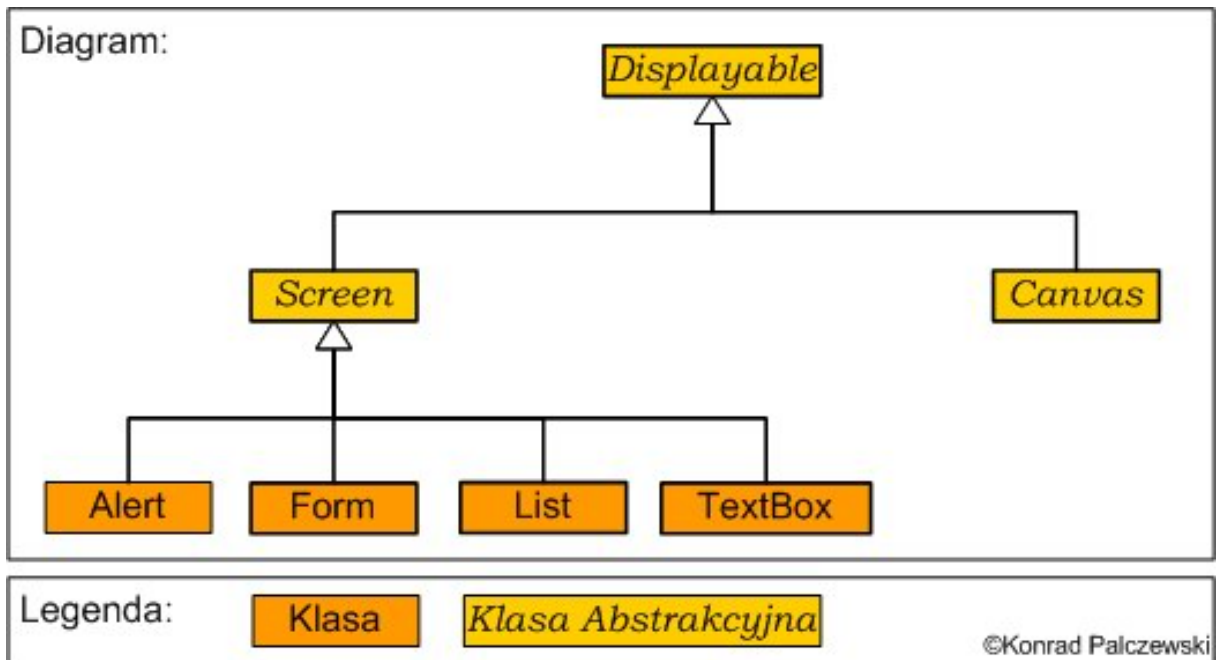


## Podstawy - klasa Screen.

Jak napisałem w artykule "Pierwszy MIDlet", wyświetlić na ekranie możemy tylko obiekty klas wywiedzionych z klasy `Displayable`. Podałem również diagram i ogólną charakterystykę klas wywiedzonych z abstrakcyjnej `Displayable`. Warto więc przed przeczytaniem tego artykułu przestudiować "Pierwszy MIDlet". Przy rozwijaniu naszej wiedzy na temat GUI wysokiego poziomu przydatny nam będzie jednak przytoczony w wymienionym artykule diagram:



*Diagram klas dziedziczących z Displayable.*

Pisałem również, że to klasy dziedziczące z klasy `Screen` pozwalają nam tworzyć GUI wysokiego poziomu. Teraz przybliżę zatem troszkę samą klasę `Screen`, abyśmy mogli przekonać się, jaki wspólny interfejs posiadają dziedziczące z niej klasy.

Otóż API tej klasy wygląda następująco:

```

public Ticker Ticker();
public String getTitle();
public void setTicker(Ticker ticker);
public void setTitle(String title);
  
```

Widzimy zatem, iż klasa `Screen` dodaje możliwość wyświetlania nagłówka (ang. *title*) do każdego elementu, który z niej dziedziczy. Można go zmienić w każdym momencie działania aplikacji za pomocą metody `setTitle(String title)`. `Ticker` z kolei jest ciągiem znaków, który jest przewijany w dostępnym dla niego miejscu przez cały czas wyświetlania elementu `Screen`.

Należy zaznaczyć, iż w specyfikacji profilu MIDP 2.0, *wszystkie* metody przeszły z klasy `Screen` do klasy `Displayable`. Jest to oczywiście zgodne z zasadą "zgodności wstecz", czyli

aplikacje napisane pod MIDP 1.0 zostaną bez problemów uruchomione na urządzeniu implementującym profil MIDP 2.0. Ma jednak inną konsekwencję. Mianowicie i nagłówki i "ticker'y" będą mogły być wyświetlane również przy korzystaniu z interfejsu niskiego poziomu, co jest konsekwencją ich dostępności w klasie `Canvas`.

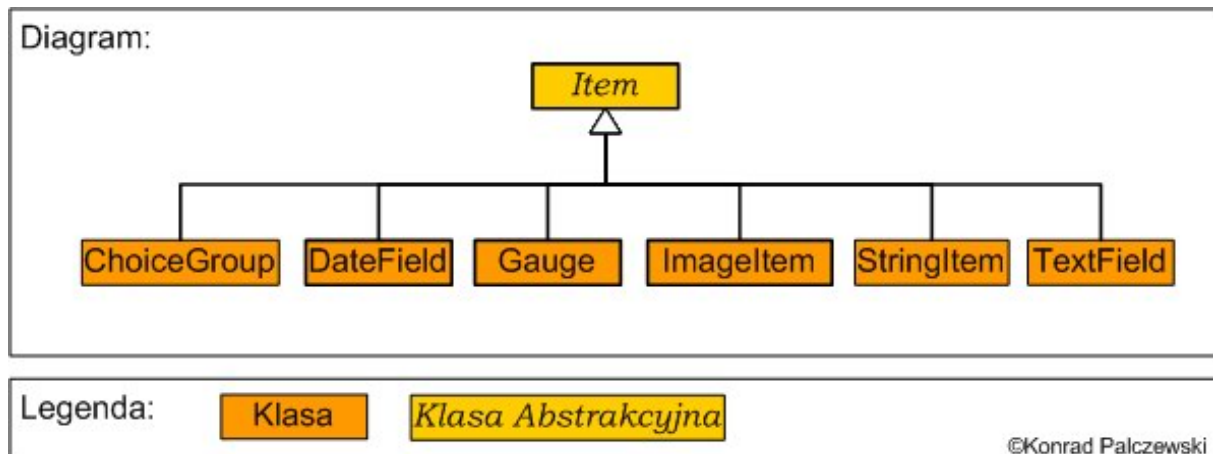
Aby utworzyć obiekt klasy `Ticker` korzystamy z jedyne w niej konstruktora:

```
public Ticker(String str);
```

W kolejnych sekcjach artykułu przybliżę klasy, z których budować możemy GUI wysokiego poziomu.

### Tworzenie formularzy - klasa `Form`

`Form` jest klasą wywiedzioną z klasy `Screen`, a umożliwiającą w bardzo prosty sposób tworzyć formularze z pojedynczych elementów. Wszystkie te "klocki" wywiedzione są z klasy `Item` pakietu `javax.microedition.lcdui`. Poniższy diagram prezentuje klasy wywiedzione z `Item`:



*Diagram klas dziedziczących z `Item`.*

Dokładniejszy opis tych klas znajdziesz w kolejnej sekcji tego artykułu. Ich wspólną cechą będzie posiadanie etykiety. Cała funkcjonalność klasy `Item` wygląda bowiem następująco:

```
public String getLabel();
public void setLabel(String label);
```

Formularz tworzyć możemy na dwa sposoby. Pierwszy polega na stworzeniu pustego obiektu klasy `Form`, posiadającego jedynie nagłówek i dodawaniu do niego kolejnych elementów. Pierwszy konstruktor wygląda więc następująco:

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

Dodawanie kolejne elementy możemy na przykład za pomocą metody:

```
public void append(Item item);
```

Możemy również przygotować zestaw elementów składających się na nasz formularz i skorzystać z poniższego konstruktora klasy `Form`:

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

Możemy również kontrolować zawartość formularza dynamicznie podczas działania aplikacji. Oprócz przywołanej wyżej `append()`, przydatne są do tego poniższe metody. Znaczenie większości jest intuicyjne.

```
public void delete(int index);
public Item get(int index);
public void insert(int index, Item item);
public void set(int index, Item item);
public int size();
```

Większość powyższych metod korzysta z indeksu elementu do działania na nim. Z kolei metoda `size()` zwraca ilość elementów w formularzu.

Warto wspomnieć, iż - w przeciwieństwie do komend - elementy `Item` mogą być dodane do jednego formularza!

Do formularza można oczywiście dodawać komendy (obiekty klasy `Command`), a ich uaktywnienie nasłuchiwać w zarejestrowanym `CommandListener`.

Czasami jednak pewne akcje chcemy podjąć nie po zaakceptowaniu formularza, a po zmianie stanu jednego z jego elementów. Możemy na przykład chcieć modyfikować zawartość formularza w zależności od dokonywanych w nim wyborów.

Zmiany w stanie elementów składających się na formularz możemy obserwować rejestrując w nim `ItemStateListener`, czyli klasę implementującą ten interfejs. Wskazujemy go w metodzie:

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

Wspomniany interfejs posiada pojedynczą metodę:

```
public void ItemStateChanged(Item item);
```

Implementując ją, możemy reagować na zdarzenia dotyczące elementów nas interesujących, gdyż to referencje do nich są nam przekazywane jako argument tej metody.

Sprawdzanie stanu pojedynczego obiektu klasy `Item` omówię w kolejnej sekcji artykułu omawiając różne typy obiektów `Item`.

## Składowe formularza - obiekty `Item`

### 1. `StringItem`

Najprostrzy z elementów budujących formularze w MIDP. Posiada jeden konstruktor:

```
StringItem(String label, String text)
```

Oczywiście atrybut *label* i akcje z nim związane odziedziczone są z klasy `Item`. Klasa `StringItem` dodaje do odziedziczonego interfejsu dwie metody:

```
public String getText()
public void setText(String text)
```

Oba pola są opcjonalne. Można więc - abstrahując od sensowności takiego działania - stworzyć obiekt, w którym oba pola będą miały wartość `null`.

Oczywiście pola *label* oraz *text* są rozróżniane i powinny być wyświetlane przez urządzenie w różny sposób. Dodatkowo, powinny być powiązane w taki sposób, iż przewijając ekran, nie możnaby zobaczyć treści elementu bez etykiety (jeśli jest dodana) i odwrotnie, czyli element powinien pojawiać się w całości.

Jak więc dodać element typu `StringItem` do formularza. Można użyć wspomnianego konstruktora klasy `StringItem` i dodać go do formularza przy pomocy przywołanej wcześniej metody `append()` z klasy `Form`. Na przykład:

```
Form form = new Form("Nagłówek");
form.append(new StringItem("Etykieta",
    "Treść elementu. +Możemy przejść do \n Nowej linii"));
```

Jest jeszcze możliwość skrócenia kodu, dzięki wersji metody `append()`, o której jeszcze nie mówiłem. Dwa poniższe zapisy są więc tożsame:

```
Form form = new Form("Nagłówek");
form.append("Treść elementu.");
```

```
Form form = new Form("Nagłówek");
form.append(new StringItem(null, "Treść elementu."));
```

Przykład zastosowania `StringItem` zademonstruję przy okazji prezentowania różnych rodzajów `TextField` w akapitach poniżej.

## 2. `TextField`

`TextField` jest elementem formularza umożliwiającym użytkownikowi wprowadzanie tekstu. Poprzednio omawiany element `Item`: `StringItem` nie umożliwiał użytkownikowi żadnej interakcji. Pole `TextField`, dzięki temu, że umożliwia zmianę swojej zawartości użytkownikowi będzie - w przeciwieństwie do `StringItem` - zgłaszać swoje zdarzenia do przypiętego do formularza `ItemStateListener'a`.

Ale pokolei. Konstruktor klasy `TextField` jest następujący:

```
public TextField(String label, String text, int maxSize, int constraints)
```

Element odziedziczony z klasy `Item` (`label`) już znamy. Omówię więc pozostałe argumenty konstruktora.

1. `text` - ciąg znaków którym pole tekstowe zostanie wypełnione przy jego inicjalizacji. Czyli początkowa jego zawartość.
2. `maxSize` - maksymalna liczba znaków wpisanych w pole tekstowe. Wartość ta może być ustawiona również w trakcie działania aplikacji dzięki metodzie `setMaxSize(int maxSize)`.
3. `constraints` - ograniczenia nałożone na dane wprowadzane do pola tekstowego. Argument ten przyjmuje wartość jednej ze stałych zdefiniowanych w klasie `TextField`. Oto te stałe wraz z ich znaczeniem:
  1. `TextField.ANY` - użytkownik może wprowadzić dowolną treść.
  2. `TextField.EMAILADDR` - użytkownik może wprowadzić adres e-mail. Specyfikacja nie wprowadza ograniczeń na tego typu pole. Producenci mogą wprowadzić swoją implementację tego ograniczenia.
  3. `TextField.NUMERIC` - ogranicza wprowadzane znaki do liczb. Pierwszy znak może być znakiem minus.
  4. `TextField.PASSWORD` - Oznacza, iż pole ma przechowywać hasło. Wiąże się to oczywiście ze specjalnym jego formatowaniem. Można tą stałą zsumować logicznie z jedną z dwóch innych stałych: `TextField.ANY` lub `TextField.NUMERIC`, np: `TextField.PASSWORD | TextField.NUMERIC` oznacza, iż wprowadzone dane będą mogły być tylko cyframi, a formatowane będą jak hasło.
  5. `TextField.PHONENUMBER` - ograniczenie wprowadzanej treści do numeru telefonu. Ta stała oznacza również specjalne formatowanie wprowadzonego numeru.
  6. `TextField.URL` - Powinno to oznaczać, że wprowadzony może być jedynie adres URL, to w specyfikacji nie ma to żadnego wpływu na możliwy do wprowadzenia ciąg znaków.

Ograniczenia na pola tekstowe możemy również zakładać po zainicjowaniu obiektu. Służy do tego metoda:

```
public void setConstraints(int constraints)
```

Jeśli formularz zawierający `TextField` ma zarejestrowany `ItemStateListener`, to zmiany wprowadzone w polu edycji będą implikowały wywołanie metody `itemStateChanged(Item item)`. W reakcji na jej wywołanie możemy chcieć pobrać zawartość pola edycji, bądź zmienić tę zawartość. Pozwalają nam na to poniższe metody:

```
public void delete(int offset, int length)
public int getChars(char[] data)
public String getString()
public void insert(char[] data, int offset, int length, int position)
public void insert(String src, int position)
public void setChars(char[] data, int offset, int length)
public void setString(String text)
```

Podkreślenia warty jest fakt, iż każda programowa zmiana w polu `TextField` za pomocą powyższych metod nie wywołuje metody `itemStateChanged(Item item)` klasy zarejestrowanej jako `ItemStateListener`.

Poniższy kod prezentuje formularz z różnymi typami pól `StringItem` i `TextField`. Cały formularz tworzony jest tu w konstruktorze, a jego wyświetlenie następuje w metodzie `startApp()`.

Do dodania dwóch elementów `StringItem` użyłem różnych wersji metody `append()`. Dalej dodaję już pola `TextField`, ale każdy z założony różnym ograniczeniem.

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

/**
 * MIDlet ma zaprezentować różne rodzaje pól TextField i StringItem.
 * @author Konrad Palczewski
 */
public class TextFieldTryout
extends MIDlet implements CommandListener {

    private Command exitCommand =
        new Command("Wyjście", Command.SCREEN, 2);
    private Display display;
    private Form form;

    public TextFieldTryout() {
        display = Display.getDisplay(this);

        form = new Form("Dane osobowe");

        //Dodaję różne StringItem:
        form.append(new StringItem("Etykieta", "Treść z etykieta."));
        form.append("I treść bez etykiety:\nNowa linia.");

        //Dodaję różne TextField:
        form.append(new TextField("Nick:", "Konrad", 30, TextField.ANY));
        form.append(new TextField("Hasło:", "mojehaslo",
            12, TextField.PASSWORD|TextField.ANY));
        form.append(new TextField("Adres e-mail:",
            "moj@host.pl", 30, TextField.EMAILADDR));
        form.append(new TextField("Numer telefonu:",
            "+48601234567", 12, TextField.PHONENUMBER));
        form.append(new TextField("Wiek:", null,
            3, TextField.NUMERIC));

        //Dodaję komendę i ustwiam klasę nasłuchującą jej aktywacji
        form.addCommand(exitCommand);
        form.setCommandListener(this);
    }

    public void startApp() {
        display.setCurrent(form);
    }
}
```

```

public void pauseApp() {}

public void destroyApp(boolean unconditional) {}

public void commandAction(Command c, Displayable s) {
    if (c == exitCommand) {
        destroyApp(false);
        notifyDestroyed();
    }
}
}

```

Jak w ten sposób zdefiniowany formularz wygląda na naszym emulatorze, prezentują screeny:

--screenshot--

TABELA1:

*Pierwszy screen...*

*...i drugi*

### 3. ChoiceGroup

ChoiceGroup - kolejny element Formularza Form - umożliwia dodanie kilku pól, z których - w zależności od typu pola ChoiceGroup - będziemy mogli wybrać kilka lub wiele możliwości. To rozróżnienie pomiędzy typami pól następuje już przy ich tworzeniu.

Oto dwa konstruktory:

```

public ChoiceGroup(String label, int choiceType);
public ChoiceGroup(String label, int choiceType,
    String[] strings, Image[] images);

```

Typ grupy elementów określamy w drugim parametrze konstruktora, podając jedną z dwóch stałych zdefiniowanych w interfejsie Choice, implementowanym m.in. przez

ChoiceGroup:

1. `Choice.EXCLUSIVE` - jedynie jeden element z całej grupy może być wybrany. Najczęściej wizualnie są te elementy przedstawiane jako zbiór HTML-owych "radio buttons".
2. `Choice.MULTIPLE` - ilość zaznaczonych elementów jest dowolna. Najczęściej przy elementach takiej grupy znaleźć można tzw. "checkbox".

Przedstawiłem dwa możliwe do wykorzystania konstruktory. Jeśli skorzystamy z pierwszego, stworzymy pusty obiekt `ChoiceGroup`. Aby dodać elementy, z których użytkownik może wybierać, korzystamy z poniższej metody (zadeklarowanej już w interfejsie `Choice`):

```
int append(String stringPart, Image imagePart);
```

Jako pierwszy jej argument, konieczny - nie może mieć wartości `null`, podajemy ciąg znaków, wyświetlany na ekranie. W drugim podajemy ikonę, która powinna być wyświetlana z tym elementem. Ten argument nie jest konieczny.

Metoda zwraca indeks dodanego elementu (zostanie on dodany na koniec grupy), który później będziemy używać do identyfikowania elementu przy jego ewentualnej modyfikacji. Jeśli natomiast od początku znamy elementy, które składać się będą na naszą grupę, to możemy stworzyć z nich tablicę i skorzystać z drugiego konstruktora. Są tutaj jednak następujące restrykcje: argument `strings` nie może być `null` i nie może zawierać elementów `null` oraz tablice `strings` i `images` muszą mieć taki sam rozmiar.

Gdybyśmy natomiast chcieli obserwować zmiany w stanie elementów naszego `choiceGroup` w dołączonym do formularza `ItemStateListener`, wówczas najpewniej będziemy chcieli sprawdzić stan elementów należących do grupy. Przydatne okażą się wówczas następujące metody (również z interfejsu `Choice`):

```
public int size();
public boolean isSelected(int index);
public int getSelectedIndex();
public int getSelectedFlags(boolean[] flags);
```

Metoda `size()` zwraca ilość elementów dołączonych do grupy, metoda `isSelected()` mówi, czy element o podanym indeksie jest zaznaczony. Skolei metoda `getSelectedIndex()` zwraca indeks wybranego elementu. Dla obiektów `ChoiceGroup` typu `MULTIPLE` metoda ta zawsze zwróci wartość `-1`.

Metoda `getSelectedFlags()` wypełnia tablicę flag podaną jako argument wartościami `true`, jeśli są zaznaczone, lub `false` - w przeciwnym wypadku (podana tablica musi mieć długość przynajmniej równą wielkości zwracanej przez metodę `size()`!). Metoda zwraca ilość wybranych elementów w grupie.

Jeśli natomiast chcielibyśmy zmienić zawartość naszego elementu `ChoiceGroup`, skorzystamy (oprócz metody `append()`) z poniższych metod. Ich znaczenie jest intuicyjne:

```
public void insert(int index, String string, Image image);  
public void set(int index, String string, Image image);  
public void delete(int index);
```

Przykład zastosowania elementów formularza typu `ChoiceGroup` znajdziesz w drugiej części tego artykułu: "Tworzenie GUI wysokiego poziomu - cz.2" w sekcji dotyczącej klasy `List`.

4. `DateField`  
todo
5. `Gauge`  
todo
6. `ImageItem`  
todo