Second Session

July 8, 2006

Java 2 Micro Edition Details

MIDlet application model

- Derived from internet Applet application model
  - Simplifications for mobile environment
    - Many libraries removed or simplified to fit in the memory of a mobile device
    - Restrictions introduced in the previous lecture
  - Modifications for mobile environment
    - Management of applications
  - Restrictions of mobile systems apply when using the model
    - Memory usage
    - Performance

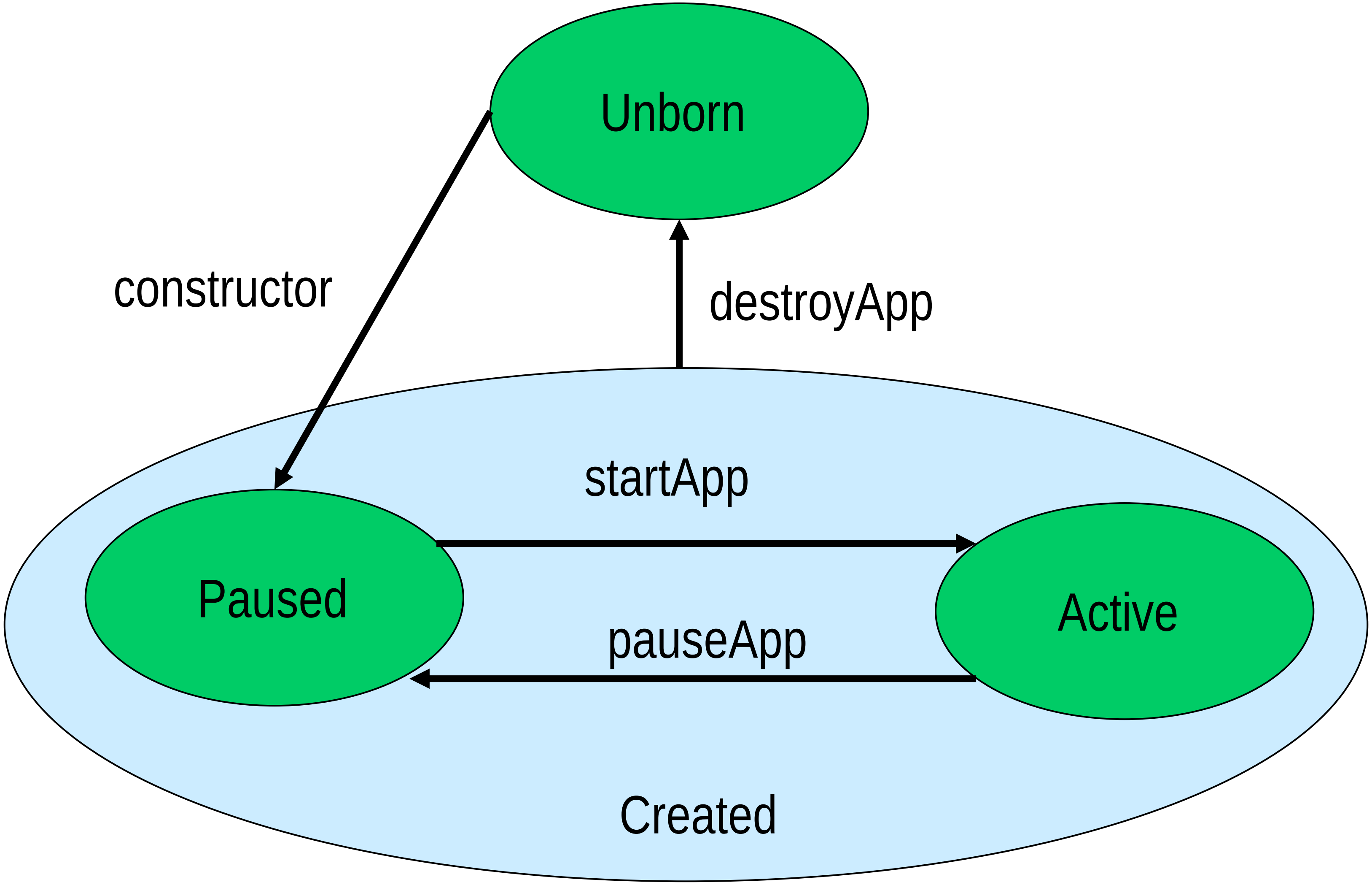
Typical application management operations

- Retrieval --- Retrieve a MIDlet suite
  - medium-identification, negotiation, retrieval
- Installation --- Install MIDlet suite on a device
  - verification, transformation
- Launch --- Invoke a MIDlet
  - inspection, invocation
- Version management --- Upgrade installed MIDlet suites
  - inspection, version management
- Removal --- Remove an installed MIDlet suite
  - inspection, deletion

MIDlet application model

- Defines the concept of an application in MIDP environment
- `javax.microedition.midlet`
- Every application must be a subclass of class MIDlet
  - `constructor`
  - `startApp()`
  - `pauseApp()`
  - `destroyApp()`
- In many cases, one also implements interface `CommandListener`

MIDlet states



Sample code: HelloMIDlet

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

public class HelloWorld extends MIDlet implements
   CommandListener {
  private Command exitCommand;
  private TextBox tb;
  public HelloWorld() {
    exitCommand = new Command("exit", Command.EXIT, 1);
    tb = new TextBox("HelloWorld", "Hello world!", 15, 0);
    tb.addCommand(exitCommand);
    tb.setCommandListener(this);
  }
  protected void startApp()
   {Display.getDisplay(this).setCurrent(tb);}
  public void commandAction(Command c, Displayable d) {
    if (c== exitCommand) { destroyApp(false);
   notifyDestroyed();}
  }
  protected void destroyApp(boolean u) {}
  protected void pauseApp() {}
}
```

MIDlet suites

- Applications are represented publicly as JAR files
  - Class files
  - Resource files
  - Manifest that describes the JAR contents
- Multiple MIDlets can reside in the same JAR file
  - MIDlet suite
  - Applications can share data only with those applications that are in the same MIDlet suite
  - Therefore MIDlet design is an important architectural decision when aiming at larger systems
- Each MIDlet suite has a short textual descriptor file
  - A device user does not have to download the entire application before knowing if it will actually run in the device
  - JAD file with the contents that are (almost) identical with JAR manifest

JAD/JAR manifest contents

- MIDlet-Name --- MIDlet suite name
- MIDlet-Version
- MIDlet-vendor
- MIDlet-Icon
- MIDlet-Info-URL
- MIDlet-<n> --- name, icon and class per midlet
- MIDlet-Jar-URL
- MIDlet-Jar-Size
- MIDlet-Data-Size
- MicroEdition-profile --- J2ME profile
- MicroEdition-Configuration --- J2ME configuration

Example: JAD/JAR manifest of a simple application

```
MIDlet-1: HelloWorld, /HelloWorld.png,
  HelloWorld
MIDlet-Jar-Size: 2179
MIDlet-Jar-URL: HelloWorld.jar
MIDlet-Name: HelloWorld
MIDlet-Vendor:
MIDlet-Version: 1.0
MicroEdition-Configuration: CLDC-1.0
MicroEdition-Profile: MIDP-1.0
```

MIDlet suite execution environment

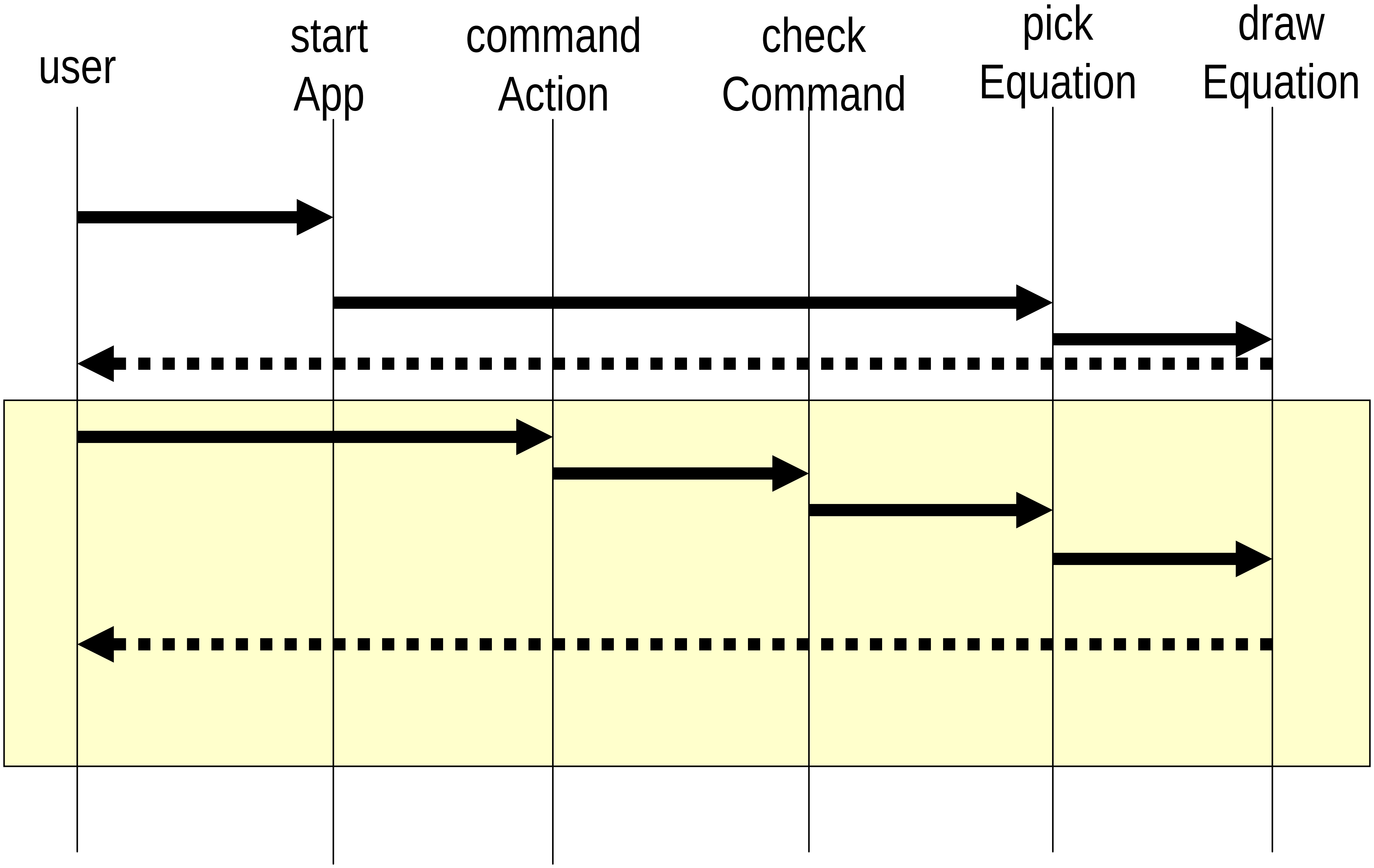
- Classes and native code that implement CLDC and MIDP
  - Only shared name space for all MIDlet suites
- Classes within MIDlet suite’s JAR file
- All non-class files in the MIDlet’s suite’s JAR file
  - Icons, images, JAR manifest
  - Accessible with java.lang.Class.getResourceAsStream
- Contents of Application Descriptor File
  - Accessible with java.lang.Class.getResourceAsStream
- Separate name space for RMS record stores

Application: MathHangman

- Simple MIDP Application
- Takes around 1 hour to build
- Puzzle type
- Next slide shows the process flow/behaviour
- Uses basic MIDP components and J2ME packages

  i.e. javax.*

Behavior


user
start App
command Action
check Command
pick Equation
draw Equation

Imports & instance/variables

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

public class mathHangman extends MIDlet implements
   CommandListener {
  private Command exitCommand, rightCommand, wrongCommand;

  private TextBox tb;  /* Quiz canvas */
  private int nth;     /* Counter for equation and answer pair.
   */
  String Questions[] = /* Questions */
       { "1 + 3 = 4",
         "3 + 12 = 14",
         "12 + 7 = 19",
         "4 + 9 = 12",
         "1 + 2 = 3",
         "2 + 12 = 15",
         "3 + 12 = 15",
         "12 + 2 = 13",
         "10 + 5 = 15",
         "12 + 5 = 16" };
  int Qlen = Questions.length;
```

Constructor and user actions

```
public mathHangman() {
    exitCommand = new Command("exit", Command.EXIT, 1);
    wrongCommand = new Command("WRONG", Command.CANCEL, 2);
    rightCommand = new Command("RIGHT", Command.OK, 3);
}
protected void startApp() { pickEquation(); }

protected void destroyApp(boolean u) {}

protected void pauseApp() {}

 public void commandAction(Command c, Displayable d) {
    if (c == exitCommand) {
      destroyApp(false);
      notifyDestroyed();
    } else ( checkCommand(c == rightCommand);
    } // else
  }
```

Displaying the next question

```
private void pickEquation(){
  displayEquation(Questions[nth]);}

private void displayEquation(String equation) {
  tb = new TextBox(”mathHangman", equation, 15, 0);
  tb.addCommand(wrongCommand);
  tb.addCommand(rightCommand);
  tb.setCommandListener(this);
  Display.getDisplay(this).setCurrent(tb);
}
```

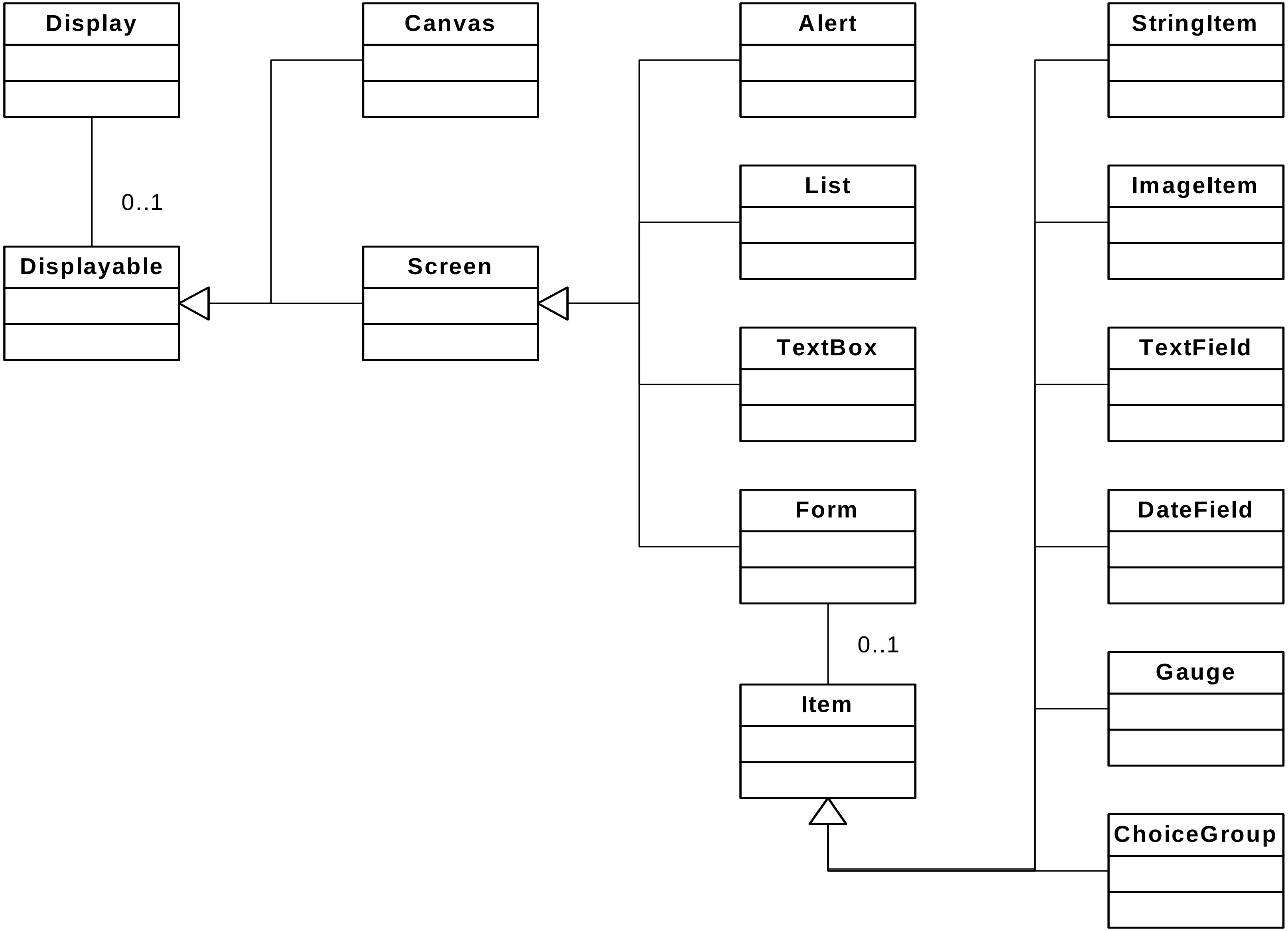
Calculating if the answer is correct

```
  private void checkCommand(boolean b) {
     if(((nth%2) == i) = b) {
       tb = new TextBox("mathHangman",
                        "You failed the challenge", 30, 0);
       tb.addCommand(exitCommand);
       tb.setCommandListener(this);
       Display.getDisplay(this).setCurrent(tb);
     } else {
       nth = nth + 1;
       if (nth == Qlen) {
          tb = new TextBox("mathHangman",
                           "You mastered the challenge", 30, 0);
          tb.addCommand(exitCommand);
          tb.setCommandListener(this);
          Display.getDisplay(this).setCurrent(tb);
        } else {
          pickEquation();
        }
     }
  }
} // class mathHangman
```

Special libraries and interfaces

- UI
  - Different devices
    - Different screen capabilities (e.g. screen, color, ...)
    - Different input mechanisms (keyboard, touch screen, ...)
- Networking interface
  - http focused
  - No link to final implementation-level protocol
- Persistence
  - RMS, record management system
  - Reflects a simple, record-oriented database
    - Records interfaced with byte arrays

UI class hierarchy (MIDP 1.0)



Explaining the hierarchy

- Canvas
  - Low-level interface
  - Application to provide graphics and handle input
    - Control what is drawn
    - Handle primitive events (e.g. key presses)
    - Access concrete keys & other input devices
- Screen
  - High-level interface
  - High-level objects that encapsulate a complete user interface component
    - Drawing by system software
    - Navigation, scrolling etc handled by system software
    - Applications cannot access concrete input mechanism

Other screen-related issues

- Game API (MIDP 2.0)
  - Sprites and Layering
  - Enables native implementation
- GUI enhancements
  - Builds on MIDP 1.0 LCDUI
  - Custom items, layout control, graphical enhancements...
  - Backward compatibility

MIDP 2.0 adds more features

- OTA (over the air) applications
  - Installation
    - Verify integrity, certificates, requested permissions
  - Invocation
    - Permissions
  - Previously extension, now required
- Networking extensions
  - HTTPS, SSL
  - Serial port
  - Sockets
  - Server Sockets
  - Datagrams
  - Network push feature
    - Network initiated MIDlet launch

Other services

- Timers

```
class myTask extends TimerTask {
  public void run() { ... }
}
myTimer = new Timer();
myTask = new myTask();
myTimer.schedule(myTask, 100, 1000);
```

- Access to resource files (`getResourceAsStream`)
- System properties (`java.lang.System.GetProperty`)
  - `microedition.platform` (laitteen nimi)
  - `microedition.encoding` (käytetyn merkistön enkoodaus)
  - `microedition.configuration` (käytetty konfiguraatio ja sen versio)
  - `microedition.profiles` (käytetty profiili ja sen versio)
  - `microedition.locale` (kieli ja maa)

J2ME extensions

- JSR-120: Wireless Messaging API
  - Short Message Service (SMS)
  - Unstructured Supplementary Service Data (USSD)
  - Cell Broadcast Service (CBS)
- JSR-135: Mobile Media API
  - Straightforward access and control of basic audio and multimedia resources and files
- JSR-172: J2MET Web Services Specification
  - Infrastructure for basic XML processing capabilities
  - Reuse of web service concepts when designing J2ME clients for enterprise services
  - Provides APIs and conventions for programming J2ME clients of enterprise services
  - Programming model for J2ME client communication with web services, consistent with that for other Java clients such as J2SE.

J2ME extensions

- JSR-177: Security and Trust Services for J2MET
  - Necessary step for a device to become trusted, i.e., to provide security mechanisms to support a wide variety of application-based services, such as access to corporate network,mobile commerce, and digital rights management
  - Model and a set of APIs that enable applications running on a J2ME device to communicate with a smartcard
- JSR-179: Location API for J2MET
  - Optional package that enables developers to write mobile, location-based applications for J2ME devices
- JSR-180: Session Initiation Protocol (SIP) for J2MET
  - Session Initiation Protocol (SIP) is used to establish and manage multimedia IP sessions
  - General SIP API for J2ME devices based on the SIP protocol defined by IETF and 3GPP, and targeting resource constrained platforms.

J2ME extensions

- JSR-184: Mobile 3D Graphics for J2MET
  - lightweight, interactive 3D graphics API, which sits alongside J2ME and MIDP as an optional package.
  - API targeted at devices that typically have very little processing power and memory, and no hardware support for 3D graphics or floating point math
- JSR-190: Event Tracking API for J2MET
  - Optional package that standardizes the tracking of application events in a mobile device
  - Submission of these event records to an event-tracking server via a standard protocol
- JSR-195: Information Module Profile

J2ME extensions

- JSR-68: J2MET Platform Specification
  - Defines the"ground rules" for the J2ME platform architecture and J2MEstandardization activities.
  - Formalizes the fundamental concepts behind J2ME, such as the notions of a configuration and profile
  - Defines how new J2ME APIs can be formed, e.g., by subsetting existingAPIs from Java 2 Platform, Standard Edition
- JSR-185: JavaT Technology for Wireless Industry
  - Defines how various technologies associated with MIDP work together to form a complete handset solution for the wireless services industry
  - Which optional packages fit with which profiles? How an end-to-end solution for interoperable Java applications will work?How the migration of applications can occur and to which profiles as the devices become more capable?

First Issue:
Event Handling

MIDP’s High-Level Interface

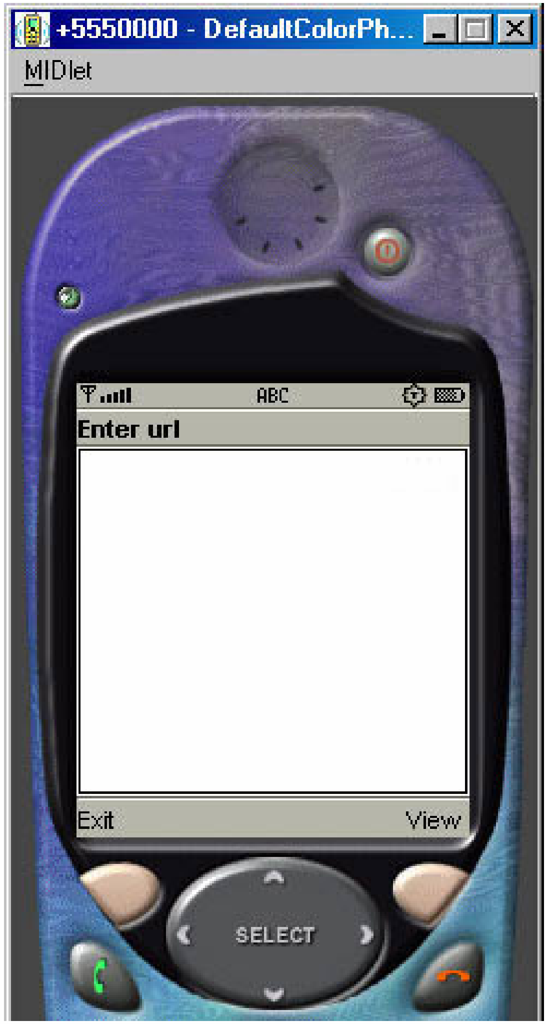
- High-Level MIDP events are divided into two categories:
  - Command : events triggered by keypresses on the device
  - Items: events that are the results of visual components changing on the display

Command Objects

- When an event occurs on a mobile device, a Command Object holds information about that event
- Commands are commonly represented with soft-buttons on the device
- The information includes:
  - Type of command executed
  - Label of the command
  - Priority

Command Objects

- Figure at the right shows two command objects, one labeled Exit and the other View
- If the current display cannot hold multiple command objects, a menu may be created to hold the objects



Event Processing

- The only MIDP components that can manage commands are the following:
  - Form
  - TextBox
  - Lixt
  - Canvas
- These will be illustrated later in the discussion

Event Processing

- The basic steps to process events with Command objects are as follows:
  - Create a Command Object
  - Add the Command to a Form, TextBox, List or Canvas
  - Create a Listener
- Upon detection of an event, the Listener generates a call to the method commandAction()

Command Processing Example

- The following code block shows an event procedure using a Command Object:

```
private Form fmMain;
private Command cmExit;
. . .
// Create Form and give it a title
fmMain = new Form(“My Form");
// Create Command object, with label, type and priority
cmExit = new Command("Exit", Command.EXIT, 1);
. . .
fmMain.addCommand(cmExit); // Add Command to Form
fmMain.setCommandListener(this); // Listen for Form events
. . .
public void commandAction(Command c, Displayable s)
{
   if (c == cmExit)
   {
        destroyApp(true);
        notifyDestroyed();
   }
}
```

Item Objects

- It is also possible to process events using Item Objects in MIDP
- Several of these Items are predefined for processing particular event types, for example:
  - DateField: allows the user to select the date and time that will display on screen
  - TextField: allows the user to enter a series of alpha-numeric and special characters

Event Processing

- With the exception of StringItem, Spacer, and ImageItem, each Item object has the ability to recognize events
- An event listener must be created before events from Item objects are acknowledged by the device
- When a change occurs in any Item Object, the method itemStateChanged() is called
- Within this method, you can identify which Item Object generated the event

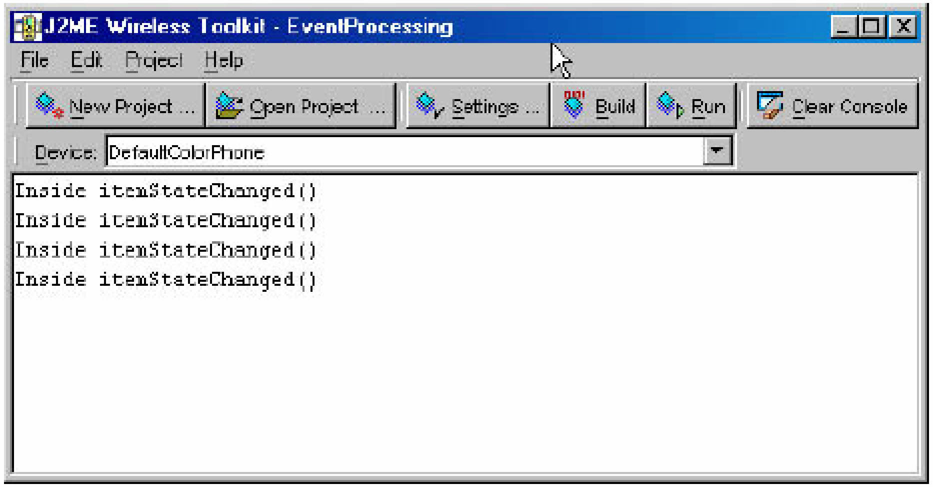
Item Processing Example

- The following code block shows simple event processing for a DateField Item Object:

```
private Form fmMain;
private DateField dfToday; // DateField Item
. . .
fmMain = new Form("Core J2ME"); // Create Form object
      // Create DateField
dfToday = new DateField("Today:", DateField.DATE);
. . .
fmMain.append(dfToday); // Add DateField to Form
fmMain.setItemStateListener(this);
      // Listen for Form events
. . .
public void itemStateChanged(Item item)
{
  // If the datefield initiated this event
  if (item == dfToday)
...
}
```

Detecting Item Events

- In our example MIDlet, as each character is input, an Item event is generated
- Monitor the WTK console to see that the print message we placed inside the method ItemStateChanged() are displayed

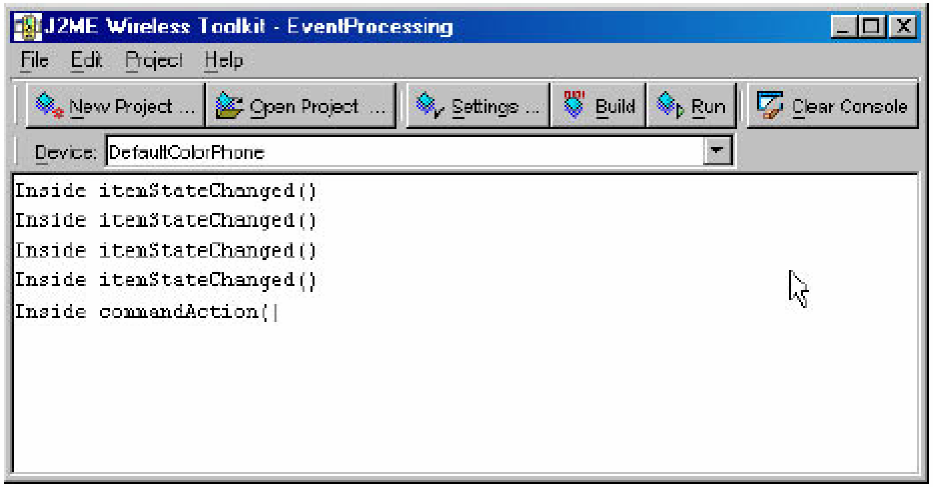


Device: DefaultColorPhone

```
Inside itemStateChanged()
Inside itemStateChanged()
Inside itemStateChanged()
Inside itemStateChanged()
```

Detecting Command Events

- When we chose to exit the MIDlet by pressing the soft-button with the Exit label, a Command event was generated
- The appropriate actions were taken inside the commandAction() method, this can be seen in the WTK console as well



Device: DefaultColorPhone

```
Inside itemStateChanged()
Inside itemStateChanged()
Inside itemStateChanged()
Inside itemStateChanged()
Inside commandAction(|
```

Display Objects

- These objects are designed to work with the device display:
  - Display
  - Displayable
  - Screen
- All three components comprise MIDP’s device display mechanism

Display Object

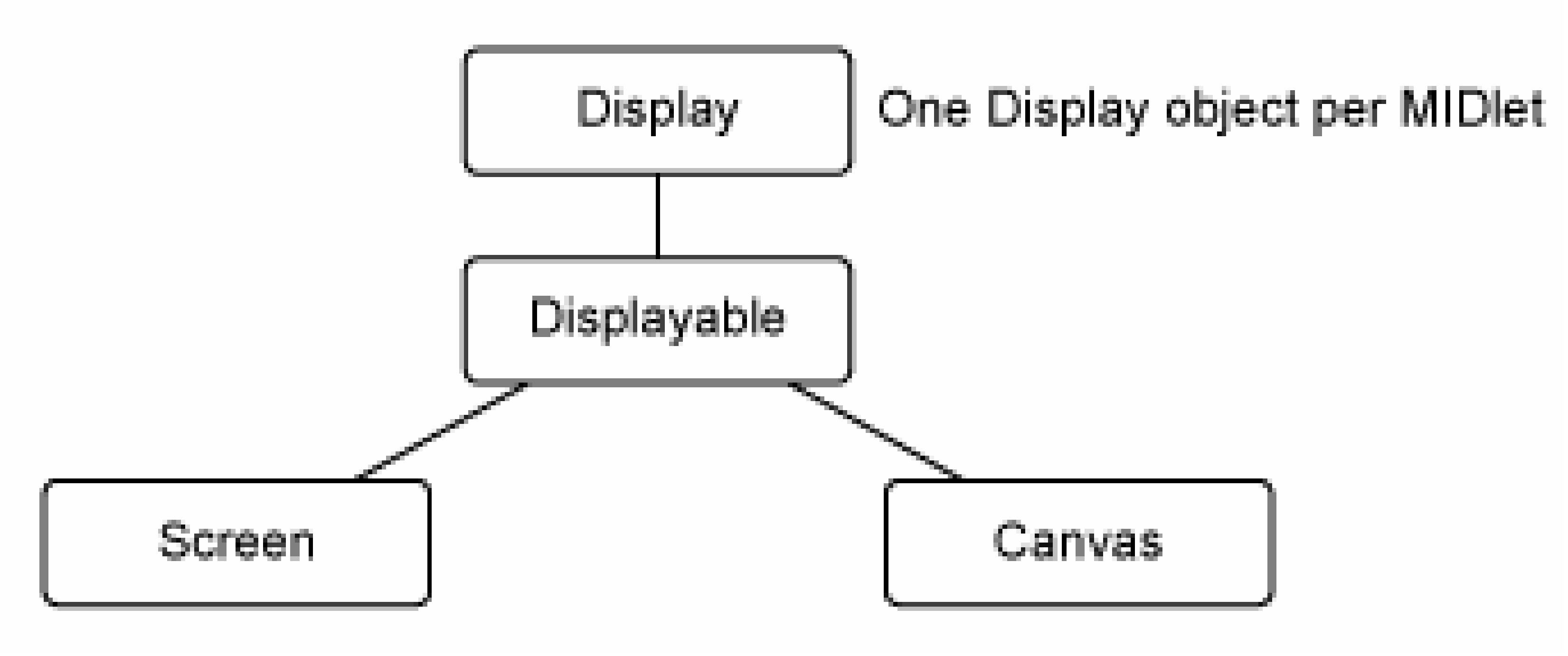
- A MIDlet has one instance of a Display Object
- Gives information about the current device display such as color support
- Includes methods for requesting that other objects be displayed
- Works as the manager of the device display controlling everything that can be seen on the device display

Displayable Object

- Although there is only one Display object per MIDlet, many objects within the MIDlet may have a displayable attribute
- The Displayable Object in MIDP contains two subclasses:
  - Screen
  - Canvas
- The following are the class definitions for each:

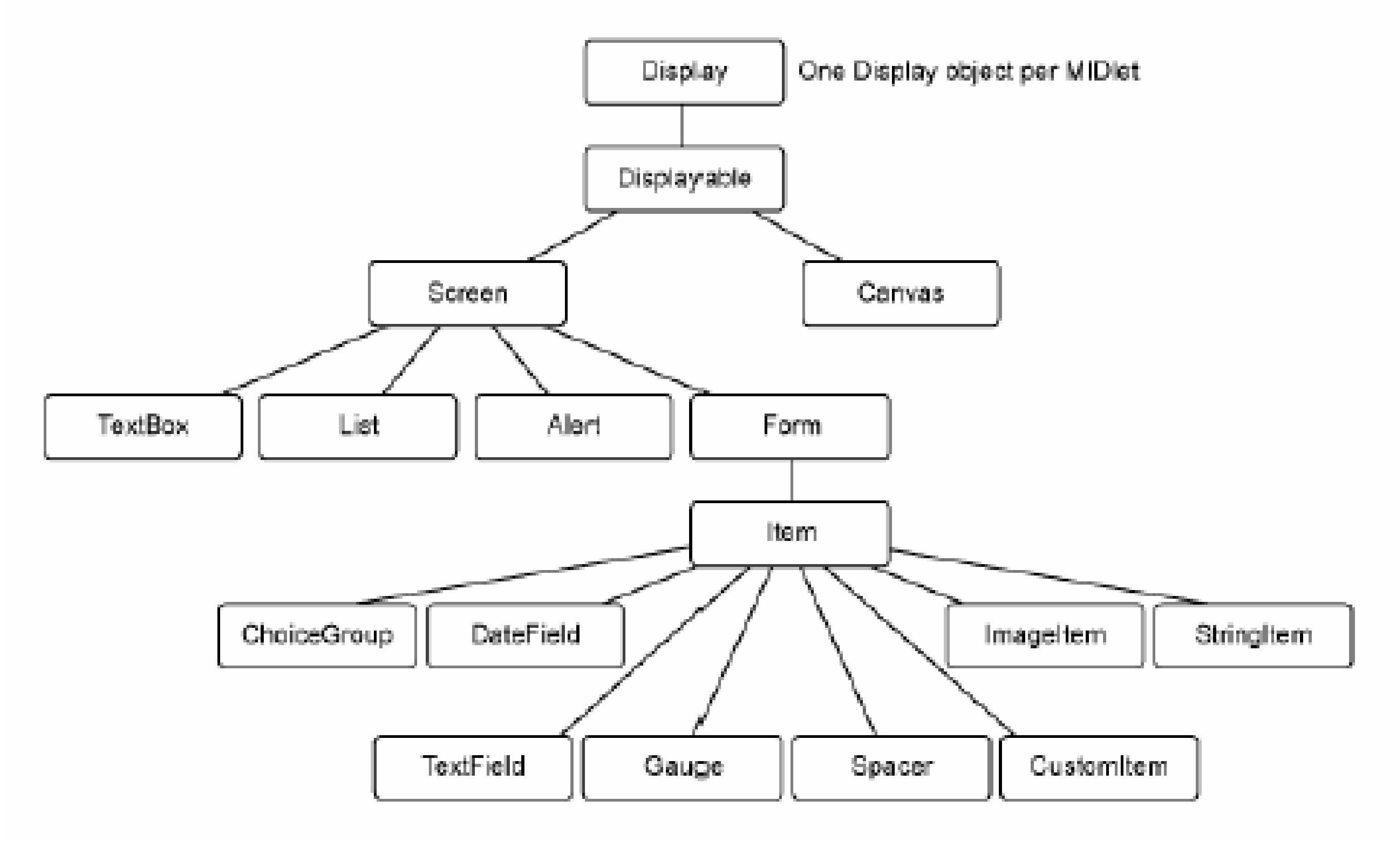
```
abstract public class Displayable
public abstract class Canvas extends Displayable
public abstract class Screen extends Displayable
```

Display Object Hierarchy



Screen Object

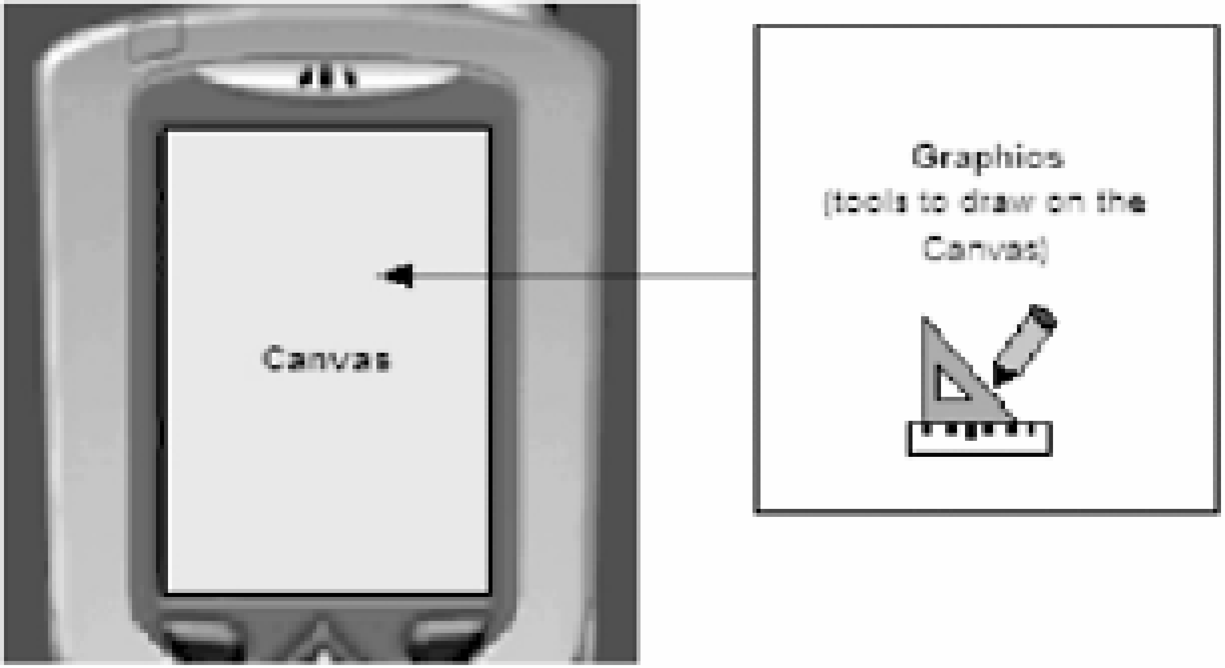
- Screen Objects are those that are visible on the device
- It has several subclasses shown below:



MIDP’s Low-Level Interface

- Low-Level UI provides a toolkit to do the following:
  - Move and draw graphics object
  - Render fonts
  - Capture direct key events
- Two classes really make up the low-level UI engine:
  - Canvas: a Displayable object (similar to Screen Object)
  - Graphics: also called the context, is the palette of tools with which to draw

Canvas and Graphics Relationship



The Canvas Object

- Canvas objects are not created similar to other Displayable objects
- This object serves as a base class from which you derive custom drawing objects
- In other words, the Canvas object provides the all the tools, but the user gives the content
- To invoke the object you derived from the Canvas class, the following method must be called:

  ```
  protected void paint(Graphics graphics)
  ```
- This method is responsible for rendering or drawing the Canvas control on to the screen

Canvas: Example

- First you have to get a handle to the Canvas object, this is done by declaring an instance of the object in the MIDlet.
- The Canvas class is extended to the users custom made Canvas tool

```
class TestCanvas extends Canvas implements CommandListener
{
      private Command cmdExit;
      . . .
      display = Display.getDisplay(this);
      cmdExit = new Command("Exit", Command.EXIT, 1);
      addCommand(cmdExit);
      setCommandListener(this);
      . . .

      protected void paint(Graphics g)
      {
      // here goes the user’s routines for drawing and object  movement
      }
}
// create an instance of the Canvas
TestCanvas canvas = new TestCanvas(this);
```

Continued. . .

- the paint() method, which is activated everytime an event on the Canvas (display) object occurs, should contain the draw and move routines for the MIDlet
- Example below shows initial setup of the Canvas object

```
protected void paint(Graphics g)
{
     // Set background color to white
     g.setColor(255, 255, 255);

     // Fill the entire canvas
     g.fillRect(0, 0, getWidth(), getHeight());
}
```

Event Handling in Canvas Objects

- Similar to Screen and other Displayable objects, the Canvas object may implement a CommandListener to process events
- The commandAction() method contains the events trapped by sift-key buttons
- The Canvas object also has access to 12 constant key-codes defining the mobile device keys:

  `KEY_NUM0, KEY_NUM1, . . ., KEY_NUM9,`

  `KEY_STAR and KEY_POUND`
- These keys are what comprises the keypad on mobile phones.
- There are exceptions on phones like the Nokia N-Gage, where the keys are placed differently

KeyCodes

- To process the key codes (trap the user input on the keypad), the following methods are intuitively defined in MIDP:

```
    void keypressed(int keyCode)
    void keyReleased(int keyCode)
    void keyRepeated(int keyCode)
    boolean hasRepeatEvents()
`   String getKeyName(int keyCode)
```

- Among the five methods, the keyRepeated() and hasRepeatEvents() doesn’t appear frequently on Game programs, since they are used to identify which letter-character is desired on a given key (i.e for ‘b’ you need to press KEY_NUM1 twice)

Game Specific Keys

- The User may map the keys on the keypad to game specific actions, typically we have the following:

```
UP, DOWN, LEFT, RIGHT, FIRE,
GAME_A, … GAME_Z
```

- Key assignment vary from device to device, most phones in the series 40 and series 60 have similar configurations
- If a device doesn’t have directional arrows, the left, right, down and up actions are assigned to keypad numbers 6, 4, 2, and 8 respectively

Processing Game Actions

- All user initiated game actions are processed through key presses and the appropriate device reaction must be placed under one of the five key processing methods
- Example below shows how different keypresses may direct the action of a character in a game:

```
protected void keyPressed(int keyCode)
{
     switch (getGameAction(keyCode))
     {
              case Canvas.FIRE:
                       shoot();
                       break;
              case Canvas.RIGHT:
                       goRight();
                       break;
              . . .
     }
}
```

Graphics Class

- Among the various objects used for J2ME game development, the Graphics class is the most extensive

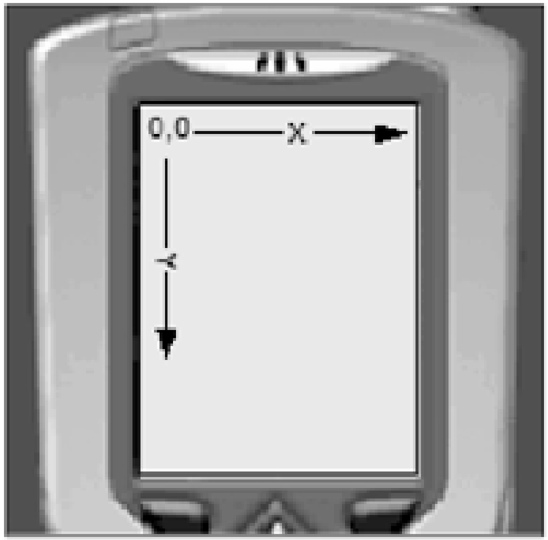
- The Graphics class provides the facility to manipulate different Displayable aspects such as:
  - Color
  - KeyStrokes and BrushStrokes
  - Shapes
  - Fonts
  - Links or Anchor Points
  - Graphical Text
  - Images and Pictures

Graphics Class and Canvas Connection

- The Graphics Class drawing tools are used to carry out basic 2D rendering on th Canvas object.
- Most attributes of the Graphics Class are shared with the Canvas Object, this translates to automatic configuration of both even when only one is manipulated

Coordinates

- Before we go further, we need to know the coordinate system in an MIDP display
- The origin is located on the display at the top left corner
- This is assigned the XY value (0,0)



Coordinates

- All draw methods (graphics) take coordinates according to the same format
- However, it is possible to change the origin on the screen, for example:

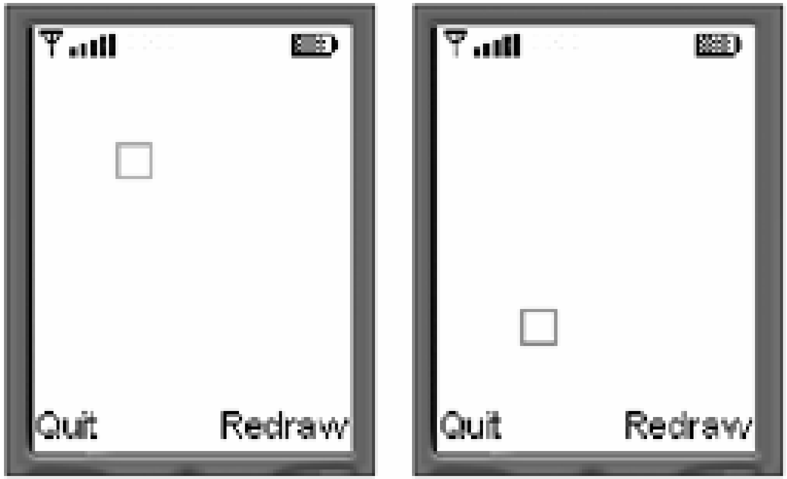
```
    graphics.drawRect(25, 25, 10, 10);
```

- This code will result in a rectangle appearing at position (25,25) and extending for 10 pixels in both directions
- Now we can translate the whole display world 50 pixels on the y-Axis only through:

```
    graphics.translate(0,50);
    graphics.drawRect(25, 25, 10, 10);
```

- The second rectangle is now drawn lower on the screen (see next slide)

Coordinates



- You should note that multiple calls to translate are cumulative for the same display object like canvas or graphics
- Therefore, the following code will set a translation of (20,70), not (20,20)
- You can also use negative numbers to adjust the translation

End of Session