

Wstęp

Poznaliśmy sposoby tworzenia GUI wysokiego poziomu. Klasy `Screen` oraz `Item` dają rzeczywiście imponujące możliwości przy bardzo niewielkiej ilości kodu. Jedynym problemem przy korzystaniu z tego sposobu tworzenia interfejsu użytkownika jest znikoma kontrola nad wyglądem elementów i ich ułożeniem na ekranie. Praktycznie całą pracę nad *look & feel* naszej aplikacji wykonuje maszyna wirtualna, której budowa zależy przecież od urządzenia, dla którego została zaprojektowana.

Zupełnie odwrotne cechy posiada przedstawiony poniżej sposób tworzenie interfejsu użytkownika. Dużo kodu, dużo pracy, efekty pożądane osiągnięte dopiero po długim czasie prób i sztukowania kolejnych pikseli i w końcu - mała przenośność w ten sposób wygenerowanego kodu. A jednak pełna kontrola nad wyglądem ekranu jest niezbędna przy tworzeniu przede wszystkim gier, zaawansowanych prezentacji, itp. Poniżej wprowadzę Was w świat pikselów, linii i prostokątów rodem z wszelkiej maści arkanoidów...

Klasa Canvas - wprowadzenie

Przypomnijmy jeszcze raz diagram klas wywiedzionych z abstrakcyjnej `Displayable`:

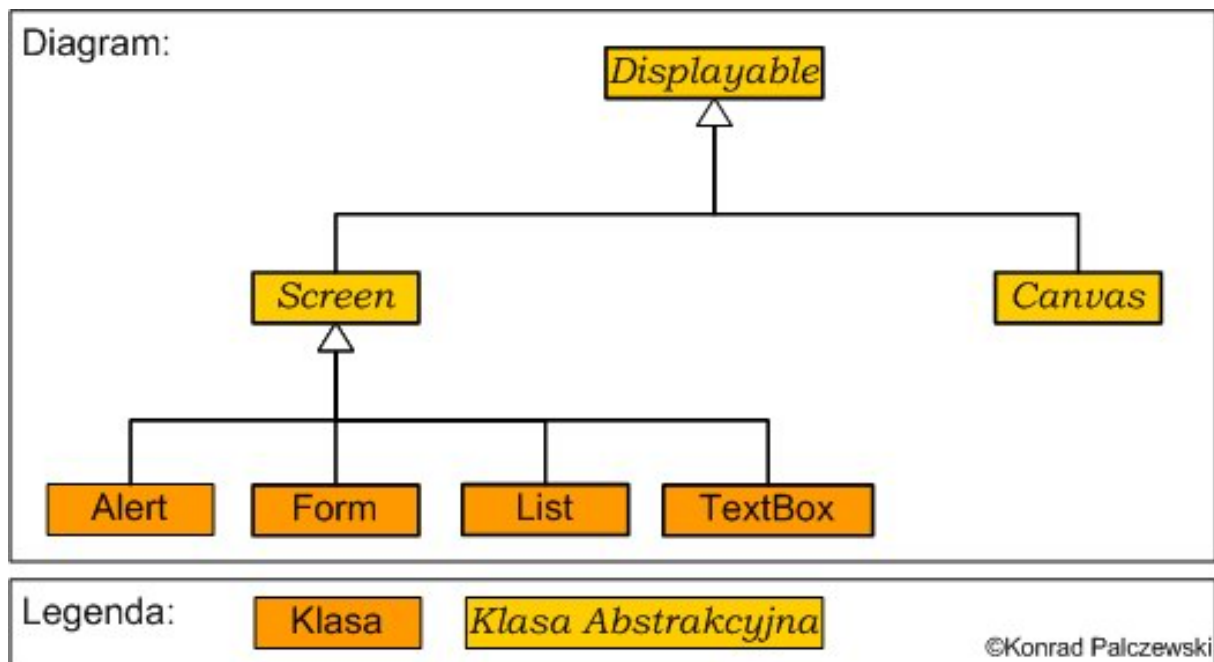


Diagram klas dziedziczących z `Displayable`.

Tym razem spojrzmy jednak na prawą stronę diagramu. Widać, że klasa `Canvas` wywiedziona z `Displayable`. A już `Displayable` daje możliwość dołączenia do niej komend. Będziemy więc mogli dołączać je do obiektów klas rozszerzających `Canvas`. Bo właśnie tak będziemy korzystać z klasy `Canvas` - będziemy definiować klasy wywodzące się z niej, wypełniając jedyną abstrakcyjną w niej metodę:

```
protected abstract void paint(Graphics g)
```

Korzystając z podanego nam jako argument (przez maszynę wirtualną) argumentu typu `Graphics`, będziemy w tej metodzie definiować wygląd naszego obiektu.

Oprócz w/w metody abstrakcyjnej, `Canvas` posiada wiele zdefiniowanych już metod, korzystając z których możemy zarządzać "odmalowania" obiektu (metody `repaint()`), przesyłając które możemy reagować na naciśnięcie lub zwolnienie przycisku klawiatury, przesunięcie rysika, lub chociażby sprawdzić właściwości ekranu.

Klasa `Graphics` - wprowadzenie

Jako argument najważniejszej metody klasy `Canvas` dostajemy obiekt klasy `Graphics`. Korzystając z jego metod możemy definiować wygląd naszego obiektu zupełnie tak, jakby był on kartką papieru o wielkości i ziarnistości odpowiadającej ekranowi naszego urządzenia.

Są więc w klasie `Graphics` metody do rysowania linii, prostokątów, wypisywania tekstu, malowania fragmentów łuków czy wstawiania gotowych obrazków i kilka ich mutacji.

Dla łatwości korzystania z tych metod warto wyobrazić sobie ekran jako jedną z ćwiartek układu współrzędnych. Tak samo było z obrazem ekranu przy pracy z grafiką w starym pascalu i jemu podobnych. Jest to dosyć intuicyjne odniesienie, w którym podstawowym kwantem skoku na osi jest jeden piksel.

Mało intuicyjne jest jednak - również tradycyjnie - położenie osi układu współrzędnych. Otóż domyślnie środek układu współrzędnych leży w lewym górnym rogu ekranu, oś odciętych (X) rośnie ku prawej stronie, a oś rzędnych (Y) ku dołowi. Takiej konwencji będę się trzymał prezentując przykłady w tych artykułach, ale umiejscowienie środka takiego układu współrzędnych można zmienić. Służy do tego metode `translate()`.

Pobieranie właściwości ekranu

Wiedza na temat możliwości wyświetlacza urządzenia, na którym uruchamiany jest MIDlet są oczywiście bardzo przydatne. W zależności od rozmiaru ekranu, możemy na przykład uzależnić wielkość elementów na nim wyświetlanych (skalowanie). Ale nie tylko. Przyda się na pewno wiedza o ilości wyświetlanych kolorów, obsłudze rysika i im podobnych.

Klasa `Canvas` zawiera metody, dzięki którym możemy sprawdzić powyższe cechy urządzenia. I tak na przykład poniższe metody pozwalają pozyskać wymiary ekranu (w pikselach) i sprawdzić, czy urządzenie dostarcza podwójne buforowanie:

```
public int getHeight()
public int getWidth()
public boolean isDoubleBuffered()
```

Jeśli urządzenie dostarcza podwójne buforowanie, to teoretycznie nie musimy się o nie martwić. Podwójne buforowanie polega na tym, iż zmiany w wyglądzie nie są widoczne od razu dla użytkownika, a przygotowywane są w buforze poza ekranem i periodycznie prezentowane na nim. Pomaga to uniknąć migotania ekranu.

Z kolei informację o liczbie wyświetlanych przez urządzenie kolorów możemy poznać dzięki metodom z klasy *Display*:

```
public boolean isColor()
public int numColors()
```

Klasa *Canvas* może również reagować na naciśnięcie klawiszy (opisane niżej). Niektóre urządzenia mogą jednak - oprócz zwykłego ich naciśnięcia rozpoznawać zdarzenie związane z jego dłuższym przytrzymaniem. Jest to tzw. "zdarzenie powtórzenia" (*repeat event*). Sprawdzić możliwość wystąpienia tego typu zdarzeń możemy dzięki metodzie w klasie *Canvas*:

```
public boolean hasRepeatEvents()
```

Oczywiście - jak napisałem wcześniej - klawisze klawiatury nie są jedynym możliwym dla MIDletów sposobem komunikacji w kierunku od użytkownika do aplikacji. W przypadku bardziej zaawansowanych urządzeń typu PDA, będzie to również rysik - jego zetknięcia z ekranem i ruch po nim. Poniższe dwie metody pozwalają sprawdzić dostępność tego typu zdarzeń:

```
public boolean hasPointerEvents()
public boolean hasPointerMotionEvents()
```

Reagowanie na zdarzenia z klawiatury

Aby móc w odpowiedni sposób reagować na zdarzenia związane z klawiszami urządzenia, przysyłamy odpowiednią z trzech metod:

```
protected void keyPressed(int keyCode)
protected void keyReleased(int keyCode)
protected void keyRepeated(int keyCode)
```

Odpowiednio: przysyłając pierwszą z nich, będziemy obsługiwać naciśnięcie klawiszy, drugiego - ich zwolnienie, a trzeciego - przedłużone jego naciśnięcie.

Na wejściu do tych metod dostajemy kod przycisku, który został naciśnięty. Korzystając z tej informacji możemy zareagować odpowiednio na każdy klawisz urządzenia. Oczywiście duże znaczenie w MIDletach ma przenośność aplikacji. Dlatego w klasie *Canvas* zadeklarowane zostały kody przycisków, których obecność urządzenia muszą przynajmniej emulować.

Przypominając, są to klawisze o numerach od 0 do 9 oraz klawisze gwiazdki(*) i hash(#). Stałe te mają postać:

1. `KEY_NUMn` - gdzie *n*, to oznaczenie kolejnych przycisków (od 0 do 9),
2. `KEY_STAR` - klawisz oznaczający klawisz ze znacznikiem gwiazdki,
3. `KEY_POUND` - klawisz ze znacznikiem hash.

Chcąc więc na dowolnym urządzeniu reagować na przyciśnięcie i zwolnienie klawisza z gwiazdką, musimy przysłonić w naszej klasie `Canvas` metodę `keyPressed()` i `keyReleased()`, i rozpoznać odpowiedni kod przycisku:

```
protected void keyPressed(int keyCode) {
    if (keyCode == KEY_STAR) {
        //Naciśnięto klawisz gwiazdki...
    }
}

protected void keyReleased(int keyCode) {
    if (keyCode == KEY_STAR) {
        //Zwolniono klawisz gwiazdki...
    }
}
```