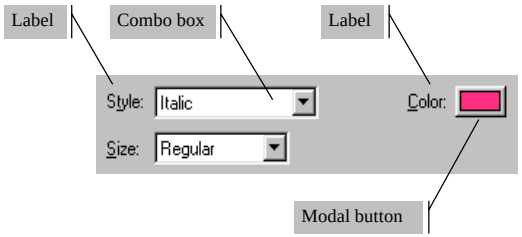
CSE 3461 W08

Text and Text Entry

Types of Text Components

- Output components
  - cannot be edited
    - Labels, Labeled borders (`JLabel`, `TitledBorder`)
    - Tool tips, Message Boxes
- Input/output components
  - can be edited
    - Text fields, Text areas, Editable combo boxes
    - Dialog boxes

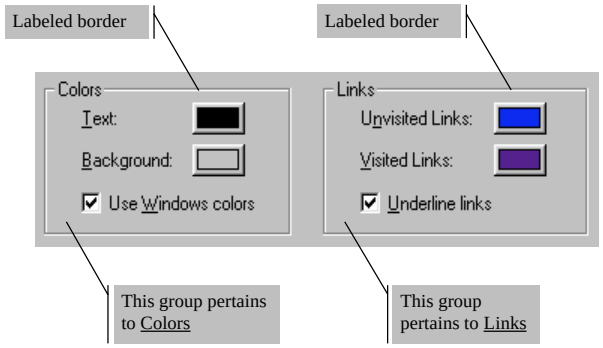
Labels: Example



Labels

- Create using `JLabel`
- Do not react to input events, cannot get keyboard focus
- Used to display information
  - in particular, placed adjacent to a component that has a keyboard alternative but can't display it
- Position is determined by Layout Manager
- Advantage:
  - The information it provides can be useful; aid user's performance of task
- Disadvantage:
  - Uses screen real estate
  - Poor wording may be worse than none at all

Labeled Borders: Example



Labeled Borders

- The `setBorder` method is defined for all instances of `JComponent`
  - Used to create visual clue about groupings
  - A label for the grouping is optional
- The parameter is an instance of a `Border`
- `Border` is an interface
  - `AbstractBorder` is an abstract class that implements it
  - `TitleBorder` extends `AbstractBorder`

E.g., in `DemoButtons`:

```
JPanel sizeGroupPanel = new JPanel();
sizeGroupPanel.setBorder(
    new TitledBorder(new EtchedBorder(), "Size"));
```


Text Fields and Text Areas

- Different types:
  - Text Field: single line
  - Text Area: multiple lines
  - The abstract class `JTextComponent` has the subclasses `JTextField`, `JTextArea`
- Challenges for design:
  - How to validate the text that has been input?
  - How to navigate within and between text elements?

Editable Combo Boxes

- Similar to a combo box, except that user may also enter text directly
- Same challenges as text fields and areas
  - validation, navigation
- Editable and non-editable combo boxes are both instantiated from `JComboBox`
  - use the methods: `setEditable(true)`, `setEditable(false)`

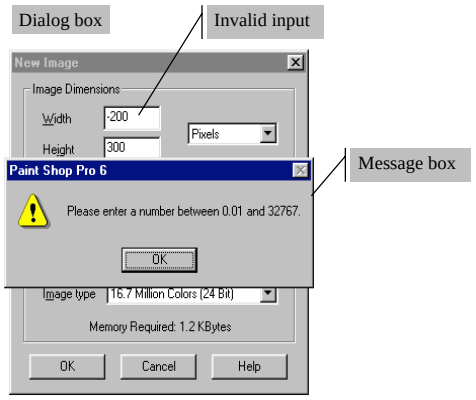
Navigation

- All components have a *focus state*
  - The possible focus states are *in focus* or *out of focus*
  - For a key press to affect a component, the component must have focus
  - Visual clues are given to show which component has focus
    - I-beam cursor appears, special highlighting
- Every time the focus changes, a `FocusEvent` is generated
  - a component loses focus, another gains focus,

Navigation

- A component generally gains the focus by the user:
  - clicking it
  - tabbing to it, or
  - otherwise interacting with a component.
- A component can also be given the focus programmatically
  - e.g., a component can request the focus when its containing frame or dialog is made visible
- The *focus traversal policy* determines the order in which a group of components are navigated

Message Box: Example



Message Boxes

- A *message box* (aka *dialog box*) is a popup window
- Primary purpose is to govern the interaction
  - presents a text message to the user
  - seeks input for confirmation (and to close the box)
- Functions:
  - Notify the user of a problem (e.g., invalid choice)
  - Notify the user of potentially destructive outcome (e.g., overwrite a file)
  - Provide information

Message Boxes vs Tool Tips

- Size/Complexity of message
  - message boxes allow more text
- Impact on flow of interaction
  - message boxes must be dismissed with user input action
  - tool tips can be made to disappear through little user action
  - message boxes demand immediate attention
  - can't close the message box (user is required to make a choice or to provide confirmation)
  - user is not able to make use of other widgets

Input-Handling Techniques

For actions with potentially serious consequences:

- require an explicit button click or key press before proceeding
- disable "enter" (which is otherwise the default action)
- when is it better to allow the action and provide undo vs. the advanced warning?

Input-Handling Techniques

For invalid input:

- provide feedback (e.g., alarm tone or visual feedback)
- feedback should not only indicate the problem, but also how it can be fixed

- Take advantage of user consistency (e.g., change the position of buttons from one invocation to the next)

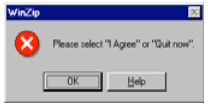
Input-Handling Techniques

- Be aware of that many users are conditioned by too many message boxes:
  - Some dismiss them with out reading their contents
  - Can subvert this by being inconsistent

Example

Button positions change from one invocation to next

Hitting ENTER produces…



Example

DemoMessageBox.java

Validating Input

- How to handle input?
  - Should it be in the format required by the application or in the format that the user wants to (is able to) provide?
  - The distinction between valid and invalid data
  - Semantic Error vs. Format Error
- When to intervene?
  - Solicit Input ⇨ Accept ⇨ Proceed to next step
- What sort of feedback?
  - popup message, generate audio alarm, system behaviour…

Input Validation

What type of text-based information might users provide?

- Numeric values
  - weight, age, font size
  - Are floating point values allowed? How many decimal places?
  - Are negative numbers allowed?
- Text
  - first name, surname, font name
- Mixed
  - Postal or zip code
  - License plate number
  - Date
  - Times

- Semantic Error vs. Format Error

Text Fields and Text Areas

- Given an instance of a text field or area:
  - The contents of the instance are contained in a data model, say `d`
  - The method `getText()` will return the contents of `d` as an instance of a `String`
  - The default data model is `PlainDocument`

Text Fields and Text Areas

- How can the user cause some text to be entered?
  - _____
  - Let us denote this text by `String s`
- This action causes the `insertString` method of `d` to be invoked
  - this causes the string `s` to be added to the data model (at the appropriate offset)

Strategies for Validation

- Suppose we want our text field to accept only strings of digits
- What possible strategies might be used to validate this?
  1. _____
  2. _____
  3. _____

Strategy #1: Keystroke-level

- Register a `KeyListener` on the text component
  - when the user types a key, an key event `ke` is generated
  - if you don't intercede, then the character gets passed to the insertString method of the data model
    - use the method `ke.getKeyChar()` to access the character
  - you can intercede
    - you can **consume** the event

Strategy #2: Focus-level

- Register a `FocusListener` on the text component
  - when the user tries to transfer focus to another component, a focus event `fe` is generated
  - check what contents of the text component is
    - if the contents are invalid, don't let the focus be transferred away

Strategy #3: Data Model-level

- Use a different data model than `PlainDocument`
  - recall that the `insertString` method gets invoked when the user causes text to be entered
  - if you override the method, then you can inspect the text that was entered
    - need to create a subclass of `PlainDocument`
    - if the entered text is valid, invoke the super's `insertString` method in order to add it to the data model
    - if not, don't add it to the data model