Szkielet MIDletu

Szablon najprostrzego MIDletu zaprezentowaliśmy już w artykule "Profil MIDP 1.0":

```
public class MyFirstMIDlet extends MIDlet {

    public void startApp() throws MIDletStateChangeException {
    }

    public void pauseApp() {
    }

    public void destroyApp(boolean unconditional) {
    }

    /** Opcjonalny konstruktor:
     */
    public MyFirstMIDlet() {
    }

}
```

Pisałem, że główna klasa MIDletu musi rozszerzać abstrakcyjną klasę `MIDlet`. Omówiłem tam również znaczenie trzech metod, które zadeklarowane są w tej klasie, a które my musimy implementować.
Jednak powyższy szkielet trzeba wzbogacić o jakąś funkcjonalność. W tym artykule omówię więc jak wypisać prosty tekst na ekranie oraz jeden ze spsobów reagowania na zdarzenia klawiatury.

Klasa Display

Obiekt klasy `Display` jest logiczną reprezentacją ekranu. Każdy uruchomiony MIDlet ma dedykowany dokładnie jeden taki obiekt i może pozyskać referencję do niego poprzez wywołanie metody

```
public static Display getDisplay(MIDlet midlet)
```

Metoda ta zwróci oczywiście ten sam wskaźnik, niezależnie od miejsca, w którym zostanie wywołana (począwszy od metody `startApp()` - na `destroyApp()` skończywszy). Warto więc przechować tę referencję w zmiennej typu `Display`.
Na tym logicznym ekranie wyświetlić możemy instancje klas wywiedzionych z abstrakcyjnej `Displayable` (o niej za chwilę).
Taki odpowiednio spreparowany wygląd ekranu wyświetlamy za pomocą następującej metody klasy `Display`:

```
public void setCurrent(Displayable displayable)
```

Pobieramy natomiast aktualnie wyświetlany `Displayable` metodą:

```
public Displayable getCurrent()
```

Klasa Displayable

Wyświetlać na ekranie możemy obiekty klas wywiedzionych z Displayable. Oto diagram prezentujący chierarchię tych klas:

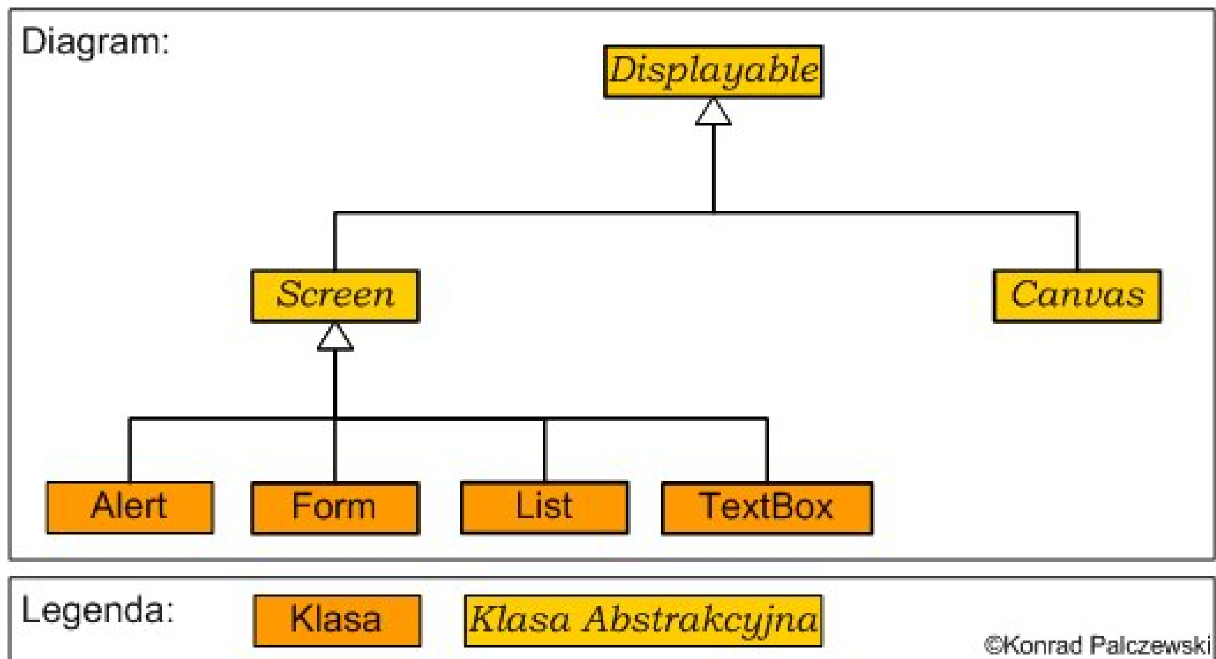


Diagram klas dziedziczących z Displayable.

Co ważne, już sam `Displayable` dostarcza funkcjolanalności, która oczywiście dostępna jest we wszystkich jego potomkach. Abstrakcyjna `Displayable` ma następujący interfejs:

```
public boolean isShown()
public void addCommand(Command cmd)
public void removeCommand(Command cmd)
public void setCommandListener(CommandListener l)
```

O znaczeniu tych metod powiem później.
Bezpośrednio z klasy `Displayable` wywiedzione są również abstrakcyjne klasy `Screen` i `Canvas`. Reprezentują one dwa sposoby tworzenia GUI naszej aplikacji.
I tak dzięki klasom wywiedzionym z klasy `Screen` możemy tworzyć *interfejs wysokiego poziomu* - wykorzystując do tego komunikaty, formularze, listy oraz pola tekstowe. Dają one bardzo duże możliwości. W łatwy sposób, bo z gotowych elementów, tworzyć możemy części naszej aplikacji (menu z ikonami, zaawansowane formularze, itp.), które dodatkowo swój wygląd przystosowują do urządzenia, na którym aplikacja jest uruchamiana. Znaczy to, iż wygląd elementów GUI wysokiego poziomu determinuje maszyna Javy konkretnego urządzenia.
Oznacza to jednak również to, iż wyglądu tych elementów nie jesteśmy w stanie kontrolować.

Totalną kontrolę nad wyglądem ekranu, który chcemy wyświetlić, daje nam za to klasa `Canvas`. To za jej pomocą możemy "malować" poszczególne elementy na ekranie, tworząc je z takich atomów jak linie, elipsy, prostokąty czy wyświetlając wczytane z pliku JAR obrazy.
Tworzone w ten sposób GUI nazywać będę *interfejsem nieskiego poziomu*, gdyż defiować tu będziemy wygląd ekranu na najniższym możliwym poziomie.
Niestety, te dwa sposoby nie potrafią ze sobą współgrać. Nie możemy więc wyświetlić elementu `TextBox` wewnątrz namalowanego "na niskim poziomie" obrazu.
W tym artykule skorzystamy z klasy `Canvas`. Omówię jej najważniejsze częsci i dodamy możliwą funkcjonalność do naszego MIDletu.

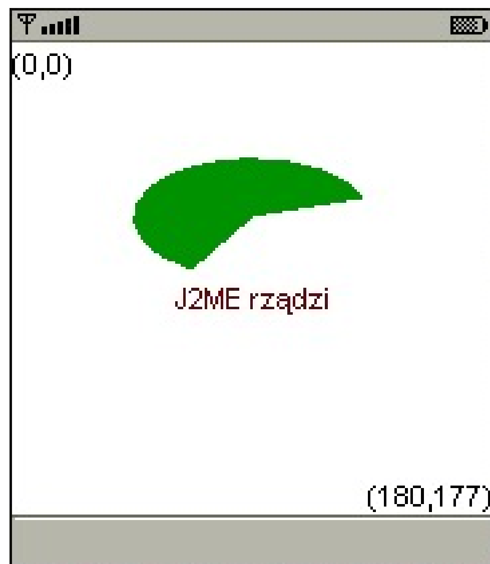
MyFirstMIDlet wyświetla pierwszy tekst

Wiedząc już wszystko, co zostało powyżej napisane, możemy wzbogacić nasz bezużyteczny szablon o - jak bardzo użyteczną - możliwość wyświetlenia tekstu i namalowania dowolnego kształtu na ekranie.
Aby tego dokonać, dodajemy klasę rozszerzającą klasę `Canvas` i wypełniamy jej metodę `paint()` definiując w niej wygląd ekranu.
Kod wygląda wówczas następująco:

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

/**
 * MIDlet prezentujący proste wypisywanie na ekranie
 * za pomocą GUI niskiego poziomu.
 * @author  Konrad Palczewski
 * @version 1.0
 */
public class MyFirstMIDlet extends MIDlet {

    //W tej zmiennej zapiszę obiekt pobrany metodą Display.getDisplay()
    private Display display;

    /** Wewnętrzna klasa Ekran rozszerzająca klasą Canvas
     */
    private class Ekran extends Canvas {
        String label = "Tekst domyślny";//tekst do wypisania na ekranie
        int elipsewidth = 90;           //szerokość elipsy na środku ekranu
        int ScreenWidth, ScreenHeight;  //szerokość i długość ekranu

        /** Konstruktor wewnętrznej klasy Ekran
         */
        public Ekran(String inparam){
          //
            label = inparam;
          //Pobieram wysokość i szerokość dostępnego ekranu:
            ScreenWidth = getWidth();
            ScreenHeight = getHeight();
        }

        /** W matodzie paint() definiujemy wygląd elementu
```

```
         */
        protected void paint(Graphics g){
          //Naalowanie tła:
            g.setColor(0xffffff);   //zmiana aktualnego koloru
                    //Namalowanie wypełnionego aktualnym kolorem prostokąta:
            g.fillRect(0, 0, ScreenWidth, ScreenHeight);
          //Współrzędne w rogach ekranu:
            g.setColor(0x000000);
            g.drawString("(0,0)", 0, 0, g.TOP|g.LEFT);
            g.drawString("(" + ScreenWidth + "," + ScreenHeight
                    + ")", ScreenWidth, ScreenHeight, g.BOTTOM|g.RIGHT);
          //Tekst no środku ekranu:
            g.setColor(0x770000);
            g.drawString(label, ScreenWidth/2, ScreenHeight/2,
                    g.HCENTER|g.TOP);
          //Fragment elipsy:
            g.setColor(0x009900);
            g.fillArc(ScreenWidth/2 - elipsewidth/2, ScreenHeight/2
                    - elipsewidth/2, elipsewidth, elipsewidth/2, 20, 220);
        }
    }


    /** Opcjonalny konstruktor
     */
    public MyFirstMIDlet() {
    }

    public void startApp() {
        display = Display.getDisplay(this);
        display.setCurrent(new Ekran("J2ME rządzi"));
    }

    public void pauseApp() {
    }

    public void destroyApp(boolean unconditional) {
    }

}
```

Widać, że zbudowałem klasę wewnętrzną `Ekran` dziedziczącą z `Canvas`. Zrobiłem tak dla zwięzłości kodu. Oczywiście definicja tej klasy może być w osobnym pliku. Naturalnie wówczas klasa musi być publiczna.
Jeśli uruchomimy zaprezentowany wyżej MIDlet, efekt powinien wyglądać tak:



Tak wygląda ekran po uruchomieniu naszego programu na tym etapie.

Klasa Command i interfejs CommandListener

Musimy jednak dodać jeszcze opcję zgrabnego zakończenia działania aplikacji. Służą do tego komendy udostępniane przez klasę `Command`, która posiada jeden konstruktor:

```
public Command(String label, int type, int priority)
```

Stworzona w ten sposób komenda posiada więc następujące atrybuty:

1. `label` - etykieta komendy wyświetlana na ekranie
2. `type` - typ komendy o wartości jednej z poniższych zdefiniowanych w klasie `Command` stałych::
   1. `OK`
   2. `BACK`
   3. `CANCEL`
   4. `STOP`
   5. `EXIT`
   6. `HELP`
   7. `SCREEN`
   8. `ITEM`

   Ich znaczenie odpowiada znaczeniu słowa je definiującemu.
3. `priority` - określa priorytet utworzonej komendy.

Znaczenie etykiety jest dosyć intuicyjne. Typ i priorytet pomagają z kolei maszynie konkretnego urządzenia zdecydować, które komendy będą dostępne bezpośrednio, a do których dostęp będzie wymagał np. rozwinięcia odpowiedniego menu. Jest to związane z ogromnym wachlarzem urządzeń, na których uruchomiane są MIDlety. Telefony na przyklad mają zazwyczaj dwa przyciski funkcyjne, więc tymi właśnie dwoma przyciskami będziemy musieli mieć dostęp do wszystkich dodanych do ekranu komend. Inaczej sytuacja wygląda w

przypadku palmtopów - oprócz dostępnych klawiszy funkcyjnych, mają one możliwość wyświetlania przycisków reagujących na dotknięcie rysikiem czy chociażby do rozwijanego menu.
O położeniu komend na ekranie decudują więc wspomniane dwa parametry. Z tym, że najpierw brany jest pod uwagę typ komendy, później jej priorytet (im mniejszy, tym komenda ważniejsza).
Utworzywszy komendę, nie możemy już zmienić żadnej z jej trzech przytoczonych parametrów. Możemy ją za to wyświetlić na ekranie. Pojedyncza instancja komendy może być dodana do wielu instancji `Displayable`. Dokonujemy tego korzystając z metody tej klasy już wspomnianej przy jej omawianiu:

```
public void addCommand(Command cmd)
```

Każdy `Displayable` ma również metodę pozwalającą zarejestrować `CommandListener`, który będzie powiadamiany o wszystkich aktywowanych komendach:

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

`CommandListener` jest interfejsem posiadającym tylko jedną metodę:

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

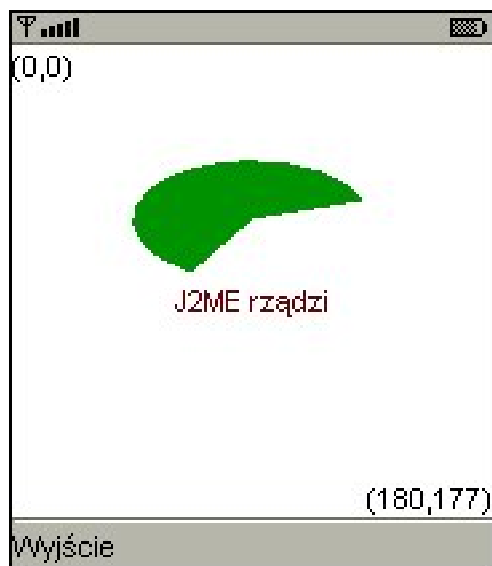
Jeśli więc chcemy operować komendami w naszej aplikacji, jedna z naszych klas musi implementować ten interfejs. Wówczas w metodzie `commandAction()` możemy rozpoznać wywołaną komendę i aktywny `Displayable`, i odpowiednio na nią zareagować.
W naszej aplikacji będzie to wyglądało następująco:

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

/**
 * MIDlet prezentujący proste wypisywanie na ekranie
 * za pomocą GUI niskiego poziomu.
 * @author  Konrad Palczewski
 * @version 1.0
 */
public class MyFirstMIDlet extends MIDlet {

    //W tej zmiennej zapiszę obiekt pobrany metodą Display.getDisplay()
    private Display display;

    /** Wewnętrzna klasa Ekran rozszerzająca klasą Canvas
     */
    private class Ekran extends Canvas implements CommandListener{
        String label = "Tekst domyślny";//tekst do wypisania na ekranie
        int elipsewidth = 90;           //szerokość elipsy na środku ekranu
        int ScreenWidth, ScreenHeight;  //szerokość i długość ekranu
        Command opcjawyjscie = new Command("Wyjście", Command.EXIT, 1);

        /** Konstruktor wewnętrznej klasy Ekran
         */
```

```
    public Ekran(String inparam){
      //
        label = inparam;
      //Pobieram wysokość i szerokość dostępnego ekranu:
        ScreenWidth = getWidth();
        ScreenHeight = getHeight();
      //Dodaję komendę:
        addCommand(opcjawyjscie);
        setCommandListener(this);
    }

    public void commandAction(Command c, Displayable d) {
        if (c == opcjawyjscie) {
            destroyApp(true);
            notifyDestroyed();
        }
    }

    /** W matodzie paint() definiujemy wygląd elementu
     */
    protected void paint(Graphics g){
      //Naalowanie tła:
        g.setColor(0xffffff);   //zmiana aktualnego koloru
                //Namalowanie wypełnionego aktualnym kolorem prostokąta:
        g.fillRect(0, 0, ScreenWidth, ScreenHeight);
      //Współrzędne w rogach ekranu:
        g.setColor(0x000000);
        g.drawString("(0,0)", 0, 0, g.TOP|g.LEFT);
        g.drawString("(" + ScreenWidth + "," + ScreenHeight
                + ")", ScreenWidth, ScreenHeight, g.BOTTOM|g.RIGHT);
      //Tekst no środku ekranu:
        g.setColor(0x770000);
        g.drawString(label, ScreenWidth/2, ScreenHeight/2,
                g.HCENTER|g.TOP);
      //Fragment elipsy:
        g.setColor(0x009900);
        g.fillArc(ScreenWidth/2 - elipsewidth/2, ScreenHeight/2
                - elipsewidth/2, elipsewidth, elipsewidth/2, 20, 220);
    }
}


/** Opcjonalny konstruktor
 */
public MyFirstMIDlet() {
}

public void startApp() {
    display = Display.getDisplay(this);
    display.setCurrent(new Ekran("J2ME rządzi"));
}

public void pauseApp() {
}

public void destroyApp(boolean unconditional) {
```

```
    }
}
```

Efektem wykonania powyższego kodu na emulatorze jest przedstawiony poniżej ekran:



MyFirstMIDlet uruchomiony na emulatorze.