Módulo 5

Desenvolvimento de Aplicações Móveis

Lição 3

Interface de Alto Nível para o Usuário

Versão 1.0 - Set/2007

Autor
xxx

Equipe
Rommel Feria
John Paul Petines

Necessidades para os Exercícios

Sistemas Operacionais Suportados

NetBeans IDE 5.5 para os seguintes sistemas operacionais:

- Microsoft Windows XP Profissional SP2 ou superior
- Mac OS X 10.4.5 ou superior
- Red Hat Fedora Core 3
- Solaris™ 10 Operating System (SPARC® e x86/x64 Platform Edition)

NetBeans Enterprise Pack, poderá ser executado nas seguintes plataformas:

- Microsoft Windows 2000 Profissional SP4
- Solaris™ 8 OS (SPARC e x86/x64 Platform Edition) e Solaris 9 OS (SPARC e x86/x64 Platform Edition)
- Várias outras distribuições Linux

Configuração Mínima de Hardware

Nota: IDE NetBeans com resolução de tela em 1024x768 pixel

Sistema Operacional	Processador	Memória	HD Livre
Microsoft Windows	500 MHz Intel Pentium III workstation ou equivalente	512 MB	850 MB
Linux	500 MHz Intel Pentium III workstation ou equivalente	512 MB	450 MB
Solaris OS (SPARC)	UltraSPARC II 450 MHz	512 MB	450 MB
Solaris OS (x86/x64 Platform Edition)	AMD Opteron 100 Série 1.8 GHz	512 MB	450 MB
Mac OS X	PowerPC G4	512 MB	450 MB

Configuração Recomendada de Hardware

Sistema Operacional	Processador	Memória	HD Livre
Microsoft Windows	1.4 GHz Intel Pentium III workstation ou equivalente	1 GB	1 GB
Linux	1.4 GHz Intel Pentium III workstation ou equivalente	1 GB	850 MB
Solaris OS (SPARC)	UltraSPARC IIIi 1 GHz	1 GB	850 MB
Solaris OS (x86/x64 Platform Edition)	AMD Opteron 100 Series 1.8 GHz	1 GB	850 MB
Mac OS X	PowerPC G5	1 GB	850 MB

Requerimentos de Software

NetBeans Enterprise Pack 5.5 executando sobre Java 2 Platform Standard Edition Development Kit 5.0 ou superior (JDK 5.0, versão 1.5.0_01 ou superior), contemplando a Java Runtime Environment, ferramentas de desenvolvimento para compilar, depurar, e executar aplicações escritas em linguagem Java. Sun Java System Application Server Platform Edition 9.

- Para **Solaris**, **Windows**, e **Linux**, os arquivos da JDK podem ser obtidos para sua plataforma em http://java.sun.com/j2se/1.5.0/download.html
- Para **Mac OS X**, Java 2 Plataform Standard Edition (J2SE) 5.0 Release 4, pode ser obtida diretamente da Apple's Developer Connection, no endereço: http://developer.apple.com/java (é necessário registrar o download da JDK).

Para mais informações: http://www.netbeans.org/community/releases/55/relnotes.html

Colaboradores que auxiliaram no processo de tradução e revisão

Aécio Júnior
Alexandre Mori
Alexis da Rocha Silva
Allan Souza Nunes
Allan Wojcik da Silva
Anderson Moreira Paiva
Andre Neves de Amorim
Angelo de Oliveira
Antonio Jose R. Alves Ramos
Aurélio Soares Neto
Bruno da Silva Bonfim
Carlos Fernando Gonçalves
Denis Mitsuo Nakasaki
Fábio Bombonato
Fabrício Ribeiro Brigagão
Francisco das Chagas
Frederico Dubiel
Herivelto Gabriel dos Santos
Jacqueline Susann Barbosa
João Vianney Barrozo Costa
Kefreen Ryenz Batista Lacerda
Kleberth Bezerra G. dos Santos
Leandro Silva de Morais
Leonardo Ribas Segala
Lucas Vinícius Bibiano Thomé
Luciana Rocha de Oliveira
Luiz Fernandes de Oliveira Junior
Marco Aurélio Martins Bessa
Maria Carolina Ferreira da Silva
Massimiliano Giroldi
Mauro Cardoso Mortoni
Paulo Afonso Corrêa
Paulo Oliveira Sampaio Reis
Pedro Henrique Pereira de Andrade
Ronie Dotzlaw
Seire Pareja
Sergio Terzella
Vanessa dos Santos Almeida
Robson Alves Macêdo

Auxiliadores especiais

Revisão Geral do texto para os seguintes Países:

- **Brasil** – Tiago Flach
- **Guiné Bissau** – Alfredo Cá, Bunene Sisse e Buon Olossato Quebi – ONG Asas de Socorro

Coordenação do DFJUG

- **Daniel deOliveira** – JUGLeader responsável pelos acordos de parcerias
- **Luci Campos -** Idealizadora do DFJUG responsável pelo apoio social
- **Fernando Anselmo -** Coordenador responsável pelo processo de tradução e revisão, disponibilização dos materiais e inserção de novos módulos
- **Rodrigo Nunes -** Coordenador responsável pela parte multimídia
- **Sérgio Gomes Veloso -** Coordenador responsável pelo ambiente JEDI™ (Moodle)

Agradecimento Especial

John Paul Petines – Criador da Iniciativa JEDI™
Rommel Feria – Criador da Iniciativa JEDI™

1. Objetivos

A interface de usuário *MIDP* foi desenhada para aparelhos portáteis. Aplicações *MIDP* são mostradas no limite de área da tela. Dispositivos de memória é o fator pelo qual os aparelhos portáteis possuem pouca memória.

Com os diferentes tipos de aparelhos móveis, dos mais variados modelos de telefones móveis aos PDAs, a interfase de usuário MIDP foi desenhada para ser flexível o bastante para ser utilizável em todos estes aparelhos.

Ao final desta lição, o estudante será capaz de:

- Conhecer as vantagens e desvantagens de se utilizar interfaces gráficas em alto e baixo nível
- Projetar *MIDlets* usando componentes de interface gráfica de alto nível
- Identificar as diferentes subclasses de tela
- Saber que diferentes itens podem ser utilizados na forma de objeto

2. Interface de usuário *MIDP*

MIDP possui classes que podem prover interfaces de usuário com altos e baixos níveis de funções. Interfaces gráficas de alto nível são desenhadas para serem flexíveis. A aparência desses componentes não está definida nas especificações. A aparência varia de aparelho para aparelho, mas o programador pode ficar seguro de que o comportamento de alto nível dos componentes será o mesmo em todas as especificações e implementações.

Interfaces Gráficas de Alto Nível	***Interfaces Gráficas de Baixo Nível***
Altamente portátil entre aparelhos	Pode ser específico do dispositivo
Aparência é a mesma nos aparelhos	Aplicação com aparência específica
Navegação do tipo "rolagem" é encapsulada	Tem que implementar suas próprias formas de navegação
Não podem definir a aparência real	Definem a aparência em nível de pixel
Não são acessadas por aparelhos com características específicas	O acesso ao baixo nível de entrada é por intermédio do pressionamento de teclas

Figura 1: Comparação entre alto nível e baixo nível

2.1. Display

A principal parte da interface de usuário *MIDP* é a classe *Display*. Existe uma e apenas uma instância do *Display* para cada *MIDlet*. *MIDlet* pode pegar a referência do objeto *Display* utilizando o método estático *getDisplay*() da classe *Display*, e enviar a referência de um objeto do *MIDlet*.

MIDlet garante que o objeto *Display* não será modificado enquanto existir uma instância de *MIDlet*. Isso significa que o objeto retornado ao executamos o método *getDisplay*() é o mesmo não importando se essa chamada foi realizada nos métodos *startApp*() ou *destroyApp*() (veja mais sobre o ciclo de vida do *Midlet*).

2.2. Displayable

Apenas um componente *Displayable* pode ser mostrado de cada vez. Por padrão, *Displayable* não é mostrado no visor. Pode ser visível por intermédio da chamada do método *setCurrent*(). Este método pode ser chamado quando a aplicação se inicia, caso contrário, uma tela escura será mostrada ou a aplicação não se iniciará.

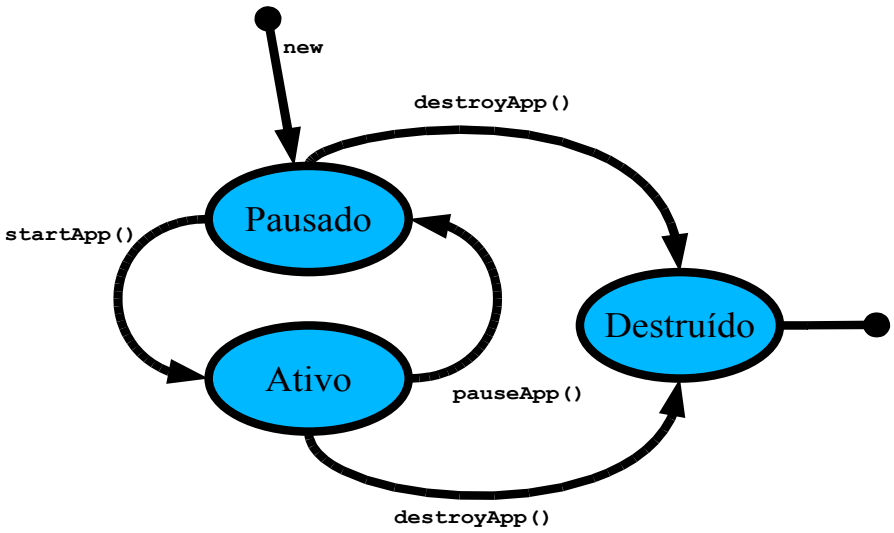


Figura 2: Ciclo de Vida do MIDlet

O método *startApp*() da classe *MIDlet* é o local onde é possível inserir a chamada do método *setCurrent*(). Entretanto é necessário levar em consideração que o método *startApp*() pode ser chamado mais de uma vez durante a existência do *MIDlet*. Quando um *MIDLet* entra em pausa por intermédio do método *pauseApp*(), pode ser que tenha ocorrido uma chamada telefônica para atender. O método *startApp*() pode ser chamado novamente (após o finalização da chamada telefônica), assim como pela chamada do método *setCurrent*() dentro do método *startApp*(). Iremos obscurecer a tela que estava sendo mostrada antes do aplicativo entrar em pausa (ou seja, antes da chamada telefônica).

O componente *Displayable* pode ter um título, um conjunto de comandos, um *commandListener* e um Relógio.

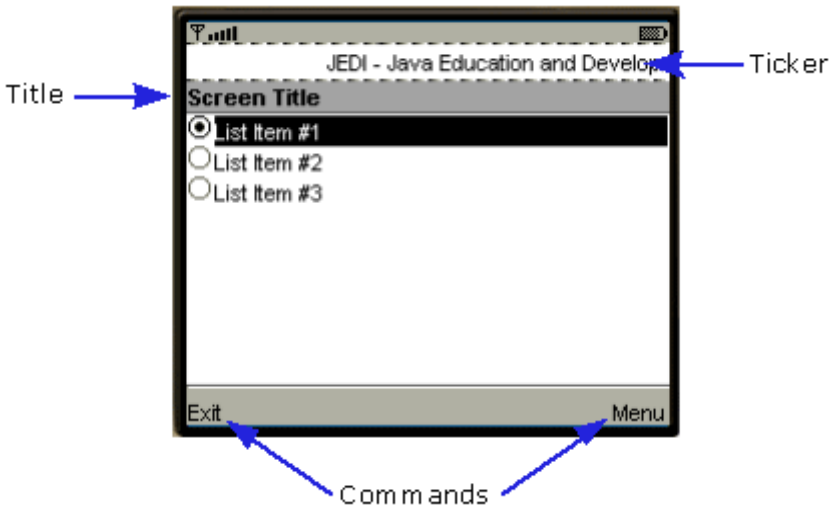


Figura 3: Propriedades de Displayable

2.3. Title

O componente *Displayable* possui um título associado. A posição e aparência deste título é específica para cada tipo de aparelho e só pode ser determinada quando a aplicação estiver rodando. O título é associado ao *Displayable* por intermédio da chamada do método *setTitle*(). Chamando este método, o título do componente *Displayable* poderá mudar imediatamente. Se um *Displayable* está sendo apresentado na tela corrente, a especificação de estado do *MIDP* e o título devem ser alterados pela implementação "tão logo quanto seja permitido fazê-la".

Passando um parâmetro nulo (*null*) para o método *setTitle*() remove-se o título do componente *Displayable*. Mudando ou removendo o título, pode-se afetar o tamanho da área disponível. Se a troca na área do visor ocorrer, o *MIDlet* pode ser modificada por intermédio do método *sizeChanged*().

2.4. Comando

Devido ao limite do tamanho da tela, *MIDP* não define uma barra de menu dedicada. No lugar da barra de menu *MIDlets* possuem comandos. Comandos são usualmente implementados pelo *MIDP* que podem ser chaves rápidas ou itens de menu. O objeto da classe *Command* contém informações somente sobre as ações que foram capturadas quando ativado, e não códigos que serão executados quando for selecionado.

A propriedade *commandListener* do componente *Displayable* contém as ações que serão executadas na ativação dos comandos. A *commandListener* é a interface que especifica um método simples:

```
public void commandAction(Command comando, Displayable mostravel)
```

O mapeamento dos comandos no aparelho depende do número de botões rápidos ou botões programáveis disponíveis no aparelho. Se o número de comandos não se ajustar ao número de botões rápidos (*softbuttons*), o aparelho pode colocar alguns ou todos os comandos dentro do menu e mapear este menu para os botões rápidos com o título.

```
Command exitCommand = new Command("Exit", Command.EXIT, 1);
Command newCommand = new Command("New Item", Command.OK, 1);
Command renameCommand = new Command("Rename Item", Command.OK, 1);
Command deleteCommand = new Command("Delete Item", Command.OK, 1);
...
list.addCommand(exitCommand);
list.addCommand(newCommand);
list.addCommand(renameCommand);
list.addCommand(deleteCommand);
```

Figura 4: Amostra de mapeamento para múltiplos comandos

Um *Command* possui um label curto, um label longo e opcional, um tipo e uma prioridade.

2.4.1. Label

O tamanho reduzido da tela dos dispositivos é sempre um fator relevante quando se desenvolvem aplicações *MIDP*. Para os *labels* de comando, essa suposição também é aplicável. Os *labels* devem ser curtos, porém descritivos, para tanto devem caber na tela e serem compreensíveis para o usuário final.

Quando um *label* longo for especificado, deverá ser apresentado toda vez que o sistema considerar apropriado. Não existe uma chamada de API que especifique qual *label* deverá ser mostrado. É perfeitamente possível que alguns comandos apresentem um *label* curto enquanto que outros apresentem um longo.

2.4.2. Tipo de Comando

A maneira pela qual um comando é apresentado depende do dispositivo utilizado. Um programador pode especificar o tipo para este comando. Este tipo servirá como um auxílio para saber onde o comando deverá ser colocado. Não existe nenhuma forma de definir perfeitamente onde o comando deve ser apresentado na tela.

Os diferentes tipos de comandos são:

```
Command.OK, Command.BACK,
Command.CANCEL, Command.EXIT,
Command.HELP, Command.ITEM,
Command.SCREEN, Command.STOP
```

Figura 5: A apresentação dos comandos é diferente em cada aparelho.

2.4.3. Prioridade de Comandos

A aplicação pode especificar a importância dos comandos na propriedade *priority* (prioridade). Esta é uma propriedade com um argumento do tipo *Integer* e quanto menor o número, maior a prioridade. Esta propriedade é também uma dica para ajudar a saber como o comando deverá ser posicionado. Casualmente a implementação determina a posição dos comandos pelo seu tipo. Se existir mais de um comando do mesmo tipo, a prioridade é normalmente considerada no posicionamento dos comandos.

2.5. CommandListener

O *CommandListener* é uma interface que possui um único método:

```
void commandAction(Command command, Displayable displayable)
```

O método *commandAction*() é chamado quando um comando é selecionado. A variável do comando é uma referência para o comando que foi selecionado. *Displayable* é onde o comando está localizado e onde a ação selecionada acontece.

O método *commandAction* deverá retornar imediatamente, caso contrário a execução da aplicação poderá ser bloqueada. Isto se deve pelo fato das especificações do *MIDP* não requisitarem implementações para a criar uma tarefa separada para cada evento entregue.

2.6. Ticker

O *Ticker* é uma contínua linha de rolagem de texto que acompanha a tela. O método construtor do *Ticker* aceita uma String para ser exibida. Ele só possui dois outros métodos, o padrão get e set para este texto: *String getString*() e *void setString*(String texto). Não existe possibilidade da aplicação controlar a velocidade e a direção do texto. A rolagem não pode ser pausada ou interrompida.

Se uma quebra de linha for embutida no texto, está não será apresentada na tela. Todas as linhas de texto deverão ser exibidas como uma única linha de texto rolante.

Um objeto *Ticker* é anexado a um dispositivo de exibição chamando-se o método *setTicker*(). Caso já exista um objeto *Ticker* anexado ao dispositivo, este será substituído pelo novo objeto passado como parâmetro.

Passando-se um parâmetro null para o método *setTicker*() ocorre a remoção qualquer objeto *Ticker* anexado do dispositivo. A remoção de um objeto *Ticker* do dispositivo pode afetar o tamanho de área válido para o conteúdo a ser exibido. Se uma mudança no tamanho da área de

exibição ocorrer, o *MIDlet* deverá ser notificado pelo método *sizeChanged*().

Os objetos *Displayable* podem compartilhar uma instância do objeto *Ticker*.

2.7. Screen

A classe *Screen* é a principal classe abstrata utilizada para aplicações gráficas de alto nível enquanto que a classe *Canvas* é a principal classe abstrata para aplicações gráficas de baixo nível.

Existem quatro subclasses da classe abstrata *Screen*: *Form*, *TextBox*, *List* e *Alert*.

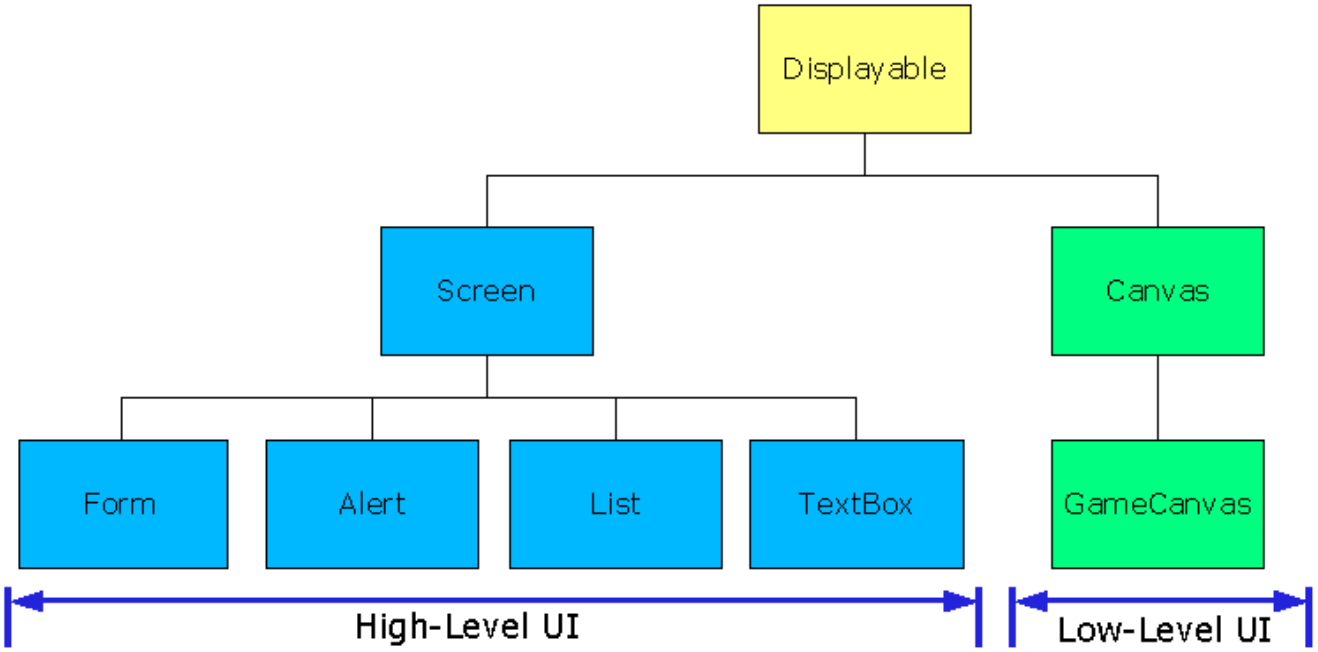


Figura 6: Hierarquia de classes

2.8. Item

Componentes do tipo *Item* podem ser colocados em um componente do tipo *container*, como um *Form* ou um *Alert*. Um item pode possuir as seguintes propriedades:

Propriedade	***Valor Padrão***
Label	Especificado no construtor da subclasse
Commands	nenhum
defaultCommand	nulo
ItemCommandListener	nulo
Layout directive	LAYOUT_DEFAULT
Preferências de altura e largura	-1 (destravado)

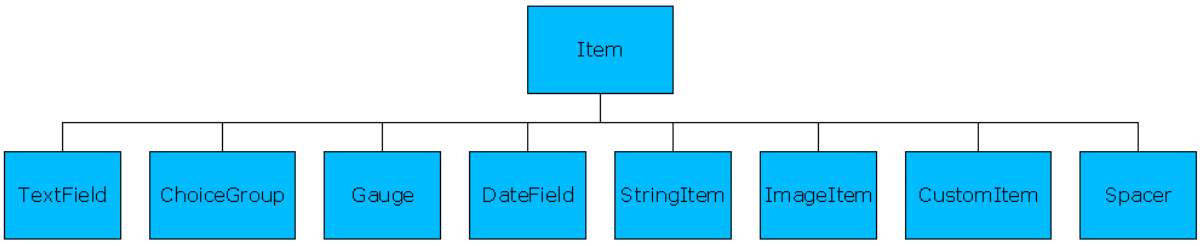


Figura 7: Hierarquia de Classe Item

A diretiva *Layout* especifica o formato de um item dentro de um objeto do tipo *Form*. A diretiva *Layout* pode ser combinada usando o operador de comparação binária OR (|). Todavia, diretivas de mesma orientação são mutuamente exclusiva.

Estas são as diretivas de alinhamento horizontal mutuamente exclusivas:

```
LAYOUT_LEFT
LAYOUT_RIGHT
LAYOUT_CENTER
```

Estas são as diretivas de alinhamento vertical mutuamente exclusivas:

```
LAYOUT_TOP
LAYOUT_BOTTOM
LAYOUT_VCENTER
```

Outras diretivas de *layout* não mutuamente exclusiva são:

```
LAYOUT_NEWLINE_BEFORE
LAYOUT_NEWLINE_AFTER
LAYOUT_SHRINK
LAYOUT_VSHRINK
LAYOUT_EXPAND
LAYOUT_VEXPAND
LAYOUT_2
```

3. Alert

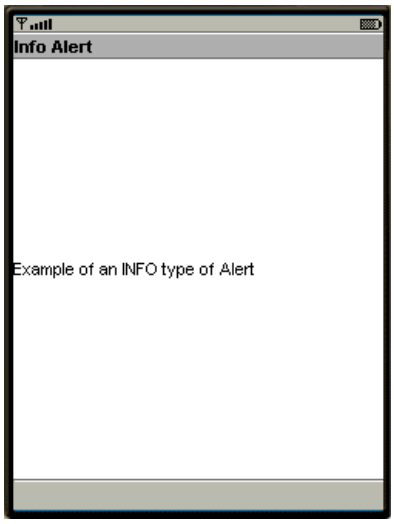
A classe *Alert* gera uma tela na qual é possível exibir um texto e uma imagem. É um componente para exibir erro, aviso, texto e imagem informativa ou trazer uma tela de confirmação para o usuário.

É exibido por um determinado período de tempo. Este tempo pode ser fixado utilizando-se o método *setTimeout*() e especificado em milissegundos. Pode ser construído para ser exibido até que o usuário ative um comando *Done* dentro do intervalo de tempo de espera especificado pela contstante Alert.FOREVER.

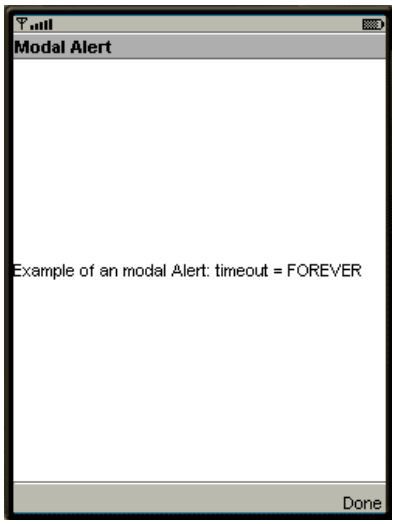
O *Alert* pode também exibir um componente do tipo *Gauge* como um indicador. Quando um *Alert* conter um texto esse não ajustará a tela inteira e deverá ser rolado. *Alert* é fixado automaticamente para modal (tempo de espera fixado para Alert.FOREVER).

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

public class AlertMidlet extends MIDlet implements CommandListener {
    Display display;
    Form mainForm;
    Command exitCommand = new Command("Exit", Command.EXIT, 0);
    Command okCommand = new Command("Ok", Command.OK, 0);
    Gauge gauge = new Gauge(null, false, 5, 0);
    Command[] commands = {
        new Command("Alarm", Command.OK, 0),
        new Command("Confirmation", Command.OK, 0),
        new Command("Info", Command.OK, 0),
        new Command("Warning", Command.OK, 0),
        new Command("Error", Command.OK, 0),
        new Command("Modal", Command.OK, 0)
    };

    Alert[] alerts = {
        new Alert("Alarm Alert",
                "Example of an Alarm type of Alert",
                null, AlertType.ALARM),
        new Alert("Confirmation Alert",
                "Example of an CONFIRMATION type of Alert",
                null, AlertType.CONFIRMATION),
        new Alert("Info Alert",
                "Example of an INFO type of Alert",
                null, AlertType.INFO),
        new Alert("Warning Alert",
                "Example of an WARNING type of Alert, w/ gauge indicator",
                null, AlertType.WARNING),
        new Alert("Error Alert",
                "Example of an ERROR type of Alert, w/ an 'OK' Command",
                null, AlertType.ERROR),
        new Alert("Modal Alert",
                "Example of an modal Alert: timeout = FOREVER",
                null, AlertType.ERROR),
    };
    public AlertMidlet(){
        mainForm = new Form("JEDI: Alert Example");

        mainForm.addCommand(exitCommand);
        for (int i=0; i< commands.length; i++){
            mainForm.addCommand(commands[i]);
        }
        mainForm.setCommandListener(this);
        // Adiciona um objeto Gauge e envia o tempo limite
        alerts[3].setIndicator(gauge);
        alerts[3].setTimeout(5000);
        // Adiciona um comando para este Alert
```

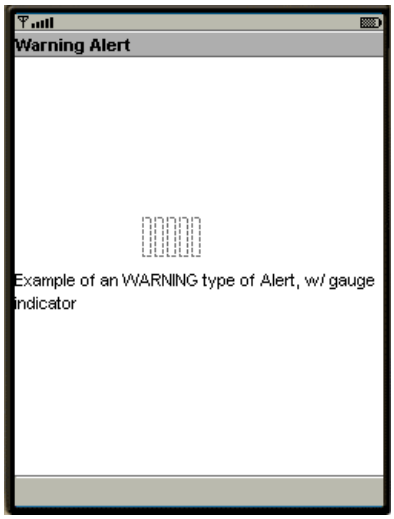
```
            alerts[4].addCommand(okCommand);
            // Define o Alert como Modal
            alerts[5].setTimeout(Alert.FOREVER);
        }

        public void startApp() {
            if (display == null){
                display = Display.getDisplay(this);
                display.setCurrent(mainForm);
            }
        }

        public void pauseApp() {}

        public void destroyApp(boolean unconditional) {}

        public void commandAction(Command c, Displayable d){
            if (c == exitCommand){
                destroyApp(true);
                notifyDestroyed(); // Exit
            }
            for (int i=0; i<commands.length; i++){
                if (c == commands[i]){
                    display.setCurrent(alerts[i]);
                }
            }
        }
    }
```



INFO Alert



Alert Modal



Alert com o indicador gauge

Figura 8: Diferentes tipos de Alert

4. List

A classe *List* é uma subclasse de *Screen* e fornece uma lista de escolhas. Este objeto pode assumir três modelos: IMPLICIT, EXCLUSIVE ou MULTIPLE.

List é IMPLICIT e o usuário executar o botão "select", o método *commandAction*() da classe *List* será chamada. O comando padrão é List.SELECT_COMMAND. O comando *commandListener* pode ser chamado através do comando padrão List.SELECT_COMMAND.

O método *getSelectedIndex*() retorna o índice do elemento atualmente selecionado para os tipos IMPLICIT e EXCLUSIVE. Para o tipo MULTIPLE, o método *getSelectedFlags*() retorna um atributo do tipo *boolean* contendo o estado dos elementos. O método *isSelected*(int index) retorna o estado do elemento na posição de índice dada.

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

public class ListMidlet extends MIDlet implements CommandListener {
    Display display;
    List list;
    Command exitCommand = new Command("Exit", Command.EXIT, 1);
    Command newCommand = new Command("New Item", Command.OK, 1);
    Command renameCommand = new Command("Rename Item", Command.OK, 1);
    Command deleteCommand = new Command("Delete Item", Command.OK, 1);
    Ticker ticker = new Ticker(
         "JEDI - Java Education and Development Initiative");

    public ListMidlet(){
        list = new List("JEDI: List Example", List.IMPLICIT);
        list.append("List Item #1", null);
        list.append("List Item #2", null);
        list.append("List Item #3", null);
        list.setTicker(ticker);
        list.addCommand(exitCommand);
        list.addCommand(newCommand);
        list.addCommand(renameCommand);
        list.addCommand(deleteCommand);
        list.setCommandListener(this);
    }

    public void startApp() {
        if (display == null){
            display = Display.getDisplay(this);
            display.setCurrent(list);
        }
    }

    public void pauseApp() {
    }

    public void destroyApp(boolean unconditional) {
    }

    public void commandAction(Command c, Displayable d){
        if (c == exitCommand){
            destroyApp(true);
            notifyDestroyed(); // Exit
        }
        if (c == List.SELECT_COMMAND){
            int index = list.getSelectedIndex();
            String currentItem = list.getString(index);
            // realiza algo
        }
    }
}
```

Lista IMPLICIT ***Lista EXCLUSIVE*** ***Lista MULTIPLE***

Figura 9: Tipos de lista

5. TextBox

A classe *TextBox* é a subclasse de *Screen* que pode ser usada para se obter a entrada de texto do usuário. Permite que o usuário incorpore e edite o texto. É similar à classe *TextField* (ver o item sobre TextField) pois permite a entrada de *constraints* e de modalidades. Sua diferença em relação a classe *TextField* é que o usuário pode inserir uma nova linha (quando a *constraint* da entrada é informada).

O texto digitado no objeto *TextBox* pode ser recuperado utilizando o método *getString*().

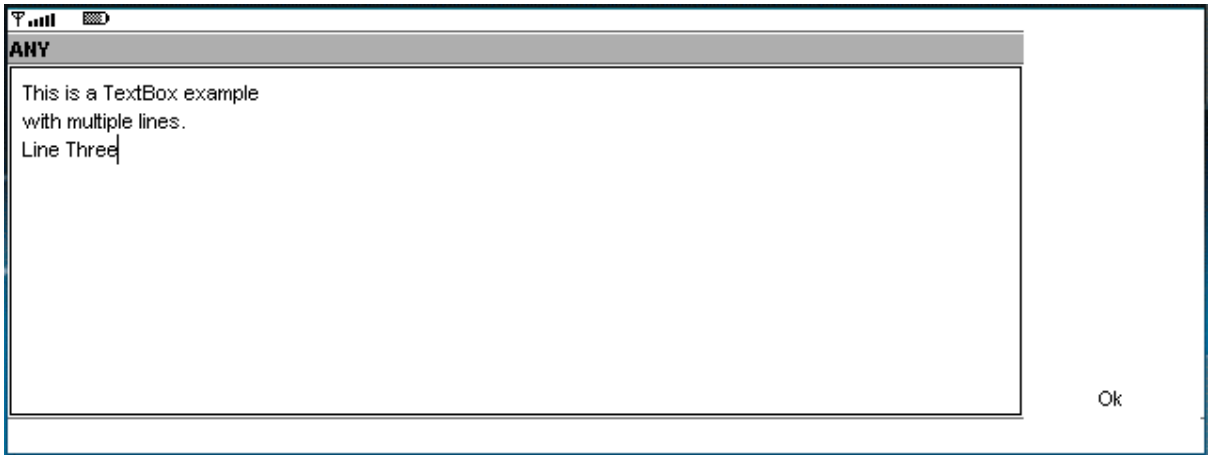


Figura 10: TextBox com múltiplas linhas

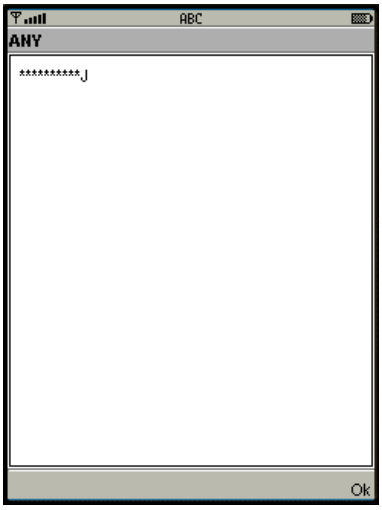


Figura 11: TextBox com o PASSWORD modificado

6. Form

A classe *Form* é uma subclasse da *Screen*. É um contêiner para itens das subclasses, tais como os objetos das classes *TextField*, *StringItem*, *ImageItem*, *DateField* e *ChoiceGroup*. Controla a disposição dos componentes e a transversal entre os componentes e o desdobramento da tela.

Itens são adicionados e inseridos a um objeto do tipo *Form* usando os métodos *append*() e *insert*(), respectivamente. Itens são eliminados usando o método *delete*(). Itens podem ser substituídos usando o método *set*(). Itens são referenciados usando o índice de base zero.

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

public class MidletPrinc extends MIDlet implements CommandListener {

  private Command exitCmd = new Command("Exit", Command.EXIT, 0);
  private ScreenForm form;
  private Display display;

  public void startApp() {
    if (display == null) {
      form = new ScreenForm("Test Form");
      display = Display.getDisplay(this);
    }
    display.setCurrent(form);
  }
  public void commandAction(Command command, Displayable displayable) {
    if (command == exitCmd) {
      try {
        destroyApp(true);
      } catch (MIDletStateChangeException ex) {
        ex.printStackTrace();
      }
      notifyDestroyed();
    }
  }
  protected void pauseApp() {
  }
  protected void destroyApp(boolean b) throws MIDletStateChangeException {
  }
  class ScreenForm extends Form {
    public ScreenForm(String title) {
      super(title);
      addCommand(exitCmd);
      setCommandListener(MidletPrinc.this);

      // Instruções para o Form
    }
  }
}
```

Neste MIDlet observe o comentário "Instruções para o Form". Este servirá como ponto de entrada para os próximos exemplos, que deverão serem inseridos exatamente a partir deste ponto.

7. ChoiceGroup

Um componente *ChoiceGroup* representa grupos de escolhas selecionadas. A escolha pode conter um texto, uma imagem ou ambas.

As escolhas podem ser EXCLUSIVE (somente uma opção pode ser selecionada) ou MULTIPLE (várias opções podem ser selecionadas de uma vez). Caso um objeto do tipo *ChoiceGroup* seja um tipo de *POPUP*, somente uma opção poderá ser selecionada. Uma seleção de *popup* será exibida quando este item for selecionado. Cabe ao usuário efetuar uma única escolha. A opção exibida é sempre a seleção escolhida.

O método *getSelectedIndex*() retorna o índice do elemento selecionado de um *ChoiceGroup*. O método *getSelectedFlags*() retorna um grupo de atributos do tipo boolean que corresponde ao estado de cada um dos elementos. O método *isSelected*(int index) retorna o estado de um elemento a partir da posição informada no atributo index.

```
// Insira estas instruções no Form
ChoiceGroup choiceExclusive = new ChoiceGroup("Exclusive", Choice.EXCLUSIVE);
choiceExclusive.append("Male", null);
choiceExclusive.append("Female", null);
append(choiceExclusive);

ChoiceGroup choiceMultiple = new ChoiceGroup("Multiple", Choice.MULTIPLE);
choiceMultiple.append("Apple", null);
choiceMultiple.append("Orange", null);
choiceMultiple.append("Grapes", null);
append(choiceMultiple);

ChoiceGroup choicePopup = new ChoiceGroup("Popup", Choice.POPUP);
choicePopup.append("Asia", null);
choicePopup.append("Europe", null);
choicePopup.append("Americas", null);
append(choicePopup);
```

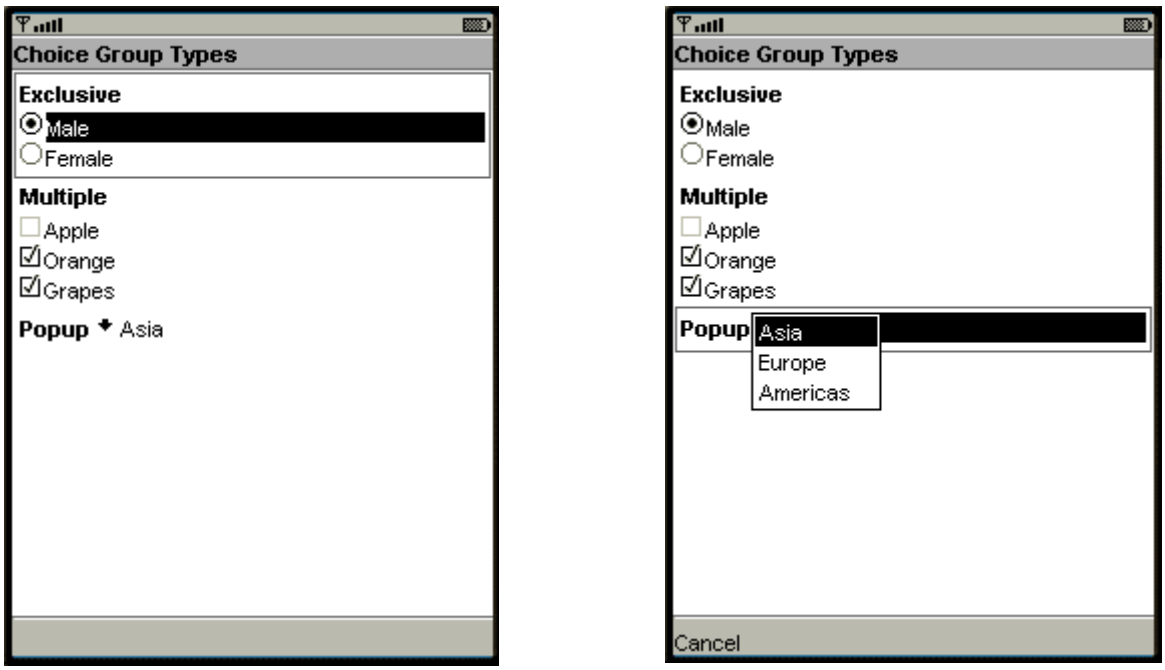


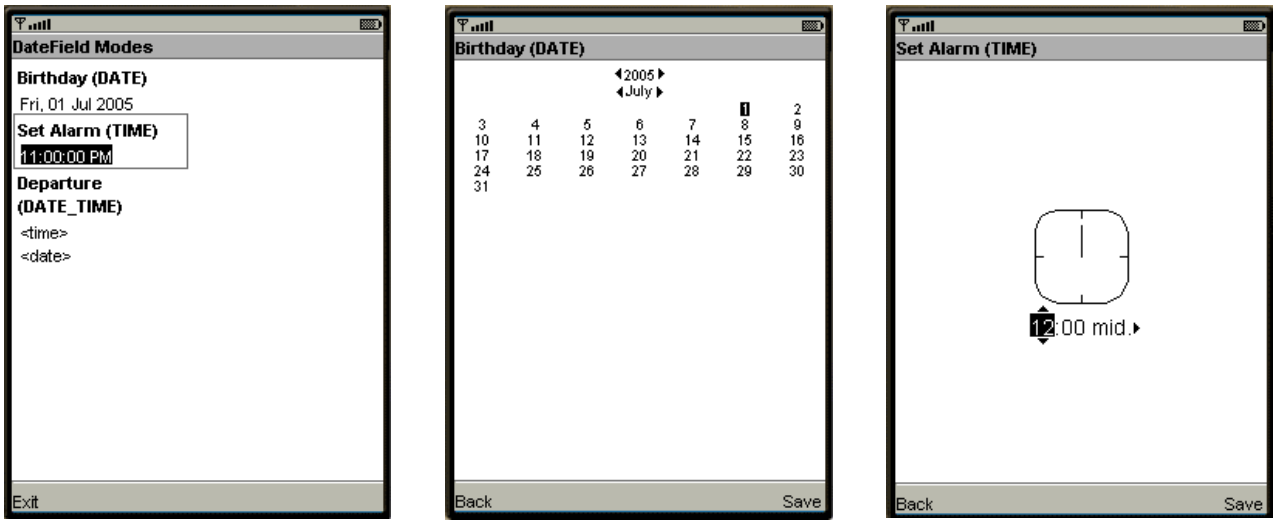
Figura 12: Modelos de Grupos de Escolha

8. DateField

O componente DateField é utilizado para as entradas de data e hora. Pode conter uma entrada de data (modo *DATE*), uma entrada de hora (modo *TIME*) ou ambas (modo *DATE_TIME*).

O método *getDate*() retorna o valor atual de um item e retornará *null* caso este item não seja inicializado. Caso o modo do *DataField* seja *DATE*, a hora do componente irá retornar zero. Se o modo for *TIME*, a data do componente é definido para "Janeiro 1, 1970".

```
// Insira estas instruções no Form
DateField dateonly = new DateField("Birthday (DATE)", DateField.DATE);
DateField timeonly = new DateField("Set Alarm (TIME)", DateField.TIME);
DateField datetime =
   new DateField("Departure (DATE_TIME)", DateField.DATE_TIME);

append(dateonly);
append(timeonly);
append(datetime);
```



Entrada de DateField ***Selecionando Datas*** ***Selecionando Horas***

Figura 13: Modelos de DateField e telas de entrada

9. StringItem

Um componente *StringItem* é um componente somente de leitura. Sendo composto de um *label* e um texto.

Um objeto do tipo *StringItem*, opcionalmente, permite um argumento que representa a aparência. O modo de aparência pode ser definido através das constantes Item.PLAIN, Item.HYPERLINK ou Item.BUTTON.

Caso o modo da aparência seja do tipo HYPERLINK ou BUTTON, o comando padrão e o *ItemCommandListener* precisam ser definidos no item.

```
// Insira estas instruções no Form
StringItem plain = new StringItem("Plain", "Plain Text", Item.PLAIN);
StringItem hyperlink =
   new StringItem("Hyperlink", "http://www.sun.com", Item.HYPERLINK);
hyperlink.setDefaultCommand(new Command("Set", Command.ITEM, 0));
StringItem button = new StringItem("Button", "Click me", Item.BUTTON);
button.setDefaultCommand(new Command("Set", Command.ITEM, 0));
append(plain);
append(hyperlink);
append(button);
```

Figura 14: StringItem

10. ImageItem

O componente ImageItem é uma imagem gráfica que pode ser colocada em um componente, tal como um *Form*. O objeto *ImageItem* permite um objeto do tipo *layout* como parâmetro (veja mais na seção sobre ***Item***):

```
public ImageItem(
  String label,
  Image img,
  int layout,
  String altText)
```

Outro construtor aceita uma modalidade de aparência, podendo ser um dos seguintes atributos definidos: *Item.PLAIN*, *Item.HYPERLINK* ou *Item.BUTTON* (ver mais na seção sobre ***StringItem***):

```
public ImageItem(String label,
  Image image,
  int layout,
  String altText,
  int appearanceMode)
```

O arquivo "jedi.png" é importado para o projeto usando-se a operação de gerenciamento de arquivos do sistema e inserido no diretório do projeto sobre o subdiretório "src". O projeto é atualizado pelo clique com botão direito do mouse sobre o nome do projeto e seleção de "*Refresh Folders*".

```
// Insira estas instruções no Form
try {
  Image img = Image.createImage("/jedi.png");
  ImageItem image = new ImageItem("JEDI", img, Item.LAYOUT_CENTER, "JEDI Logo");
  append(image);
} catch (Exception e){
  e.printStackTrace();
}
```

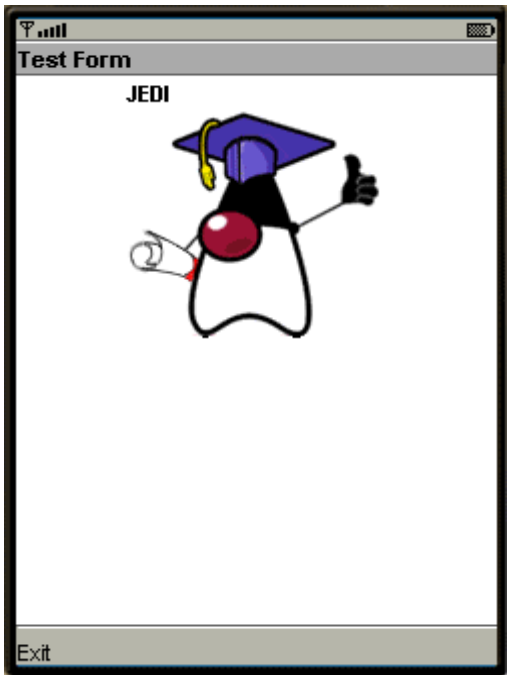


Figura 15: ImageItem

11. TextField

Um componente *TextField* é um item onde o usuário pode codificar a entrada. Diversas regras de entrada, mutuamente exclusivas, podem ser ajustadas:

```
TextField.ANY                    TextField.EMAILADDR
TextField.NUMERIC                TextField.PHONENUMBER
TextField.URL                    TextField.DECIMAL
```

A entrada pode também ter estes modificadores:

```
TextField.PASSWORD               TextField.UNEDITABLE
TextField.SENSITIVE              TextField.NON_PREDICTIVE
TextField.INITIAL_CAPS_WORD      TextField.INITIAL_CAPS_SENTENCE
```

Estes modificadores podem ser atribuídos usando-se o operador de comparação binária *OR* (|) (ou pelo uso do operador de comparação binária: *XOR* ^) sobre as regras de entrada. Consequentemente, modificadores podem ser derivados do valor do retorno do método *getConstraints()* ao operador de comparação binária *AND* (&).

Já que os modificadores são retornados pelo método *getConstraints()*, a entrada principal pode ser extraída pelo uso do operador de comparação binária com *TextBox.CONSTAINT_MASK* e o retorno do valor do método *getConstaints()*.

O método *getString()* retorna o conteúdo do *TextField* como um valor do tipo caractere.

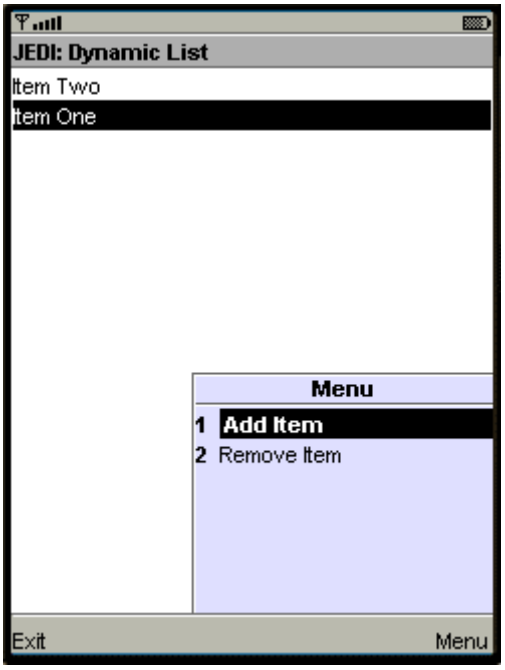
```
// Insira estas instruções no Form
TextField ANY = new TextField("ANY", "", 64,  TextField.ANY);
TextField EMAILADDR = new TextField("EMAILADDR", "", 64,  TextField.EMAILADDR);
TextField NUMERIC = new TextField("NUMERIC", "", 64,  TextField.NUMERIC);
TextField PHONENUMBER =
  new TextField("PHONENUMBER", "", 64,  TextField.PHONENUMBER);
TextField URL = new TextField("URL", "", 64,  TextField.URL);
TextField DECIMAL = new TextField("DECIMAL", "", 64,  TextField.DECIMAL);

append(ANY);
append(EMAILADDR);
append(NUMERIC);
append(PHONENUMBER);
append(URL);
append(DECIMAL);
```

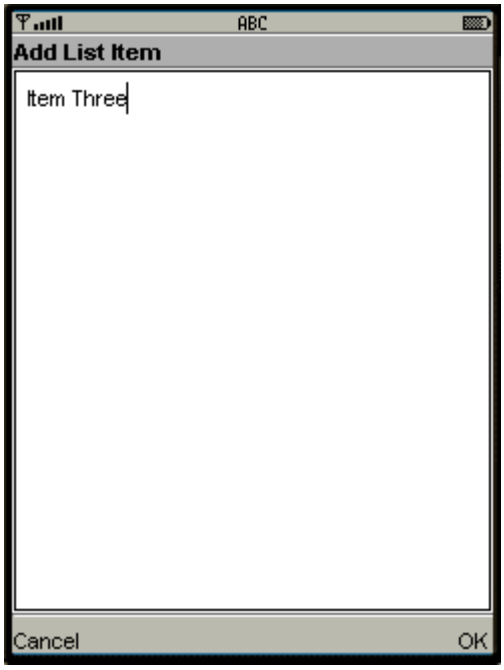
Figura 16: TextField Items

12. Exercícios

12.1. Lista dinâmica

Criar um *MIDlet* com uma *List* do tipo IMPLICIT na tela principal. Anexar três comandos para esta *List* – "*Add Item*", "*Remove Item*" e "*Exit*". O comando "*Add Item*" alertará o usuário para uma entrada a lista usando um *TextBox*, então adicionará este antes do item selecionado na lista. "*Remove Item*" removerá o item selecionado da lista (dica, veja o método *getSelectedIndex*). O Comando "*Exit*" finalizará o *Midlet*.





Parceiros que tornaram JEDI™ possível

Instituto CTS
Patrocinador do DFJUG.

Sun Microsystems
Fornecimento de servidor de dados para o armazenamento dos vídeo-aulas.

Java Research and Development Center da Universidade das Filipinas
Criador da Iniciativa JEDI™.

DFJUG
Detentor dos direitos do JEDI™ nos países de língua portuguesa.

Banco do Brasil
Disponibilização de seus *telecentros* para abrigar e difundir a Iniciativa JEDI™.

Politec
Suporte e apoio financeiro e logístico a todo o processo.

Borland
Apoio internacional para que possamos alcançar os outros países de língua portuguesa.

Instituto Gaudium/CNBB
Fornecimento da sua infra-estrutura de hardware de seus servidores para que os milhares de alunos possam acessar o material do curso simultaneamente.