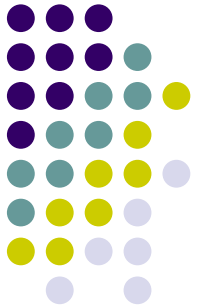
Understanding the Advantages of Object-Oriented Exception Handling (continued)

- General principle of exception handling in OOP:
  - Module that uses data should be able to detect errors, but is not required to handle them
  - Handling is left to the application that uses the object



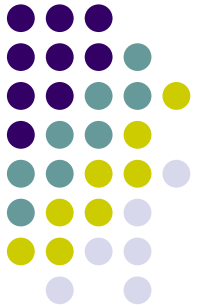
Summary

- Event-driven GUI interface allows users to manipulate objects on the screen
- Source of event: component that generates event
- Listener: an object interested in an event
- GUI components are created from predefined classes, and act like black boxes
- GUI interface should be neutral, predictable, attractive, easy to read, and non-distracting



Summary (continued)

- GUI program should allow user to customize the application and should be forgiving
- Modify the GUI component attributes to change its appearance
- Use storyboards to help plan a GUI program
- Traditional error–handling methods have limitations
- OOP error handling involves throwing exceptions



Readings

- Coding an IQ Test with Conditionally Driven Event Handlers in ASP.Net 3.5 (2010). http://www.aspfree.com/c/a/ASP.NET/Coding-an-IQ-Test-with-Conditionally-Driven-Event-Handlers-in-ASPNET-35-Part-2/

- Poor Interface Design Example (2007). http://www.bennadel.com/blog/869-Poor-Interface-Design-Example.htm