Java 2 Micro Edition
Creating a User Interface

F. Ricci

2010/2011

Contents

- General principles for building MIDLet GUI
- Main classes (the `Displayable` hierarchy)
- Event management with commands
- Different kind of screens: `TextBox`, `Alert`, `List`
- A flexible screen: `Form`
  - Items: `Gauge`, `TextField`, `StringItem`, `ImageItem`, `DataField`, `ChoiceGroup`
- Commands on items
- Responding to item's changes
- Summary of the `Displayable` hierarchy
- Summary of commands and events management.

Creating a User Interface

- Different devices with vary input capabilities
  - Screen size (minimum 96x54 pixels)
  - Keypads (numerical or alphabetic), soft keys
- **Abstraction**
  - Specify UI in abstract terms: e.g., “show somewhere the ‘next’ command”
  - Relying on the MIDP implementation to create something concrete: e.g., the MIDP implementation prints a ‘next’ string in the interface and associate the command to a key
- **Discovery**
  - The application learns about the device at runtime (.e.g. the foreground color) – and adapts to it
  - Used for games or graphic applications.

MIDP – User Interface

- Not a subset of AWT or Swing because:
  - AWT is designed for desktop computers
  - AWT is used in CDC configuration - Personal Profile
  - Assumes certain user interaction models (pointing device such as a mouse)
  - Window management (resizing overlapping windows). This is impractical for cell phones
- MIDP (`javax.microedition.lcdui`) consists of high-level and low-level APIs

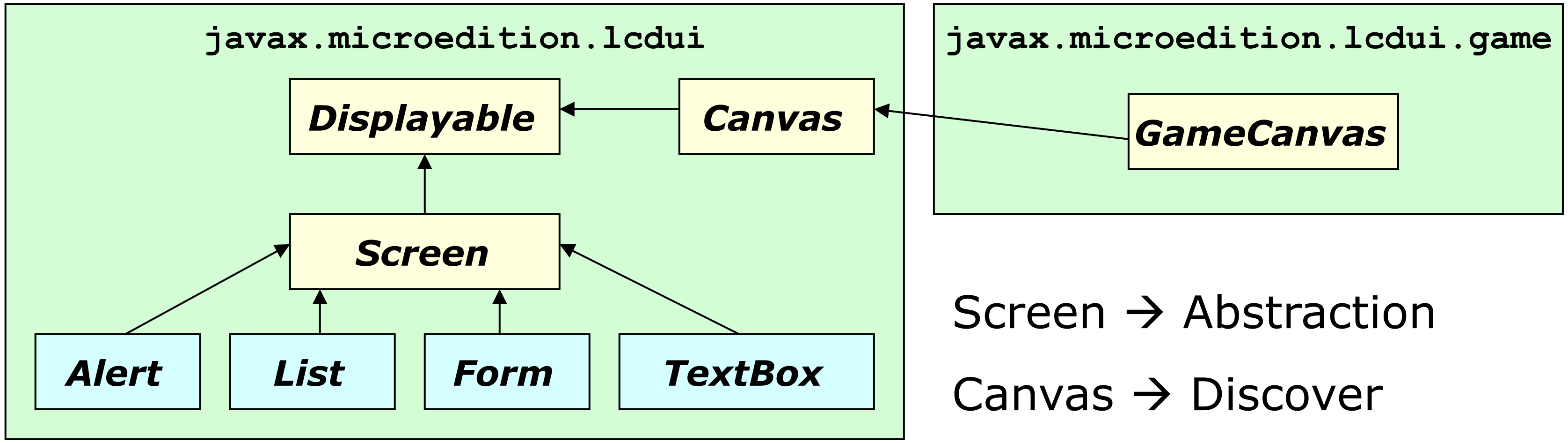
MIDP - UI APIS

- **High-level API**
  - Applications should be *runnable* and usable in all MIDP devices
  - No direct access to **native** device features
- **Low-level API**
  - Provide access to **native** drawing primitives, device key events, native input devices
  - Allows developers to choose to compromise portability for user experience
  - Examples:
    - draw in a canvas: `paint(Graphics g)`
    - call back when you press a key `keyPressed (intkeyCode)`

MIDP - UI Programming Model

- The central abstraction is a **screen**
- Only one screen may be visible at a time
- Three types of screens:
  - **Predefined screens** with complex UI components (`List`, `TextBox`)
  - **Generic screens** (`Form` where you can add text, images, etc)
  - Screens used with **low-level API** (`Canvas`)
- The `Display` class is the display **manager**
- It is instantiated for each active `MIDlet`
- Provides methods to retrieve information about the device’s display capabilities.

Displayable structure



Screen → Abstraction

Canvas → Discover

Basic function of a typical MIDlet

1. Show a `Displayable`
2. Wait for input
3. Decide what `Displayable` should be next
4. Repeat

The device’s display is represented by an instance of `Display` class.

🟨 = abstract class

Using `Display`

- `Display` manages device's screen
- Use `getDisplay()` (static method of `Display`) to access the display - in the `startApp()` method
- Use the returned `Display` object to determine device capabilities or current displayable
  - `isColor()` → color or grayscale device
  - `numColors()` → number of colors supported
  - `numAlphaLevels()` → number of transparency level
  - `getCurrent()` → a reference to what (displayable) currently being shown
- After creating something to show as an instance of `Displayable` you can display it:
  - `setCurrent(Displayable next)`
  - `setCurrent(Alert alert, Displayable nextDisplayable)`

Event Handling with Commands

- `Displayable` supports a flexible user interface concept: the **command**
- You can add and remove Commands (to a `Displayable`) using the following methods:
  - `Public void addCommand(Command cmd)`
  - `Public void removeCommand(Command cmd)`
- To create a command you have to supply a name, a type and a priority:
  - **Name** (usually shown on the screen)
  - **Type** (to identify the type of command)
  - **Priority** (the importance of a command respect to the others).

Command Types

OK	Confirm a selection
CANCEL	Cancel pending changes
BACK	Moves the user back to a previous screen
STOP	Stops a running operation
HELP	Show application instructions
SCREEN	Generic type for specific application commands
EXIT	A command used for exiting from the application
ITEM	Command used for items (focused item or element)

- The application uses the command type to specify the **intent** of this command
- Example:
  - If the application specifies that the command is of type BACK,
  - and **if the device has a standard** of placing the "back" operation on a certain soft-button,
  - then **the implementation can follow the style** of the device by using the semantic information as a guide.

Creating Commands

- A standard OK command

```
    Command c = new Command (“OK”, Command.OK, 0);
```

- A command specific to your application

```
Command c = new Command (“Launch”, Command.SCREEN,
  0);
```

- It is **up to the MIDP implementation** to figure out how to show the commands
- A simple priority scheme determines who wins when there are more commands than available screen space
  - *Low values = higher priority*
- The priority with low value will show on screen directly the other will most likely end up in a secondary menu.

Responding to commands

- An object called a **listener** is notified when the user invokes any command in a `Displayable`
- The listener is an object that implements the `CommandListener` **interface**
- To **register** the listener with a **Displayable** use:

```
public void setCommandListener(CommandListener l)
```

- Implementing a `CommandListener` is a matter of defining a single method:

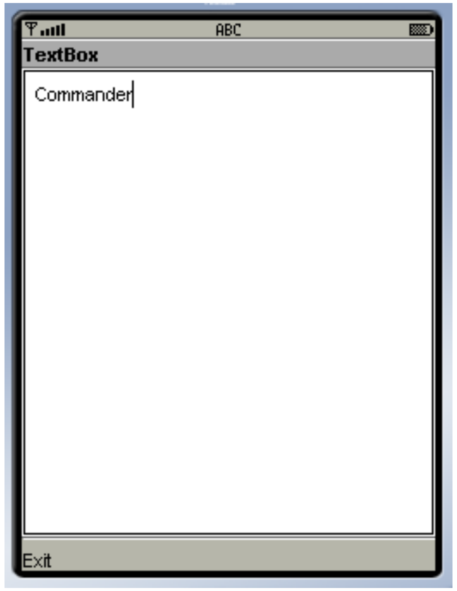
```
public void commandAction(Command c, Displayable s)
```

Event listeners should not perform lengthy processing inside the event-handling thread. The system uses its own thread to call `commandAction()` in response to user input.
If your implementation of `commandAction()` does any heavy thinking, it will tie up the system’s event handling thread.
If you have anything complicated to do use your own thread.

Command example

code

```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;

public class Commander extends MIDlet {
  public void startApp() {
    Displayable d = new TextBox("TextBox", "Commander", 20, TextField.ANY);

    Command c = new Command("Exit", Command.EXIT, 0);
    d.addCommand(c);
    d.setCommandListener(new CommandListener() {
      public void commandAction(Command c, Displayable s) {
        notifyDestroyed();
      }
    });

    Display.getDisplay(this).setCurrent(d);
  }

  public void pauseApp() {}

  public void destroyApp(boolean unconditional) {}
}
```

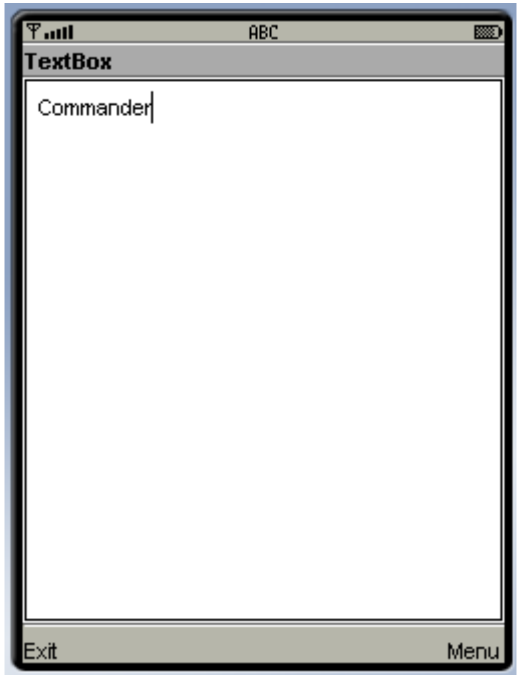


Command example - II

- If you add another command to this MIDlet, it will be mapped to the other soft button
- If you continue adding commands, the ones that don’t fit on the screen will be put into an off-screen menu
- If you press the soft button for Menu you’ll see the remainder of the commands

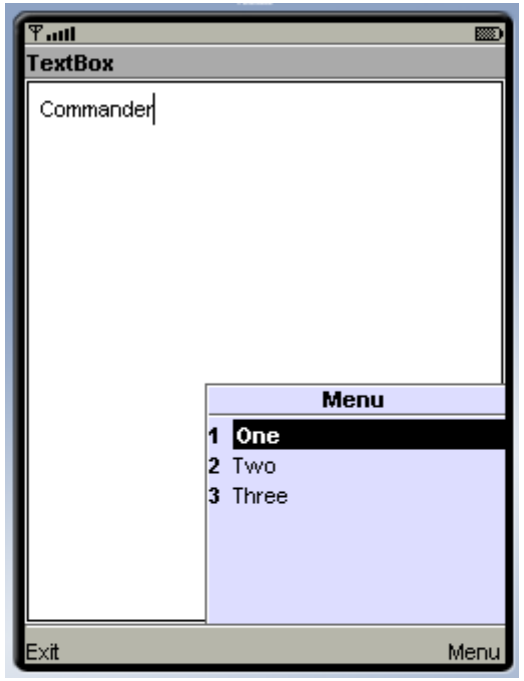
In this case, Exit command has higher priority (lower number) than the other commands, which ensures that it appears directly on screen

The other commands, are relegated to the command menu.



TextBox

Commander

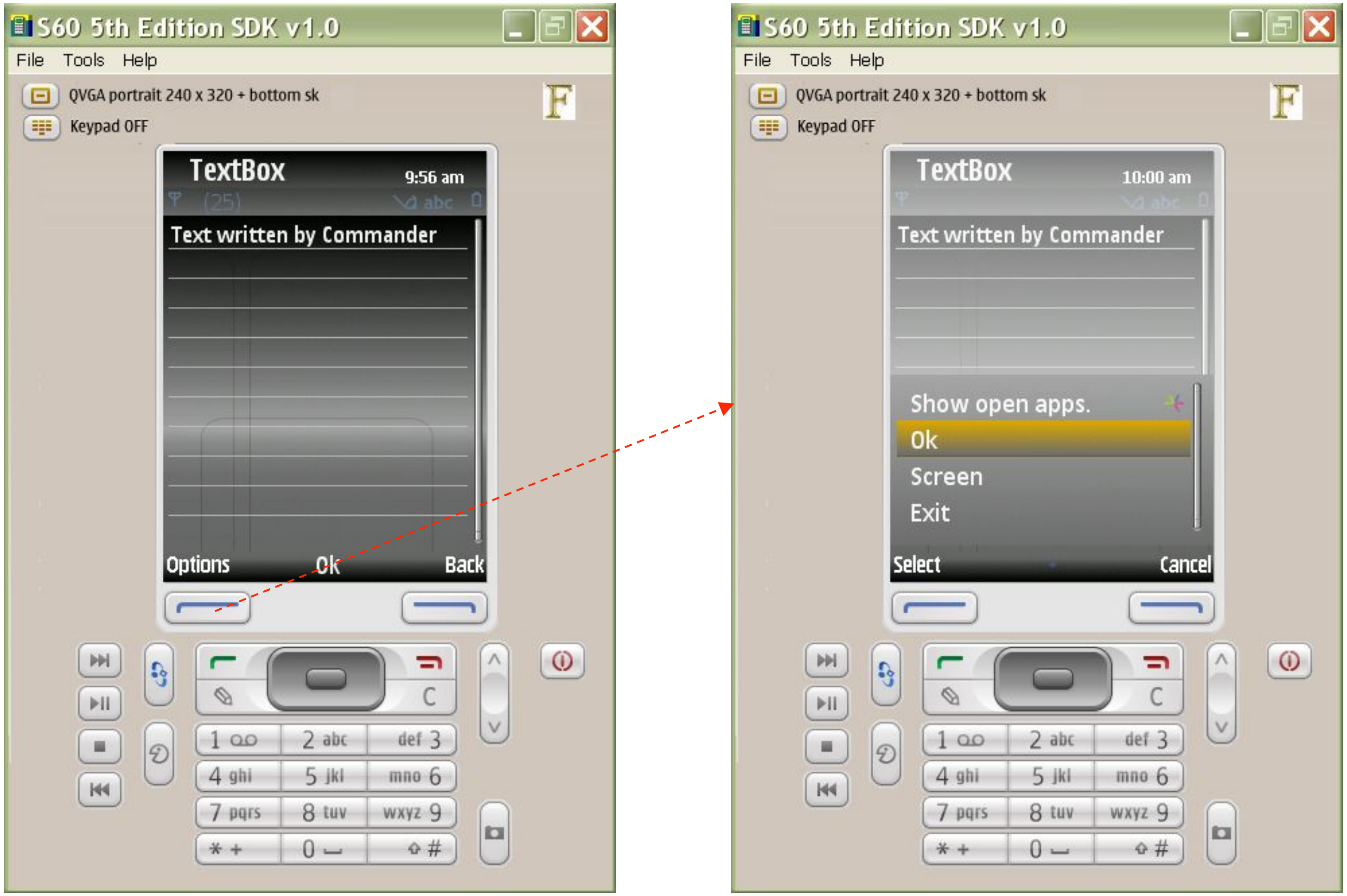


TextBox

Commander

Command example III - Nokia S60

```
Command c1 = new Command("Exit", Command.EXIT, 0);
Command c2 = new Command("Screen", Command.SCREEN, 0);
Command c3 = new Command("Ok", Command.OK, 0);
Command c4 = new Command("Back", Command.BACK, 0);
```

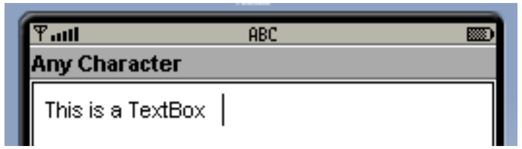


Text written by Commander

Text written by Commander

Screens

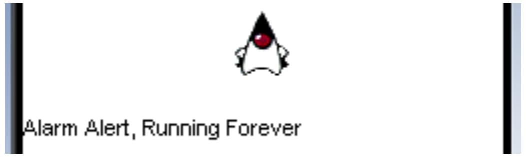
- `Screen` is the base class for all classes that represent generalized user interfaces
- The class `Screen` has no methods of it’s own, but inherits all from `Displayable`
- In the coming slides we’ll explore each of Screen’s child classes:
  - Text Box



Any Character

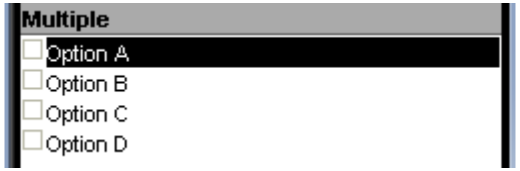
This is a TextBox

  - Alert



Alarm Alert, Running Forever

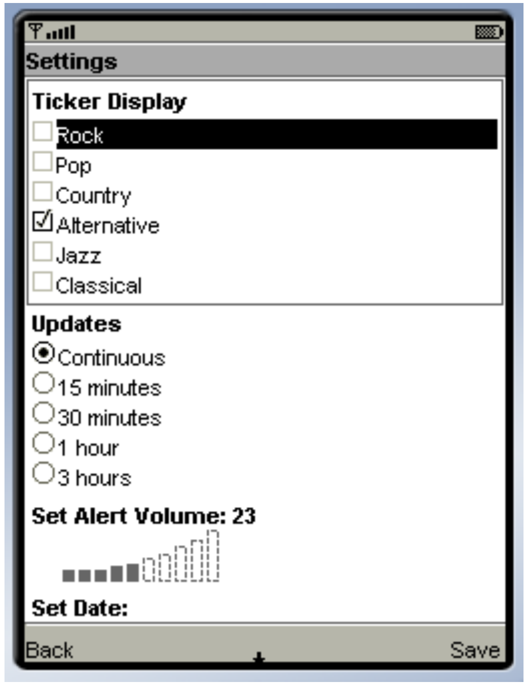
  - List



Multiple

☐ Option A
☐ Option B
☐ Option C
☐ Option D

  - Form →



Settings

Ticker Display

☐ Rock
☐ Pop
☐ Country
☑ Alternative
☐ Jazz
☐ Classical

Updates

● Continuous
○ 15 minutes
○ 30 minutes
○ 1 hour
○ 3 hours

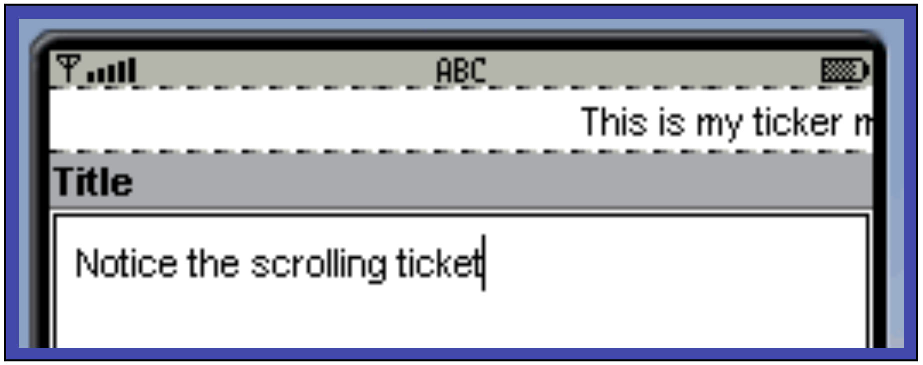
Set Alert Volume: 23

Set Date:

Title and Tickers

- All `Displayable` have a **title** and an optional **ticker**
- The **title** usually is set when you create a new screen object, e.g.,: `TextBox(String title, String text, int maxSize, int constraints)`
- but it can be access also using:
  - `Public void setTitle(String newTitle)`
  - `Public void getTitle()`
- A ticker is simply a text that scrolls across the top of Displayable
- To add a ticker to a displayable `d`:

```
Ticker ticker = new Ticker(“This is my ticker message!”);
d.setTicker(ticker);
```



Notice the scrolling ticket

Textbox

- `TextBox` allows the user to enter a string
- Keep in mind to minimize text input in MIDlet
- A TextBox is created by specifying four parameters:
  - `public TextBox(String title, String text, int maxSize, int constraints)`
- **title**: screen title
- **text**: the initial text
- **maxSize**: maximum size of textbox
- **constraints**: ANY, NUMERIC, DECIMAL, PHONENUMBER, EMAILADDR, URL
  combined with
  PASSWORD, UNEDITABLE, SENTENCE, NON_PREDICTIVE, INITIAL_CAPS_WORD, INITIAL_CAPS_SENTENCE

Textbox II

- If you don’t want the `TextBox` to perform any validation use ANY for the constraints parameter in the constructor
- The flags may be combined with any of the other constraints using the OR operator
- For example to create a `TextBox` that constrains input to an email address but keeps the entered data hidden, you would do something like this:

```
Displayable d = new TextBox(“Email”, “”, 64,
  TextField.EMAILADDR | TextField.PASSWORD);
```

Using Alerts

- An alert is an informative message shown to the user
- There are two types of alerts
- A **timed** alert is **shown for a certain amount of time**, typically just a few seconds
- A **modal** alert **stays up until the user dismisses it** - modal alerts are useful when you need to offer the user a choice of actions (commands)
- Alerts can have an associated **icon**, like a stop sign or question mark
- May also have a sound, but it depend on the implementation.

Alerts

- The constructors of alerts:

```
public Alert()

public Alert(String title, String alertText,
  Image alertImage, AlertType alertType)
```

- By default Alerts are created using default timeout value
- You could create a simple alert with the following

```
Alert alert = new Alert(“Alert”, “Pay attention!”, null,
null);
```

- To use a timeout of five seconds:

```
alert.setTimeout(5000);
```

- If you want a modal alert use the value FOREVER

```
alert.setTimeout(Alert.FOREVER);
```

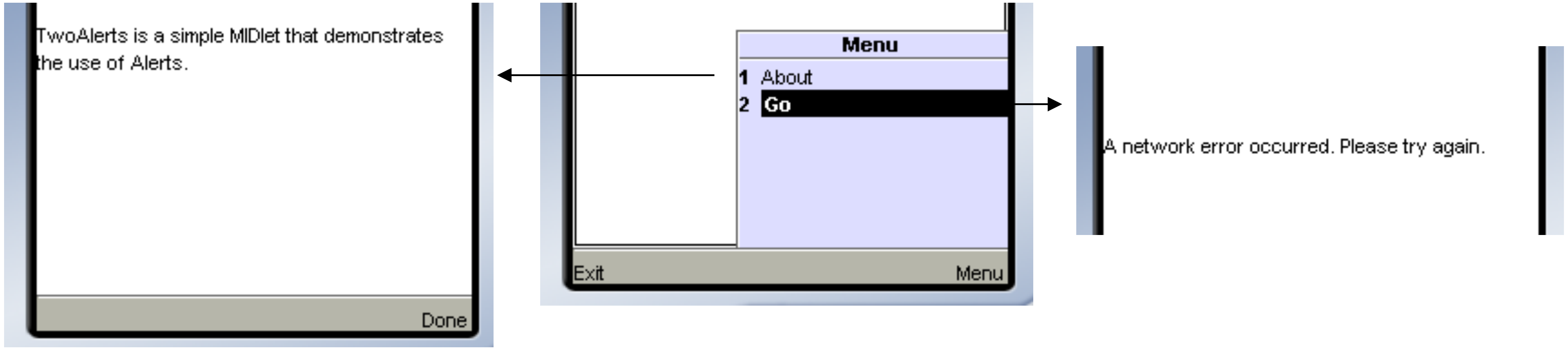
- The MIDP implementation will automatically supply a way to dismiss a modal alert (WTK implementation provides a done command mapped to a soft button)

Alerts

- By default, Alerts automatically advances to the *next screen* when the Alert is dismissed or times out
- You can specify the *next screen* by passing it and the Alert to the **two-arguments** `setCurrent()` menthod in `Display`
- If you call the **one argument** `setCurrent()` method, **the previous screen is restored** when the Alert is dismissed
- A type serve as hints to the underlying MIDP implementation (different rendering)
- The `AlertType` class provides five types:
  - ALARM, CONFIRMATION, ERROR, INFO, WARNING.

Alerts Example

- In the example we'll see both type of alerts
- It displays a `TextBox` when the MIDlet begins
- Two commands Go and About for showing alerts
- **Go**: shows **timed** alert with a network error
- **About**: displays a **modal** alert with some info
- **Exit**: provides a way to exit the MIDlet



TwoAlerts is a simple MIDlet that demonstrates the use of Alerts.

A network error occurred. Please try again.

Two Alerts (I)

code

```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;

public class TwoAlerts
    extends MIDlet
    implements CommandListener {
  private Display mDisplay;

  private TextBox mTextBox;
  private Alert mTimedAlert;
  private Alert mModalAlert;

  private Command mAboutCommand, mGoCommand, mExitCommand;

  public TwoAlerts() {
    mAboutCommand = new Command("About", Command.SCREEN, 1);
    mGoCommand = new Command("Go", Command.SCREEN, 1);
    mExitCommand = new Command("Exit", Command.EXIT, 2);
```

Two Alerts (II)

```
 mTextBox = new TextBox("TwoAlerts", "TwoAlerts is ..",
32, TextField.ANY);
  mTextBox.addCommand(mAboutCommand);
  mTextBox.addCommand(mGoCommand);
  mTextBox.addCommand(mExitCommand);
  mTextBox.setCommandListener(this);

  mTimedAlert = new Alert("Network error",
      "A network error occurred. Please try again.",
      null,
      AlertType.ERROR);
  mModalAlert = new Alert("About TwoAlerts",
      "TwoAlerts demonstrates the use of Alerts.",
      null,
      AlertType.INFO);
  mModalAlert.setTimeout(Alert.FOREVER);
}
```

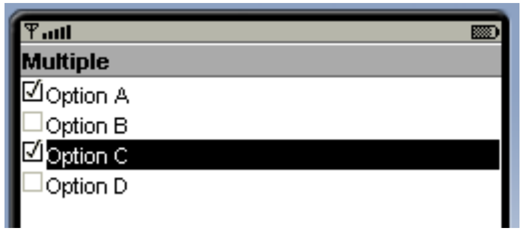
Two Alerts (III)

```
 public void startApp() {
    mDisplay = Display.getDisplay(this);

    mDisplay.setCurrent(mTextBox);
  }

  public void pauseApp() {
  }

  public void destroyApp(boolean unconditional) {}

  public void commandAction(Command c, Displayable s) {
    if (c == mAboutCommand)
      mDisplay.setCurrent(mModalAlert);
    else if (c == mGoCommand)
      mDisplay.setCurrent(mTimedAlert, mTextBox);
    else if (c == mExitCommand)
      notifyDestroyed();
  }
}
```

Using List

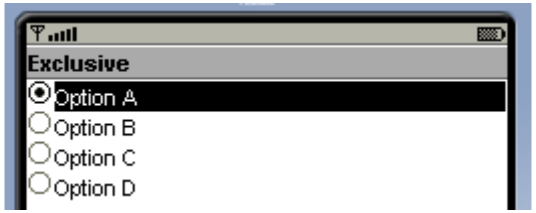
- A list allows the user to select items from a list of choices

- There are three different types of list
  - MULTIPLE: like check box, where multiple elements may be selected simultaneously
  - EXCLUSIVE: like radio buttons, where you can select only one element
  - IMPLICIT: it combines the steps of selection and confirmation - acts just like a menu.



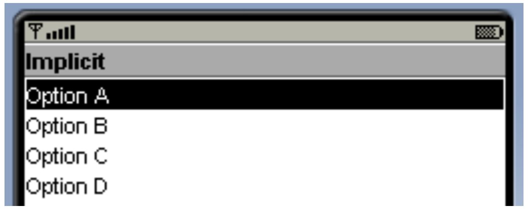
Multiple

☑ Option A
□ Option B
☑ Option C
□ Option D



Exclusive

● Option A
○ Option B
○ Option C
○ Option D



Implicit

Option A
Option B
Option C
Option D

Creating List

- To create a `List`, specify a title and a list type
- `List` constructors:

```
Public List(String title, int type)

Public List(String title, int type,
     String[] stringElements, image[] imageElements)
```

- *stringElements* parameter cannot be null, but can be empty
- *imageElements* may contain null array elements
- If both string and image are defined an element will display using the image and the string, otherwise it will display only a string
- If the *List* exceed the dimension of the screen, *List* automatically handle scrolling up and down to show the full contents of the *List.*

Editing a List

- The first element is at index 0
- You can use the following methods (of `List`) to add, remove or examine an element:

`public int append(String stringPart, Image imagePart)`	Add at the end
`public void insert(int elementNum, String stringPart, Image imagePart)`	Insert at index
`public void set(int elementNum, String stringPart, Image imagePart)`	Set at index
`public String getString(int elementNum)`	Retrieve the String
`public Image getImage(int elementNum)`	Retrieve the Image
`public void delete(int elementNum)`	Remove an Element
`public void deleteAll()`	Remove all Elements
`public int size()`	Number of Elements

Other methods of `List`

- `setFitPolicy(int fitPolicy)` → how it should be handled when wider than the screen
- Fit policies:
  - TEXT_WRAP_ON: long elements wrapped on multi lines
  - TEXT_WRAP_OFF: long elements will be truncated at the edge of the screen
  - TEXT_WRAP_DEFAULT: the implementation should use the default policy
- `setFont(int elementNum, Font font)` → you can specify what type of font to be used for a specific *List* element
- The current Font for an element can be retrieved by calling `getFont(int elementNum)`

Working with List Selections

- You can check what element is selected on a *List* using the following methods:

```
public boolean isSelected(int index)
```

- For EXCLUSIVE and IMPLICIT lists, the index of the single selected element is returned from the following method:

```
public int getSelectedIndex()
```

- Lists allows to set or get the whole selection state

```
public int getSelectedFlags( boolean[]
    selectedArray_return)
public void setSelectedFlags(boolean[]
    selectedArray)
```

- The supplied arrays must have the same number of Lists elements.

Event Handling for IMPLICIT List

- When a user makes a selection the `commandAction()` of the list’s `CommandListener` is invoked
- A special value is passed to `commandAction()` as the `Command` parameter
  - `public static final Command SELECT_COMMAND`
- For example, you can test the source of command events like this:

```
Public void commandAction(Command c, Displayable s) {
   if (c == nextCommand)
        // ...
   else if (c == List.SELECT_COMMAND)
        // ...
}
```

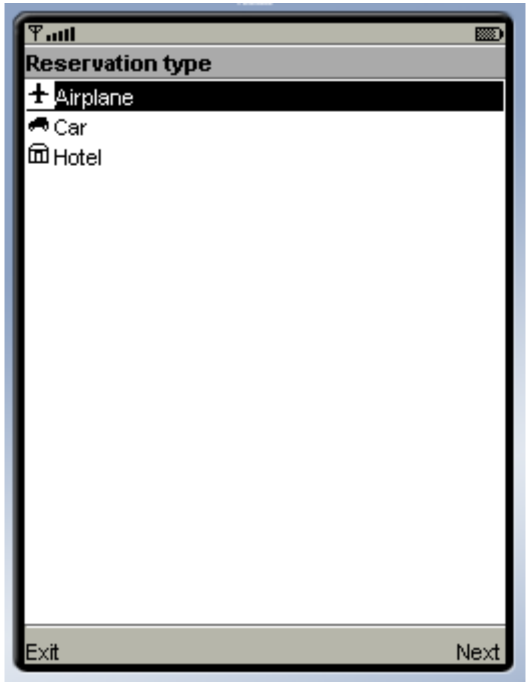
Example using List

- Following a simple MIDlet that could be part of travel reservation application
- The user chooses what type of reservation to make
- This example uses an IMPLICIT list, which is substantially a menu
- The list has images: `airplane.png, car.png, hotel.png`
- The application includes a *Next* command and an *Exit* command both added to the *List*
- When you select an item, the item description is retrieved from the List and shown in an Alert.

TravelList code example (I)

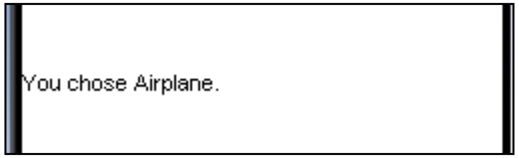
```
import java.io.*;
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;

public class TravelList
    extends MIDlet
    implements CommandListener {
  private List mList;
  private Command mExitCommand, mNextCommand;

  public TravelList() {
    String[] stringElements = { "Airplane", "Car", "Hotel" };
    Image[] imageElements = { loadImage("/airplane.png"),
        loadImage("/car.png"), loadImage("/hotel.png") };
    mList = new List("Reservation type", List.IMPLICIT,
        stringElements, imageElements);
    mNextCommand = new Command("Next", Command.SCREEN, 0);
    mExitCommand = new Command("Exit", Command.EXIT, 0);
    mList.addCommand(mNextCommand);
    mList.addCommand(mExitCommand);
    mList.setCommandListener(this);
  }

  public void startApp() {
    Display.getDisplay(this).setCurrent(mList);
  }
```

[code]()

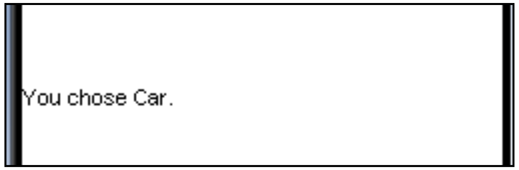

Reservation type

TravelList code example (II)

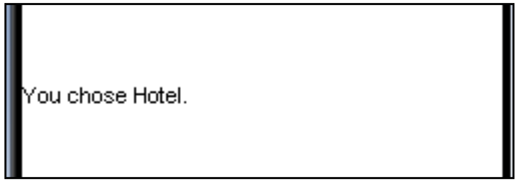
```
 public void commandAction(Command c, Displayable s) {
    if (c == mNextCommand || c == List.SELECT_COMMAND) {
      int index = mList.getSelectedIndex();
      Alert alert = new Alert("Your selection",
          "You chose " + mList.getString(index) + ".",
          null, AlertType.INFO);
      Display.getDisplay(this).setCurrent(alert, mList);
    }
    else if (c == mExitCommand)
      notifyDestroyed();
  }

  public void pauseApp() {}

  public void destroyApp(boolean unconditional) {}

  private Image loadImage(String name) {
    Image image = null;
    try {
      image = Image.createImage(name);
    }
    catch (IOException ioe) {
      System.out.println(ioe);
    }

    return image;
  }
}
```



You chose Airplane.



You chose Car.



You chose Hotel.

About Images

- Instances of `Image` class represent images in MIDP
- Implementation is able to **load** file in PNG format (not all varieties) - as we have seen before
- Image has no constructor, you have to use `createImage()` factory methods for obtaining Image instances
  - `createImage(String name)`

    From a file packed in jar
  - `createImage(byte[] imagedata, int imageoffset, int imagelength)`

    From a buffer
  - `createImage(InputStream stream)`

    From a stream

About Images (II)

- Images may be *mutable* (can be modified) or *immutable* (cannot be modified)
- A list accepts only **immutable** images
- You can create an immutable image from a mutable one using the following method:
  - `public static Image createImage(Image image)`
- You can create an Image from a portion of another Image:
  - `public static Image createImage(Image image, int x, int y, int width, int height, int transform)`
- You could ask the `Display` what is the best image size for a particular usage (LIST_ELEMENT, ALERT, or CHOICE_GROUP_ELEMENT):
  - `public int getBestImageHeight(int imageType)`
  - `public int getBestImageHeight(int imageType)`

Advanced Interfaces with Forms

- Form is a screen that **can include a collection of** user-interface controls called **`Item`**
- Forms that don't fit in the screen will automatically be made scrollable if needed
- One way to create a `Form` is by calling

  ```
  public Form(String title)
  ```

- *Or* if you have all *items* defined you can pass them by calling:

  ```
  public Form(String title, Item[] items)
  ```

- Form displays command and fires events (inherited from `Displayable`)
- To set the focus on a particular item when a Form is shown use the method `setCurrentItem()`.

Managing Items

- Items may be added and removed
- The order of items is important
- The following methods are available to manage items

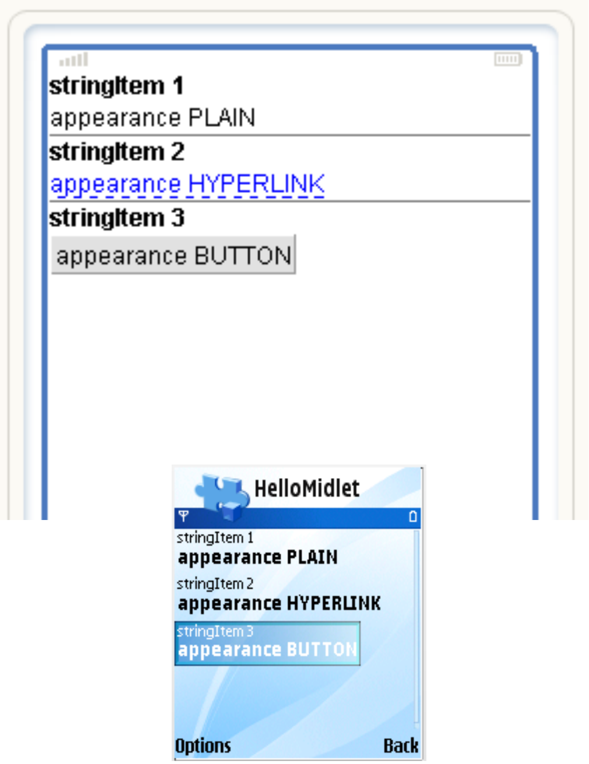
```
public int append(Item item)
public int append(String str)
```
create and append a
```
  StringItem
public int append(Image image)
public void set(int index, Item item)
public void insert(int index, Item item)
public void delete(int index)
public void deleteAll()
public int size() //number of items
public Item get(int index)
```

Items - StringItem

- `StringItem` represents a simple text label (it is not editable by the user)
- There are two types of constructors:
  - `StringItem(String label, String text)`
  - `StringItem(String label, String text, int appearanceMode)`
- You can use `null` for `StringItem label` or `text` to indicate that it should not be shown
- Use `getText()` and `setText()` to access the `text` value
- Use `getLabel()` and `setLabel()` to access the `label` value.

Items - StringItem

- The appearance of both `StringItem` and `ImageItem` can be controlled using *appearance mode*
- The appearance mode allows the item to look like an URL link or a button - three appearance modes:
  - `PLAIN`: normal state
  - `HYPERLINK`: shows the item as URL
  - `BUTTON`: shows the item as a button
- Remember, the appearance may look different on different device like almost everything else in the `javax.microedition.lcdui` package
- *In the SDK there is no difference, unless the button is associated with a command.*



stringItem 1
appearance PLAIN
stringItem 2
appearance HYPERLINK
stringItem 3
appearance BUTTON

HelloMidlet
stringItem 1
appearance PLAIN
stringItem 2
appearance HYPERLINK
stringItem 3
appearance BUTTON

Items and commands

- All items have a string label (may be not visible)
- Items can have **commands** - shown together with the commands in the form
- You can manage commands on an item using `addCommand()` and `removeCommand()`
- Note that the command type should be **ITEM**
  - `new Command("Exit from Item", Command.ITEM, 0);`
- So if the item has appearance mode `BUTTON` you can fire commands with buttons - DO NOT DO THAT - avoid buttons for firing commands.

Item Commands

```
public class MyHello extends MIDlet  {
  private Form mform;
  private StringItem mItem;

  public MyHello () {
   mform = new Form("Hello Form");
   mItem = new StringItem("hello: \n", "Hello my students!");
   mform.append(mItem);

   mItem.addCommand(new Command("Exit from Item", Command.ITEM, 0));
   mItem.setItemCommandListener(new ItemCommandListener() {
     public void commandAction(Command command, Item item) {
       notifyDestroyed();
     }
   });
  }

  public void startApp() {
    Display.getDisplay(this).setCurrent(mform);
  }

  public void pauseApp() {}

  public void destroyApp(boolean unconditional) {}
}
```

code

The command "Exit from Item" will be available if the focus is on the item.

Item size

- Item have a **minimum size** and **preferred size** that can be use to control how large an item appears in a form
- Minimum size is computed by implementation and is not possible to change it serves in deciding how to layout a form
- The preferred size can be computed by implementation or specified by you
- If you specify a size that dimension will be used during layout (`setPreferredSize(w,h)`)

Item Layout

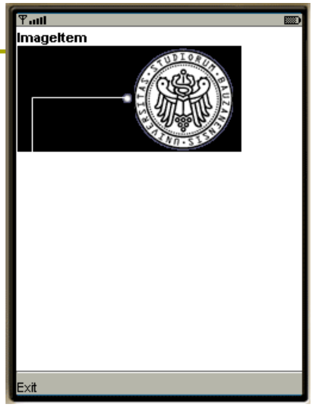
- Item has methods - `getLayout()` and `setLayout()` - related to **layout control -** *There is no Form layout*
- `Form` attempts to lay out items left-to-right in rows, stacking rows top-to-bottom
- Usually the layout value is a combination of `LAYOUT_2` (MIDP 2.0 rules) with a horizontal value and vertical value
- Horizontal Values: `LAYOUT_LEFT,LAYOUT_RIGHT, LAYOUT_CENTER`
- Vertical Values: `LAYOUT_TOP, LAYOUT_BOTTOM, LAYOUT_VCENTER`
- Request a new-line after/before the item: `LAYOUT_NEWLINE_AFTER, LAYOUT_NEWLINE_BEFORE`
- Example

```
int prefLayout = Item.LAYOUT_2 | Item.LAYOUT_LEFT
    | Item.LAYOUT_NEWLINE_AFTER;
mVideoItem.setLayout(prefLayout);
```

Items – TextField

- `TextField` represent an editable string (like `TextBox`)
- **`TextField`**`(String label, String text, int maxSize, int constraints)`
- Ex: `mform.append(new TextField("Phone", "+390471555888", 50, TextField.PHONENUMBER));`
- `TextFields` can limit input using the following
  - `ANY` any type of input
  - `NUMERIC` restrict input to numbers
  - `DECIMAL` allows numbers with fractional parts
  - `PHONENUMBER` requires a telephone number
  - `EMAILADDR` input must be an e-mail address
  - `URL` input must be an URL
- The input constraints are the same as `TextBox` and can be combined with the other constraints using the `OR` operator (see `TextBox` constants)
- For an initial empty `TextField` pass `null` for the `text` parameter.

Items - ImageItem



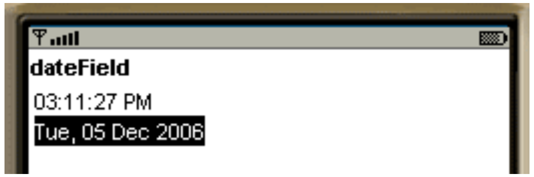
- Forms can contain images, which are represented by an instance of `ImageItem`
- To create an `ImageItem` just supply the `Image` to be displayed, the *label*, *layout* and *alternate text*
- Example:

```
waitImage = Image.createImage("/wait.png");
mColorSquare = new ImageItem(“label”, waitImage,
     ImageItem.LAYOUT_DEFAULT, ”alt text");
```

- Layout is controlled by the constants:
  - `LAYOUT_DEFAULT, LAYOUT_CENTER, LAYOUT_LEFT, LAYOUT_RIGHT`
  - `LAYOUT_NEWLINE_BEFORE, LAYOUT_NEWLINE_AFTER`
- `ImageItem` support *appearance modes* just like `StringItem` (using the appropriate constructor).

Items - DateField

- `DateField` is an useful mechanism by which users can enter dates, times or both
- To create a `DateField` specify a label and a mode:
  - `DateField.DATE` display an editable date
  - `DateField.TIME` displays an editable time
  - `DateField.DATE_TIME` displays both date and time



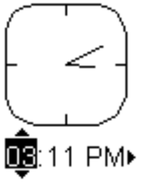
dateField

03:11:27 PM

Tue, 05 Dec 2006

- There are two constructor the first uses default `TimeZone` using the second you have to specify `TimeZone`

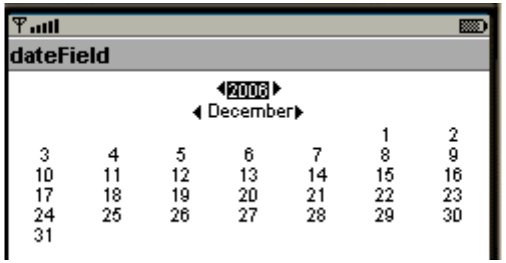
```
DateField(String label, int mode)
DateField(String label, int mode, TimeZone timeZone)
```



03:11 PM

- To get or set the `Date` represented by the `DateField` use the Following methods:

```
Public Date getDate()
Public void setDate(Date date)
```



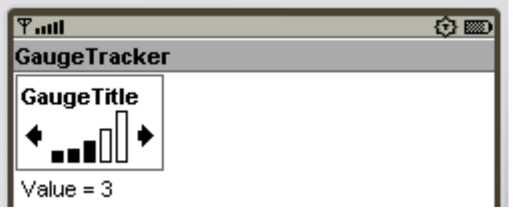
dateField

2006

December

					1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30
31						

Items - Gauge


GaugeTracker

GaugeTitle

Value = 3

- `Gauge` represents an integer value
- The value of a `Gauge` instance can be retrieved or modified with the `getValue()` and `setValue()` - this value runs from 0 to a maximum value
- Maximum value can be retrieved and modified with the `getMaxValue()` and `setMaxValue()`
- In an **interactive** *gauge* the **user can modify** the value
- To create a `Gauge` use the following constructor

```
public Gauge(String label, boolean
interactive,
int maxValue, int initialValue)
```

Non Interactive gauges

- There are **three varieties** of **non interactive** gauges than can be useful as progress indicators:
- With **definite maximum value** the application can increase and decrease its value
- With **indefinite maximum value**
  - **Incremental:** your application forces advance one step or set to idle
    - `setValue(Gauge.INCREMENTAL_UPDATING)` = advances by one step
    - `setValue(Gauge.INCREMENTAL_IDLE)` = changes color to show that it is idle
  - **Continuous:** it advances continuously unless is idle
    - `setValue(Gauge.CONTINUOUS_RUNNING)` = the gauge is animated
    - `setValue(Gauge.CONTINUOS_IDLE)` = changes color to show that it is idle

GaugeMIDlet – source (I)

code

```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;

public class GaugeMIDlet
    extends MIDlet
    implements CommandListener {
  private Display mDisplay;

  private Form mGaugeForm;
  private Command mUpdateCommand, mIdleCommand;

  private Gauge mInteractiveGauge;
  private Gauge mIncrementalGauge;
  private Gauge mContinuousGauge;
```

GaugeMIDlet – source (II)

```
public GaugeMIDlet() {
    mGaugeForm = new Form("Gauges");
    mInteractiveGauge = new Gauge("Interactive", true, 5, 2);
    mInteractiveGauge.setLayout(Item.LAYOUT_2);
    mGaugeForm.append(mInteractiveGauge);
    mContinuousGauge = new Gauge("Non-Interactive continuous", false,
        Gauge.INDEFINITE, Gauge.CONTINUOUS_RUNNING);
    mContinuousGauge.setLayout(Item.LAYOUT_2);
    mGaugeForm.append(mContinuousGauge);
    mIncrementalGauge = new Gauge("Non-Interactive incremental", false,
        Gauge.INDEFINITE, Gauge.INCREMENTAL_UPDATING);
    mIncrementalGauge.setLayout(Item.LAYOUT_2);
    mGaugeForm.append(mIncrementalGauge);

    mUpdateCommand = new Command("Update", Command.SCREEN, 0);
    mIdleCommand = new Command("Idle", Command.SCREEN, 0);
    Command exitCommand = new Command("Exit", Command.EXIT, 0);
    mGaugeForm.addCommand(mUpdateCommand);
    mGaugeForm.addCommand(mIdleCommand);
    mGaugeForm.addCommand(exitCommand);
    mGaugeForm.setCommandListener(this);
  }
```

GaugeMIDlet – source (III)

```
public void startApp() {
    if (mDisplay == null) mDisplay = Display.getDisplay(this);
    mDisplay.setCurrent(mGaugeForm);
  }

  public void pauseApp() {}

  public void destroyApp(boolean unconditional) {}

  public void commandAction(Command c, Displayable s) {
    if (c.getCommandType() == Command.EXIT)
      notifyDestroyed();
    else if (c == mUpdateCommand) {
      mContinuousGauge.setValue(Gauge.CONTINUOUS_RUNNING);
      mIncrementalGauge.setValue(Gauge.INCREMENTAL_UPDATING);
    }
    else if (c == mIdleCommand) {
      mContinuousGauge.setValue(Gauge.CONTINUOUS_IDLE);
      mIncrementalGauge.setValue(Gauge.INCREMENTAL_IDLE);
    }
  }
}
```

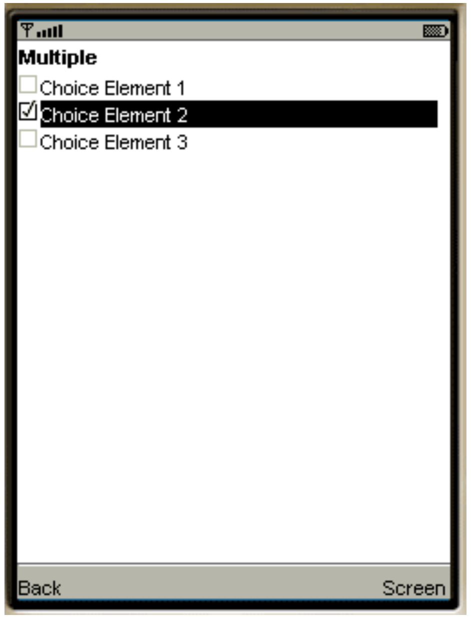
Items - ChoiceGroup

- `ChoiceGroup` offers a list of choices
- Is similar to `List` and implements the same `Choice` interface - the constructors are:

  ```
  ChoiceGroup(String label, int choiceType)
  ```

  ```
  ChoiceGroup(String label, int choiceType, String[]
      stringElements, Image[] imageElements)
  ```

- `choiceType` can be `EXCLUSIVE`, `MULTIPLE` or `POPUP` (there is no `IMPLICT` as in `List`)

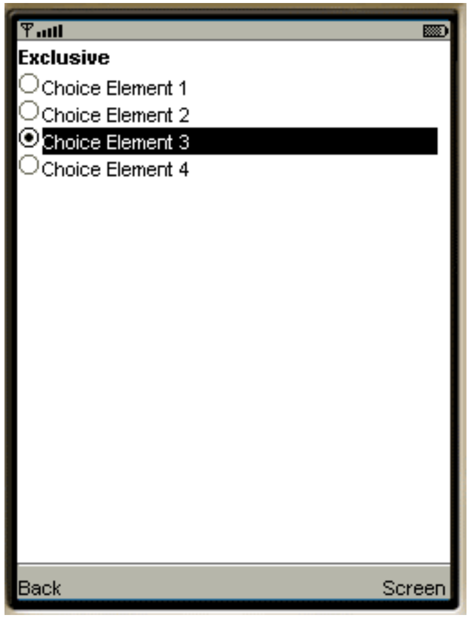


Multiple

☐ Choice Element 1

☑ Choice Element 2

☐ Choice Element 3



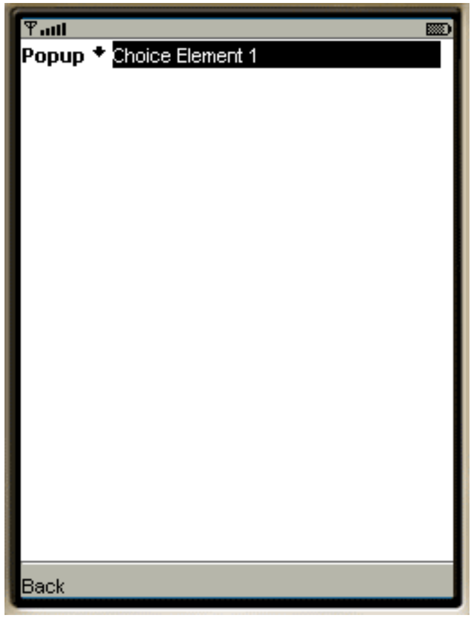
Exclusive

○ Choice Element 1

○ Choice Element 2

● Choice Element 3

○ Choice Element 4



Popup ▾ Choice Element 1

Responding to Item Changes

- Most items in a `Form` fire events when the user changes them
- The application can listen for these events by registering an `ItemStateListener` with the `Form` using the following method:

```
public void setItemStateListener(ItemStateListener
    iListener)
```

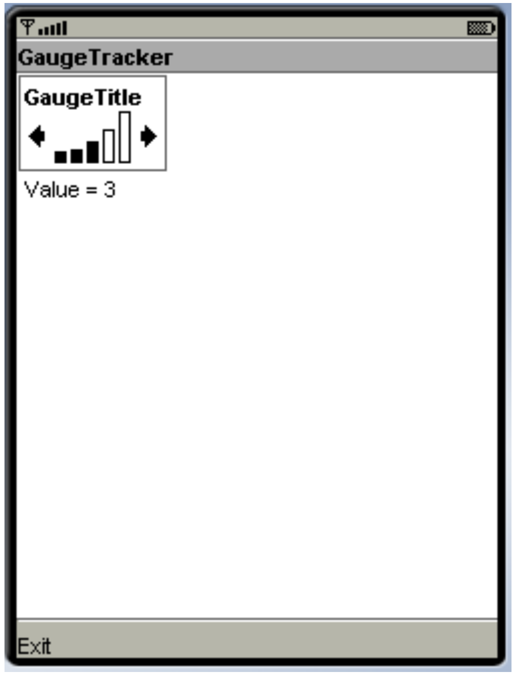
- `ItemStateListener` is an interface with a single method:

```
    public void itemStateChanged(Item item)
```

- That method is **called every time an item** (whatever) in a `Form` **is changed**
- In the following example there is a `Form` with two items, an interactive `Gauge` and a `StringItem`
- As you adjust the `Gauge`, its value is reflected in the `StringItem` using the `ItemStateListener` mechanism.

GaugeTracker example (I)

```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;

public class GaugeTracker
    extends MIDlet
    implements ItemStateListener, CommandListener {
  private Gauge mGauge;
  private StringItem mStringItem;

  public GaugeTracker() {
    int initialValue = 3;
    mGauge = new Gauge("GaugeTitle", true, 5, initialValue);
    mStringItem = new StringItem(null, "[value]");
    itemStateChanged(mGauge);
  }
  // the midlet implements ItemStateListener
  public void itemStateChanged(Item item) {
    if (item == mGauge)
      mStringItem.setText("Value = " + mGauge.getValue());
  }
 // the midlet implements CommandListener
  public void commandAction(Command c, Displayable s) {
    if (c.getCommandType() == Command.EXIT)
      notifyDestroyed();
  }
```



Value = 3

code

GaugeTracker example (II)

```
 public void startApp() {
    Form form = new Form("GaugeTracker");
    form.addCommand(new Command("Exit", Command.EXIT, 0));
    form.setCommandListener(this);
    // Now add the selected items.
    form.append(mGauge);
    form.append(mStringItem);
    form.setItemStateListener(this);

    Display.getDisplay(this).setCurrent(form);
  }

  public void pauseApp() {}

  public void destroyApp(boolean unconditional) {}
}
```

ItemStateChanged

- `public void` **`itemStateChanged`**`(Item item)`
- Called when internal state of an Item has been changed by the user
- This happens when the user:
  - changes the set of selected values in a ChoiceGroup;
  - adjusts the value of an interactive Gauge;
  - enters or modifies the value in a TextField;
  - enters a new date or time in a DateField;
  - `Item.notifyStateChanged()` was called on an Item – force a notification of a state change.

Summary on GUI and MIDlet

- `Displayable` has two sub-types: `Canvas` and `Screen`
- `Screen` has four sub-types:
  - `TextBox`: displaying and editing text
  - `List`: selecting one option among a list
  - `Alert`: pop up a message to the user
  - `Form`: aggregates simple GUI `Item`
- `Item` has eight sub-types:
  - `StringItem:` display text -has three appearances: `PLAIN, HYPERLINK,BUTTON`
  - `TextField` is like a `TextBox`
  - `ImageItem` to display images
  - `DateField` to input a date
  - `Gauge` to show and input an integer in a range
  - `ChoiceGroup` similar to `List`
  - `Spacer` to set some space between items
  - `CustomItem` to do whatever you like!

Summary on Commands

- A `Command c` can be added to any `Displayable Screen d`
  - `d.addCommand(c);`
- The `Command` is managed by registering a `CommandListener cl` to the `Displayable`
  - `d.setCommandListener(cl);`
- The `CommandListener` must implement the method
  - `commandAction(Command c, Displayable s)`
- A `Command c` can be added to an `Item i` in a Form `(i.addCommand(c);)`
- It is managed by registering an `ItemCommandListener icl` to the `Item i:`
- `i.setItemCommandListener(icl);`
- An `ItemCommandListener` must implement:
  - `commandAction(Command command, Item item)`

Summary on Events on Items

- When an `Item` is changed, the method
  - `public void itemStateChanged(Item item)`
- of the `ItemStateListern` object registered to the item is called
- To register a `ItemStateListener` with a form, use the method:
  - `public void setItemStateListener (ItemStateListener   iListener)`