Wstęp do programowania

Aby zacząć pisać programy, należało by najpierw zrozumieć jak one działają.
Rozważmy taką sytuację:
Spotykamy turystę, który prosi nas byśmy mu wytłumaczyli jak dojść na pocztę?
A no, powiedzielibyśmy pewnie coś w stylu: "Prosto, za bankiem w lewo, a potem..."
No tak, ale jak by to wytłumaczyć gdyby nasz turysta był niewidomy?
Więc możemy spróbować tak: "Proszę się kierować 20 metrów prosto, potem skręcić w lewo o kąt X, a potem ..., i tak dalej.
Oczywiście w prawdziwym życiu nasz turysta w optymistycznym wariancie zaraz by zapomniał jak miał iść, a w pesymistycznym rozkwasił by sobie nos o jakiś budynek, lub co gorsza wpadł by pod samochód niczego nieświadomej kobiety poprawiającej sobie właśnie makijaż(bez aluzji ;)).
Ale dosyć już tych makabresek!
Rozważmy dlaczego by nam się to nie udało:

1. Turysta ma ograniczoną pamięć.
2. My nie jesteśmy w stanie z miejsca podać współrzędnych szukanego obiektu.
3. Nasz turysta nie jest w stanie precyzyjnie odmierzyć długości i kąta skrętu.

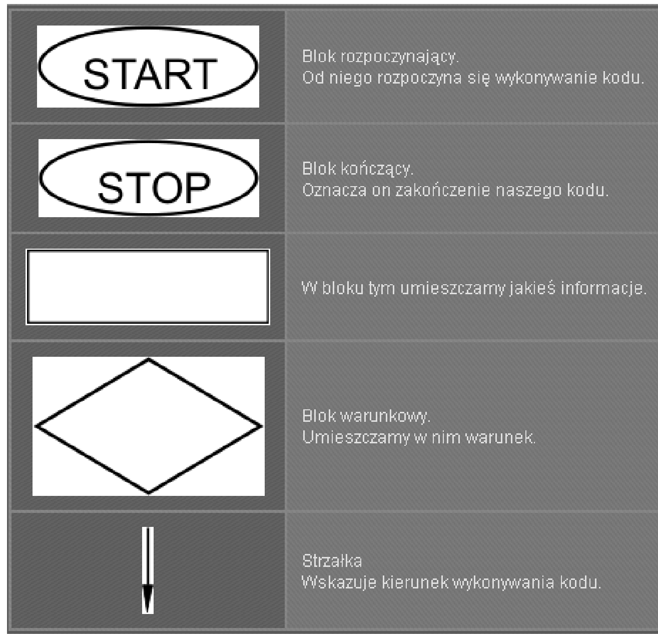
Ale dlaczego o tym piszę, a no załóżmy że ten turysta to nasz komputer, a my pokazujemy mu drogę przez napisany przez nas kod.
Rozważmy teraz ponownie powody naszych niepowodzeń:

1. Turysta ma ograniczoną pamięć -> komputer nie! (w sumie to ma :), ale nie przez to że zapomina, tylko po prostu czasem tej pamięci brakuje)
2. My nie jesteśmy w stanie z miejsca podać współrzędnych szukanego obiektu -> zakładając że mamy te współrzędne zapisane na twardym dysku, możemy je bez problemu wczytać.
3. Nasz turysta nie jest w stanie precyzyjnie odmierzyć długości i kąta skrętu -> w komputerach najważniejsza jest precyzja i perfekcja, bo nigdy nie jest tak że komputer zrobił błąd (wyjątkiem są tu wady sprzętowe), to po prostu programista źle napisał kod.

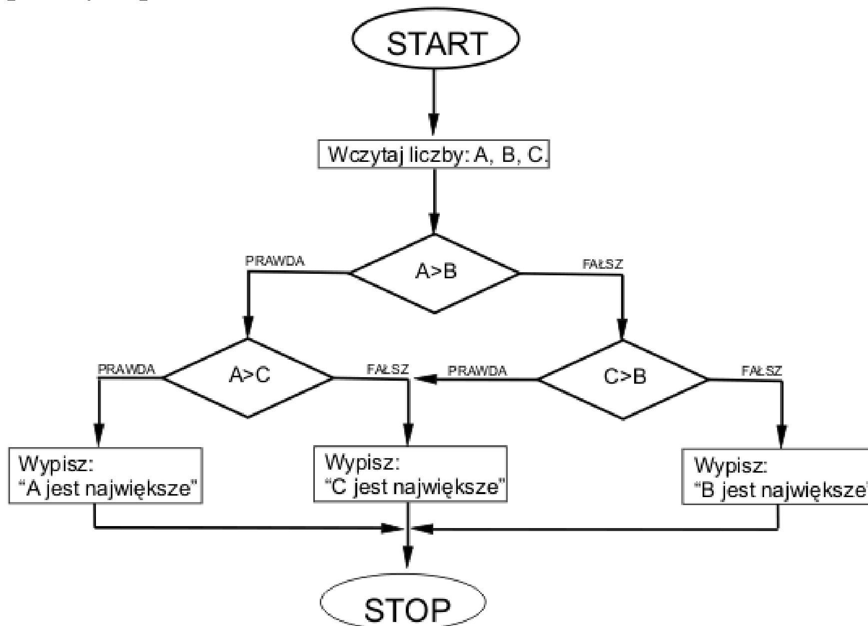
Co dociekliwsi zastanawiają się pewnie dlaczego pisania programów nie można porównać do widzącego turysty.
A no, bo gdyby kompilator program zamieniający nasz kod na gotową aplikacje zobaczył: "Prosto, za bankiem w lewo, a potem..."
Znając życie wygenerował by zapewne coś takiego:

> Nie znaleziono obiektu typu 'bank'
> Nieprawidłowe wykonanie operacji lewo - nie podano żadnego argumentu (oczywiście chodzi o kąt skrętu)
> I wiele innych błędów.

Dlatego też, powiedzenie że pisanie programów jest jak tłumaczenie ślepemu turyście jak dojść na pocztę(jak bardzo absurdalnie by to nie brzmiało), jest o wiele trafniejsze.
Tym oto może niezbyt najmądrzejszym przykładem, pokazałem wam sposób 'myślenia' komputera.
Kod można rozrysować tzw. schematem blokowym. Oto jego najważniejsze elementy:



Schemat blokowy doskonale nadaje się do pokazania o co chodzi w programowaniu.
Teraz czas narysować jakiś prosty przykład.
Niech będzie, że narysujemy schemat który będzie znajdował największą liczbę z trzech podanych przez nas.



Java jest prawie całkowicie obiektowym językiem programowania, tzn. wszystko jest obiektem, a wyjątkiem w tej kwestii są tylko zmienne.
Został on stworzony przez firmę Sun Microsystems w roku 1991 (początkowo język nosił nazwę Oak).
Java jest językiem kompilowanym do postaci wykonywanej przez maszynę wirtualną czyli tzw. kodu bajtowego, co bardzo ułatwia jej przenośność.
Składnia języka Java jest podobna do C++, co skraca czas jej nauki.

Główne cechy języka java to:
- prostota
- bezpieczeństwo
- przenośność
- solidność
- wielowątkowość
- neutralność architektury
- interpretowalność
- wysoka wydajność
- rozproszenie
- dynamika

Najnowsza wersja tego języka (J2SE 5), wniosła wiele nowości:
- typy sparametryzowane
- metadane
- automatyczne otaczanie i wydobywanie typów prostych
- wyliczenia
- zmienna liczba argumentów
- pętla for-each
- import statyczny
- formatowane wejście-wyjście
- unowocześnienie interfejsu programistycznego
- narzędzia związane ze współbieżnością

Składnia Javy, jest bardzo zbliżona do języka C++. Występują w niej m.in. te same nazwy zmiennych, operatory, pętle. Podobna jest także składnia klas, funkcji (w javie nazywanych metodami). W javie również jest dziedziczenie klas, występują też przeciążenia metod. Java składa się z wbudowanych pakietów(coś podobnego do .NET), to za ich pomocą możemy pisać aplety i programy.

Zmienne

Zmienne to podstawa w programowaniu.
Przechowują one określone dane (np. liczby, albo znaki).

Nazwa	Rozmiar [w bitach]	Zakres
byte	8	od -128 do 127
short	16	od -32 768 do 32 767
int	32	od -2 147 483 648 do 2 147 483 647
long	64	od -9 223 372 036 854 775 808 do 9 223 372 036 854 775 807
float	32	od 1.4e-045 do 3.4e+038
double	64	od 4.9e-324 do 1.8e+308
char	16	znaki Unicode
boolean	1	true/false

int a	//deklaracja

a = 65	//inicjalizacja
char b = 'b'	//deklaracja i inicjalizacja

Komentarze

Komentarze, jak sama nazwa wskazuje, służą do opisywania poszczególnych fragmentów kodu. Podczas kompilacji są one pomijane.
Komentarze w Javie można podzielić na jedno i wieloliniowe:

```
//komentarz jednoliniowy

/* Wieloliniowe
komentarze */
```

Tablice

Tablice to inaczej zbiory zmiennych.
W Javie możemy tworzyć tablice zmiennych i tablice klas.
Tablice zaczynają się od elementu zerowego.

```
int dni[ ] = new int[31];                    /*deklaracja i inicjalizacja 31 elementowej tablicy
typu int*/
int miesiace_i_dni[ ][ ] = new int[12][31];          /*deklaracja i inicjalizacja dwu wymiarowej
tablicy typu int*/

int dzien = dni[0];        // przypisanie do zmiennej dzien pierwszego elementu tablicy dni
```

Operatory

Operatory arytmetyczne:

Operator	Działanie
+	Dodawanie
-	Odejmowanie
*	Mnożenie
/	Dzielenie
%	Dzielenie modulo
++	+1
--	-1

W języku Java istnieją też operatory z przypisaniem np. wyrażenie:

zmienna = zmienna + 5;

możemy zapisać jako:

zmienna += 5;

Operatory "--", jak i "++" możemy zapisać przed lub po zmiennej co w konsekwencji spowoduje że działanie zostanie wykonane odpowiednio przed, lub po danym wyrażeniu, np.:

```
int a = 5;
int b = a++;
int c = ++a;
```

Powyższy kod sprawi że zmienna b będzie wynosić 5, a zmienna c równać się będzie 7.

Operatory porównania:

Operator	Znaczenie
==	Równy
!=	Różny
>	Większy
<	Mniejszy
>=	Większy lub równy
<=	Mniejszy lub równy

Operatory logiczne:

Operator	Działanie
||	Alternatywa
&&	Koniunkcja

Instrukcje warunkowe

Instrukcje warunkowe sprawiają, że gdy spełniony jest warunek, wykonywany jest określony fragment kodu, w przeciwnym razie fragment ten zostaje pominięty.

Instrukcja if:

```
if(warunek==true)
{
        //instrukcje_jeśli_PRAWDA

}
```

W instrukcji if, jeśli warunek będzie spełniony, wykonają się instrukcje w nawiasie klamrowym.

Instrukcja if...else:

```
if(a>b)
{
        //instrukcje_jeśli_PRAWDA

}else
{

        //instrukcje_jeśli_FAŁSZ

}
```

W instrukcji if...else, jeśli warunek będzie spełniony, wykonają się instrukcje w pierwszym nawiasie klamrowym(tym po if),
jeśli warunek nie jest spełniony wykonają się instrukcje z nawiasu klamrowego po else.

Instrukcja if...else if:

```
if(a>30)
{
        //instrukcje_jeśli_a>30

}
else if(a>20)
{

        //instrukcje_jeśli_a>20

}
else if(a>10)
{

        //instrukcje_jeśli_a>10
```

```
}
else
{
        //instrukcje_jeśli_FAŁSZ
}
```

W instrukcji if..else if, warunki są sprawdzane z góry na dół, aż do momentu gdy dany warunek nie okaże się prawdziwy,
jeśli żaden warunek nie jest prawdziwy wykonają się instrukcje z nawiasu klamrowego po else.
Przy tej instrukcji należy zwrócić uwagę na kolejność sprawdzanych warunków.

Instrukcja switch:

```
switch(zmienna)
{
        case 10:

                //zmienna==10
                break;

        case 20:

                //zmienna==20
                break;

        case 25:
        case 30:

                //zmienna==25 || zmienna 30
                break;

        default:

                //inna wartość zmiennej
                break;

}
```

W instrukcji *switch* instrukcje wykonują się od momentu, gdy wartość przy *case* będzie równa wartości w zmiennej.
Na końcu każdego warunku jest *break*, który przerywa wykonywanie instrukcji, jeśli go nie będzie kolejne warunki będą się wykonywały.
Gdy nie będzie można dopasować zmiennej wykonają się instrukcje umieszczone w sekcji *default*.

Pętle

Pętle służą do powtarzania fragmentów kodu.

Pętla while:

```
while(stan == true)
{
        //instrukcje umieszczone w pętli

}
```

Pętla ta będzie wykonywała się dopóki zmienna *stan* nie będzie miała wartości *false*.
Jeśli zmienna *stan* nie będzie miała wartości *true* przy pierwszym sprawdzaniu, pętla ta nie wykona się ani razu.

Pętla do-while:

```
do
{
        //instrukcje umieszczone w pętli

}while(stan==true)
```

Pętla ta będzie wykonywała się dopóki zmienna *stan* nie będzie miała wartości *false*.
Nawet jeśli zmienna *stan* nie będzie miała wartości *true* przy pierwszym sprawdzaniu, pętla ta wykona się co najmniej jeden raz.

Pętla for:

```
for(int i=1 ; i<10 ; i++)
{
        //instrukcje umieszczone w pętli

}
```

Powyższa pętla dzieli się na trzy części oddzielone znakami ' **;** ', pierwsza służy do deklaracji i inicjalizacji zmiennych, druga sprawdza warunek - jeśli warunek będzie prawdziwy pętla będzie wykonywała się nadal, trzecia pozwala na dokonanie operacji na zmiennej.

Klasy i metody

Klasy to podstawa javy, wewnątrz klas znajdować się musi cały kod aplikacji. Metody są umieszczane w ciele klasy, dla znawców C++ dodam tylko że metody to nic innego jak zwykłe funkcje.

```
class Klasa1
{
	int dodaj(int a, int b)
	{

		return a+b;

	}

}
```

Tak właśnie wygląda najprostsza klasa i metoda w niej zawarta.
Metoda *dodaj* jako parametry przyjmuje dwie liczby całkowite i zwraca ich sumę.
Oto przykład użycia tej klasy:

```
Klasa1 mat = new Klasa1();
int suma = mat.dodaj(54, 34);
```

W pierwszej linii jest deklaracja i inicjalizacja klasy.
W drugiej jest wywołanie metody *dodaj* i przypisanie zwracanej wartości do zmiennej *suma*.

To tylko podstawy klas i metod, aby móc wzorowo programować w Javie należało by jeszcze znać m.in. konstruktory klas, dziedziczenie, klasy abstrakcyjne, zagnieżdżanie klas, przeładowania metod, tworzenie pakietów i interfejsów.

Pierwszy program

A teraz jak nakazuje tradycja ;) zamieszczam kod programu Hello world.

```
class First
{
	public static void main(String args[])
	{

		System.out.println("Hello world");

	}

}
```

Program zaczyna się od metody *main*. W jej ciele jest wywołanie metody println, służącej do wypisywania tekstu na ekran.

Pierwszy aplet

Szkielet apletu korzystający z klasy **Applet** wygląda następująco:

```
import java.applet.*;
import java.awt.*;

public class APLET extends Applet
{

	public void init( )
	{
	}

	public void start( )
	{
	}

	public void paint(Graphics g)
	{
	}

	public void stop( )
	{
	}
```

```
    public void destroy( )
    {
    }
}
```

Szkielet apletu korzystający z klasy **JApplet** wygląda następująco:

```
import javax.swing.*;
import java.awt.*;

public class APLET extends JApplet
{
    public void init( )
    {
    }

    public void start( )
    {
    }

    public void paint(Graphics g)
    {
    }

    public void stop( )
    {
    }

    public void destroy( )
    {
    }
}
```

Metody dziedziczone po klasie Applet są wywoływane w różnych momentach "życia" apletu:

- metoda *init*: wywoływana jako pierwsza, na początku wywoływania apletu. Jest wywołana tylko raz. W metodzie tej możemy np. zainicjować wszystkie zmienne.
- metoda *start*: wywoływana po metodzie init i po odsłonięciu przysłoniętego apletu. W metodzie tej możemy uruchamiać wątki zatrzymane przez metodę stop.
- metoda *paint*: wywoływana jest za każdym razem gdy konieczne jest odświeżenie apletu. Obiekt Graphics jest graficznym kontekstem apletu, służy on do rysowania na nim.
- metoda *stop*: wywoływana gdy nasz aplet nie jest widoczny. Możemy w niej przerwać wątki które nie muszą działać kiedy nasz aplet pozostaje niewidoczny.
- Metoda *destroy*: wywoływana pod koniec działania naszego apletu, możemy w nim zwolnić wszystkie zasoby wykorzystywane przez nasz aplet.

Kod potrzebny do wyświetlenia apletu na stronie wygląda następująco:

```
<applet code = "APLET.class" width = "400" height = "200"> </applet>
```

Oczywiście *APLET.class* to nazwa pliku z apletem, *width* to szerokość, a *height* to wysokość.

Pierwszy program

Kod najprostszego programu:

```
import java.awt.*;
import java.awt.event.*;

public class PROGRAM extends Frame
{

	public PROGRAM( )
	{
		addWindowListener(new MyWindowAdapter( ));
	}

	public static void main(String args[ ])
	{
		PROGRAM app = new PROGRAM( );

		app.setSize(new Dimension(300, 200));
		app.setTitle("Prosta aplikacja");
		app.setVisible(true);
	}

	public void init( )
	{
	}

	public void start( )
	{
	}

	public void paint(Graphics g)
	{
	}

	public void stop( )
	{
	}

	public void destroy( )
	{
	}
}
```

```
class MyWindowAdapter extends WindowAdapter
{

	public void windowClosing(WindowEvent e)
	{
		System.exit(0);
	}
}
```

Nasz program zaczyna się od metody *main( ).*
W ciele tej metody tworzony jest nowy obiekt klasy PROGRAM. W konstruktorze tej klasy dodajemy klasę PROGRAM do obsługi zdarzeń przez klasę *MyWindowAdapter*, która z kolei dziedziczy metodę wywoływaną gdy ktoś będzie chciał zamknąć nasz program(*windowClosing*).
Funkcja *exit(0)* zamyka nasz program.

W dalszej części metody *main* wywoływane są następujące metody klasy app:

- metoda *setSize(300, 200)*: ustawia rozmiary naszego programu. Jako parametry przyjmuje odpowiednio szerokość i wysokość naszego okna w pikselach.
- metoda *setTitle("Prosta aplikacja")*: ustawia nazwę naszego programu. Jako parametr przyjmuje ciąg znaków.
- metoda *setVisible(true)*: ustawia czy nasza aplikacja ma być widoczna. Wartość *true* oznacza że nasz program będzie widoczny.

Metody dziedziczone po klasie Frame są te same co w klasie Applet:

- metoda *init*: wywoływana jako pierwsza, na początku wywoływania programu. Jest wywołana tylko raz. W metodzie tej możemy np. zainicjować wszystkie zmienne.
- metoda *start*: wywoływana po metodzie init i po odsłonięciu przysłoniętego programu. W metodzie tej możemy uruchamiać wątki zatrzymane przez metodę stop.
- metoda *paint*: wywoływana jest za każdym razem gdy konieczne jest odświeżenie programu. Obiekt Graphics jest graficznym kontekstem programu, służy on do rysowania na nim.
- metoda *stop*: wywoływana gdy nasz program nie jest widoczny. Możemy w niej przerwać wątki które nie muszą działać kiedy nasz program pozostaje niewidoczny.
- metoda *destroy*: wywoływana pod koniec działania naszego programu, możemy w nim zwolnić wszystkie zasoby wykorzystywane przez nasz program.

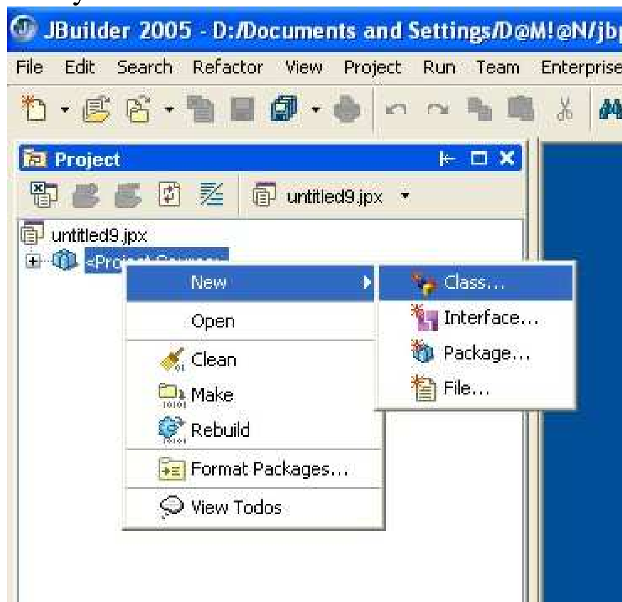
Aplet w jBuilderze

- Teraz pokażę jak wygląda tworzenie apletu w Borland jBuilderze.
  Proces instalacji pomijam, podpowiem tylko że aby go zainstalować należy być zalogowanym na koncie administratora.

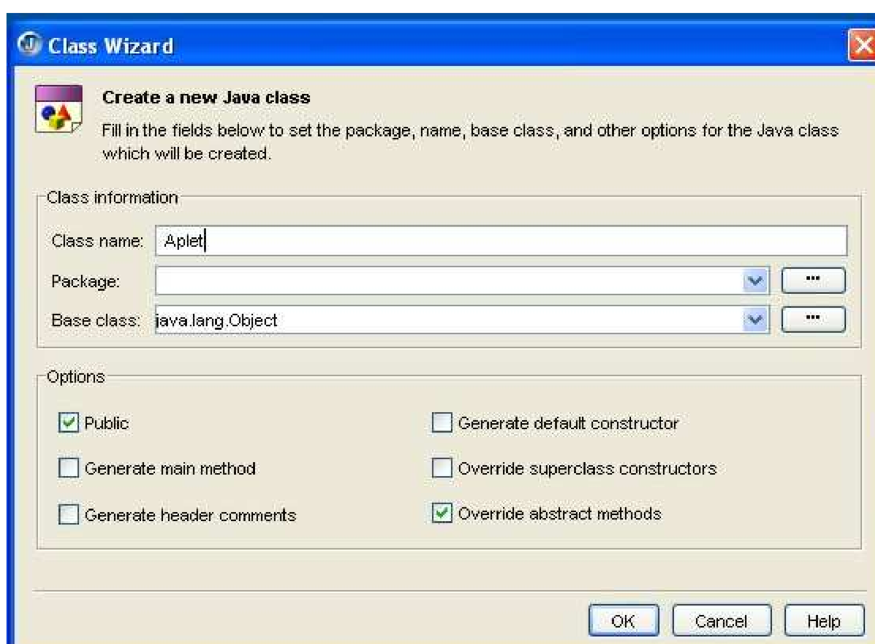
  Aby utworzyć nowy projekt z menu *File* wybieramy *New Project...*

  Nadajemy naszemu projektowi nazwę(Name) i klikamy *Finish.*

  Następnie klikamy prawym przyciskiem myszy na *Project Source* w oknie *Project*, wybieramy *new* i *class...*



W nowym oknie w polu *Class name* wpisujemy nazwę naszej klasy np. Aplet i klikamy *OK*.



Teraz wpisujemy:

- import java.applet.*;
  import java.awt.*;

  public class Aplet extends Applet
  {
- public void paint(Graphics g)
  {
- g.drawString("Pierwszy aplet", 10, 10);
- }
- }

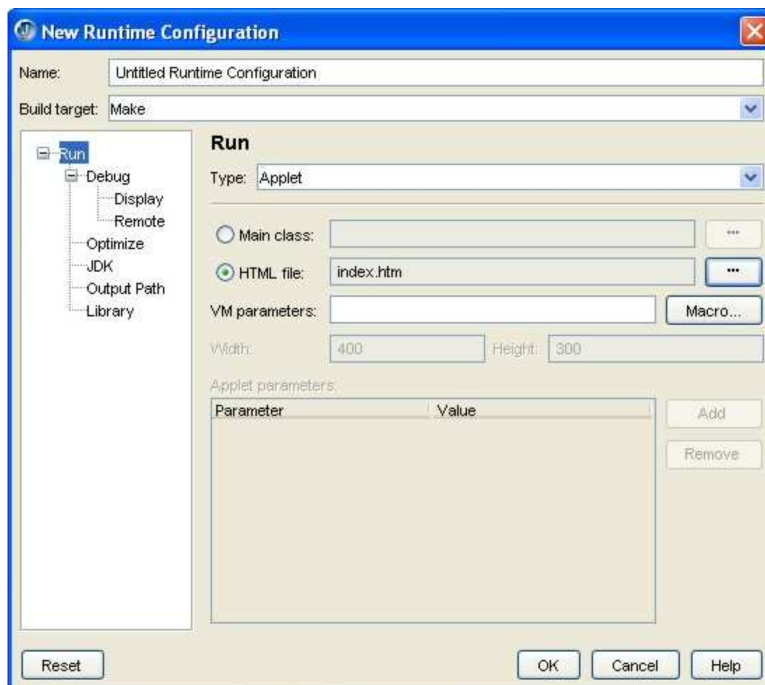
Uwaga: Nazwa klasy musi być taka sama jak nazwa pliku.

Następnie ponownie klikamy prawym przyciskiem myszy na *Project Source* w oknie *Project*, wybieramy *new* i tym razem wybieramy *File...*
Nadajemy nazwę plikowi i wybieramy rozszerzenie *.htm*

Następnie do naszego pliku htm wklepujemy:

- <applet code = "Aplet.class" width = "400" height = "200"> </applet>
- Gdzie *Aplet.class* to nazwa naszej klasy.

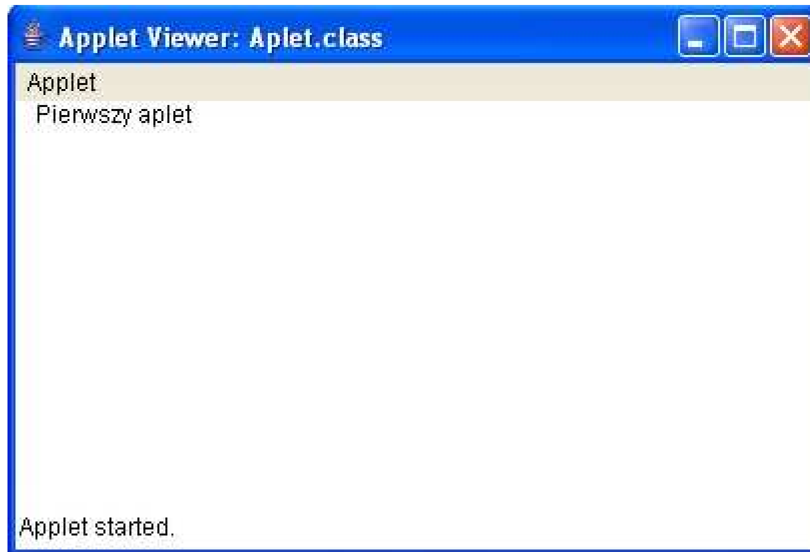
Teraz nie pozostało nam nic innego jak tylko uruchomić nasz aplet.
W tym celu wciskamy F9. Pojawi się okno konfiguracyjne:



Jako typ(Type) dajemy *Applet*
Wciskamy zaznaczenie przy *HTML file*, wciskamy przycisk "..." obok niego i wybieramy nasz plik htm.

Klikamy *OK*

Po chwili zobaczymy nasz pierwszy aplet :)



Obsługa grafiki

Rysowanie w Javie odbywa się za pomocą metod z klasy *Graphics*:

void drawString(String tekst, pozX, pozY)
Metoda ta wypisuje tekst, zaczynając od współrzędnych *pozX* i *pozY*.

void drawLine(int x1, int y1, int x2, int y2)
Metoda ta rysuje linie od współrzędnych: x1,y1 do x2,y2.

void drawRect(int x1, int y1, int szer, int wys)
Metoda rysuje prostokąt zaczynając od współrzędnych: x1,y1 i o wysokości *wys* i szerokości *szer*.

void drawRoundRect(int x1, int y1, int szer, int wys, int xArc, int yArc)
Metoda rysuje prostokąt z zaokrąglonymi rogami, zaczynający się od współrzędnych: x1,y1 i o wysokości *wys* i szerokości *szer*. Argumenty xArc, yArc reprezentują średnice zaokrąglenia dla osi X i Y.

void drawOval(int x1, int y1, int szer, int wys)
Metoda rysuje elipsę, wkomponowaną w prostokąt zaczynający się współrzędnymi: x1,y1, o wysokości *wys* i szerokości *szer*.

void drawArc(int x1, int y1, int szer, int wys, int start, int kat)
Metoda rysuje łuk, wkomponowany w prostokąt zaczynający się współrzędnymi: x1,y1, o wysokości *wys* i szerokości *szer*. Łuk jest rysowany od kąta *start*, a jego długości wynosi *kat* stopni.

void drawPolygon(int x[], int y[], int lPunktow)
Metoda rysuje dowolny wielokąt, jako współrzędne przyjmuje pary z tablic *x* i *y*, zmienna *lPunktow* odpowiada ilości kątów.

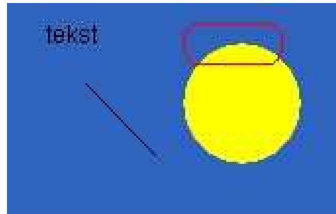
Powyższe figury można także rysować z wypełnieniem wystarczy że początek nazwy metody - draw zmienimy na fill,
np. drawRect - fillRect.
Kontekst graficzny(klasa Graphics) możemy uzyskać jako argument metody *Paint*, lub wywołując metodę *getGraphics()*.
Do zmiany tła służy metoda *setBackground(Color c)*, a do zmiany koloru rysowanych figur służy metoda klasy *Graphics* - *setColor(Color c)*.

Przykład wykorzystania powyższych metod:

```
void rysuj()
{
        setBackground(new Color(46, 100, 192));
        Graphics g = getGraphics();
        g.drawString("tekst", 20, 20);
        g.drawLine(75, 75, 40, 40);
        g.setColor(Color.yellow);
```

```
        g.fillOval(90, 20, 60, 60);
        g.setColor(Color.red);
        g.drawRoundRect(90, 10, 50, 20, 15, 15);
}
```

Wywołanie tej funkcji powinno wyglądać następująco:



Wielowątkowość

W javie wątki można tworzyć na dwa różne sposoby:

- przez wykorzystanie interfejsu Runnable
- poprzez rozszerzenie klasy Thread

Na razie opiszę tylko ten pierwszy sposób.
Interfejs Runnable wymaga dodanie do klasy tylko metody *run*.
Oto przykład klasy implementującej interfejs Runnable:

```
class watki implements Runnable
{
	Thread thread;

	watki(String name)
	{
		thread = new Thread(this, name); thread.start( );
	}

	public void run( )
	{
		for(int i=600;i>0;i--)
		{
			try
			{
				thread.sleep(100);
			} catch(InterruptedException ex) { }
		}
	}
}
```

Aby utworzyć wątek, wystarczy tylko zainicjować nawy obiekt klasy *watki*:

```
watki watek1 = new watki( "watek_pierwszy" );
```

Powyższy wątek będzie działał jedną minutę.

Omówię teraz klasę *watki*:
Konstruktor tej klasy przyjmuje jako parametr nazwę wątku.
Aby zainicjować wątek trzeba wywołać jeden z jego konstruktorów, my wybraliśmy konstruktor, który jako pierwszy parametr przyjmuje klasę implementującą interfejs *Runnable*, jako że konstruktor wywoływany jest w takiej właśnie klasie wpisujemy tylko *this*.
Metoda *start* uruchamia metodę run w nowym wątku.
W tej metodzie jest wywoływana metoda *sleep*, która zatrzymuje wątek na określony czas w milisekundach.
Metoda *sleep* może wygenerować wyjątek dlatego należy umieścić ją w bloku *try*.

Klasa String

Klasa ta służy do obsługi ciągów znaków, czyli po prostu tekstów.

Konstruktory:
Omawiana klasa ma kilka konstruktorów, ale najprościej jest zainicjalizować nowy obiekt za pomocą argumentu przypisania.

```
String str = "Ala ma kota ;)";
```

Oto niektóre konstruktory klasy String:

```
String(char text[ ])
String(byte ascii[ ])
String(char text[ ], int poczatek, int ile)
String(byte ascii[ ], int poczatek, int ile)
```

```
char txt[ ] = {'J', 'a', 'v', 'a', ' ', 'p', 'r', 'o', 'g', 'r', 'a', 'm', 'm', 'i', 'n', 'g'};
byte txt_b[ ] = {65, 66, 67};

String str = new String(txt);
String str2 = new String(txt_b);
String str3 = new String(txt, 5, 7);

System.out.println(str);
System.out.println(str2);
System.out.println(str3);
```

Dwie pierwsze metody są proste, w pierwszej jako jedyny argument podaje się tablice znaków, a w drugiej liczby odpowiadające znakom zgodnie ze standardem ASCII.
Ostatnia metoda określa dodatkowo w drugim argumencie początek zapisu (liczony od zera), i w trzecim ilość znaków.
Tak że na ekranie zobaczymy:

Java programming
abc
program

Łączenie ciągów znaków:
Łączenie ciągów znaków ze sobą jest banalnie proste, służy do tego operator + .

```
String str = "Ala " + "ma " + "kota";
String str2 = str + ",i " + 2 + " psy.";
```

Jak widać z powyższego przykładu do ciągu znaków można przyłączać także liczby.

Metody klasy String:

- void getChars(int startString, stopString, char text[ ], int textStart)
  Metoda przypisuje do tablicy *text* ciąg znaków, zaczynając od *startString*, a kończąc na *stopString-1*, przypisanie do tablicy zaczyna się od elementu wynoszącego *textStart*.

- char[] toCharArray()
  Prostsza wersja wcześniej omawianej metody *getChars*, jej wywołanie zwraca tablice znaków *Char*.

- byte[] getBytes()
  Metoda zwraca tablice liczb *byte*.

- boolean equals(String str)
  Metoda porównuje dwa ciągi znaków, jeśli są one takie sama zwraca wartość *true*.

- boolean equalsIgnoreCase(String str)
  Metoda służy do porównania dwóch ciągów znaków, bez uwzględnienia wielkości liter.

- int campareTo(String str)
  Metoda sprawdza czy w kolejności alfabetycznej String jest równy, większy lub mniejszy od podanego *str*.
  Jeśli wartość jest mniejsza od zera String jest mniejszy, jeśli jest większy od zera to String jest większy, a jeśli równy zero to oba ciągi są identyczne.

- int campareToIgnoreCase(String str)
  Metoda działa tak samo jak poprzednia, z tym tylko że ignoruje wielkość liter.

- int indexOf(int) lub int indexOf(string)
  Metoda poszukuje wystąpienia pierwszego znaku, lub ciągu znaków podanego w argumencie i zwracają indeks znalezionego znaku, lub początku ciągu znaków.
  Jeśli w Stringu nie będzie podanego znaku, lub ciągu znaków metoda zwróci wartość -1.

- int lastIndexOf(int) lub int lastIndexOf(string)
  To samo co powyżej, tylko poszukiwanie zaczyna się od końca.

- String substring(int startString) lub String substring(int startString, int stopString)
  Metoda zwraca część ciągu znaku zaczynającego się od indeksu startString, możemy także określić indeks końcowy stopString.

- String replace(char orginal, char zamiennik)
  Metoda zastępuje wszystkie wystąpienia jednego znaku na inny znak i zwraca zmieniony ciąg znaków.

- String trim()
  Metoda usuwa początkowe i końcowe białe znaki i zwraca poprawiony ciąg znaków.

- String toLowerCase()
  Zmienia wszystkie duże litery tekstu na małe.

- String toUpperCase()
  Konwertuje wszystkie małe litery tekstu na duże.

Liczby pseudolosowe

Liczby pseudolosowe tworzy się za pomocą równomiernie rozłożonych sekwencji.
Pod wieloma względami liczby te niczym nie ustępują liczbom losowym, dlatego z powodzeniem mogą być wykorzystywane w różnego rodzaju grach i aplikacjach.

Za generowanie liczb pseudolosowych w Javie odpowiada klasa *Random*, z pakietu *java.util*.
Klasa ta ma dwa konstruktory:
Pierwszy, nie przyjmuje żadnego parametru, i jako wartość początkową (tzw. ziarna)przyjmuje aktualny czas.
Drugi, jako jedyny parametr przyjmuje liczbę typu long, która jest wartością początkową sekwencji losowej.

Klasa ta ma wiele metod, zwracających liczby pseudolosowe różnych typów.
Najprostsze są metody typu nextTyp, nie przyjmujące żadnego parametru, a zwracające określony typ:

- boolean nextBoolean()
- double nextDouble()
- float nextFloat()
- int nextInt()
- long nextLong()

Ponadto klasa ta ma inne metody:

- int nextInt(int n)
  Metoda zwraca liczbę typu int z przedziału od 0 do n.
- void nextBytes(byte t[])
  Wypełnia tablice t losowo wygenerowanymi liczbami.
- double nextGaussian()
  Metoda zwraca liczbę z zakresu tzw. krzywej Gaussa.

Timer

Działanie Timeru jest z pewnością znane użytkownikom środowisk programistycznych firmy Borland (np. C++Builder, Delphi) i platformy .NET Framework (np. Visual Studio .NET). Ogólnie mówiąc Timer jest osobnym wątkiem, uruchamianym o określonym czasie, i ew. powtarzanym w równych odstępach czasu.

Klasie Timer towarzyszy zawsze klasa TimerTask.
TimerTask implementuje interfejs *Runnable*(patrz: Wielowątkowość), zawiera więc abstrakcyjną metodę *run()*, którą trzeba przykryć.
Od tej metody rozpoczyna się wykonywanie wątku.
Kolejną metodą klasy TimerTask jest metoda *cancel()*, która przerywa wątek i zwraca wartość *true* jeśli się to udało.

Klasa Timer ma kilka konstruktorów:

- **Timer()**
  Domyślny konstruktor tworzący obiekt, który będzie wykonywany jak zwykły wątek.

- **Timer(boolean tThread)**
  Jeśli argument tThread ma wartość true, wykorzystany zostanie dodatkowy wątek demona.

- **Timer(String tName)**
  Konstruktor dodany dopiero w wersji J2SE 5, umożliwia określenie nazwy wątku.

- **Timer(String tName, boolean tThread)**
  Konstruktor dodany dopiero w wersji J2SE 5, umożliwia określenie nazwy wątku i jeśli argument tThread ma wartość true, wykorzystany zostanie dodatkowy wątek demona.

Do planowania zadania służą metody schedule i scheduleAtFixedRate:

- **schedule(TimerTask TT, long wait)**
  Argument TT jest naszym zadaniem, a argument wait to czas uruchomienia zadania liczony w milisekundach od uruchomienia programu.

- **schedule(TimerTask TT, long wait, long repeat)**
  Argument TT jest naszym zadaniem, argument wait to czas uruchomienia zadania liczony w milisekundach od uruchomienia programu, a argument repeat to czas w którym będzie powtarzany wątek liczony w milisekundach.

- **schedule(TimerTask TT, Date data)**
  Argument TT jest naszym zadaniem, a argument data jest obiektem klasy Date, służącym do wyznaczenia dokładnego czasu uruchomienia zadania.

- **schedule(TimerTask TT, Date data, long repeat)**
  Argument TT jest naszym zadaniem, argument data jest obiektem klasy Date, służącym do wyznaczenia dokładnego czasu uruchomienia zadania, a argument repeat to czas w którym będzie powtarzany wątek liczony w milisekundach.

- **scheduleAtFixedRate(TimerTask TT, long wait, long repeat)**
  Argument TT jest naszym zadaniem, argument wait to czas uruchomienia zadania liczony w milisekundach od uruchomienia programu, a argument repeat to czas w którym będzie powtarzany wątek liczony w milisekundach, z tym że odstępy wykonywania wątku są uruchamianie względem pierwszego jego uruchomienia.

- **scheduleAtFixedRate(TimerTask TT, Date data, long repeat)**
  Argument TT jest naszym zadaniem, argument data jest obiektem klasy Date, służącym do wyznaczenia dokładnego czasu uruchomienia zadania, a argument repeat to czas w którym będzie powtarzany wątek liczony w milisekundach, z tym że odstępy wykonywania wątku są uruchamianie względem pierwszego jego uruchomienia.

Teraz pora zastosować klasę Timer w praktyce.
Napiszemy aplet który po upływie jednej minuty zacznie migać:

```
import java.applet.*;
import java.awt.*;
import java.util.*;

public class TIMER extends Applet
{
        static public Applet naszAplet;

        public void init( )
        {
                Timer timer = new Timer( );
                Zadanie zadanie = new Zadanie( );

                timer.schedule(zadanie, 60000, 500);

                naszAplet = this;
        }
}

class Zadanie extends TimerTask
{
        static boolean czerwony = true;
        public void run( )
        {
                if(czerwony == true)
                {
                        TIMER.naszAplet.setBackground(Color.red);
                        czerwony=false;
                }
                else
                {
                        TIMER.naszAplet.setBackground(Color.yellow);
                        czerwony=true;
                }
        }
}
```

Klasa Math

Klasa z pakietu java.lang zawiera bardzo dużo metod służących do obliczeń matematycznych. Metody tej klasy są statyczne.

- **double sin(double arg)**
  Metoda zwraca sinus kąta arg podanego w radianach.
- **double cos(double arg)**
  Metoda zwraca cosinus kąta arg podanego w radianach.
- **double tan(double arg)**
  Metoda zwraca tangens kąta arg podanego w radianach.
- **double asin(double arg)**
  Metoda zwraca kąt w radianach sinusa podanego w arg.
- **double acos(double arg)**
  Metoda zwraca kąt w radianach cosinusa podanego w arg.
- **double atan(double arg)**
  Metoda zwraca kąt w radianach tangensa podanego w arg.
- **double sinh(double arg)**
  Metoda zwraca hiperboliczny sinus kąta arg podanego w radianach.
- **double cosh(double arg)**
  Metoda zwraca hiperboliczny cosinus kąta arg podanego w radianach.
- **double tanh(double arg)**
  Metoda zwraca hiperboliczny tangens kąta arg podanego w radianach.
- **double toRadian(double arg)**
  Metoda zamienia stopnie na radiany.
- **double toDegrees(double arg)**
  Metoda zamienia radiany na stopnie.
- **double cbrt(double arg)**
  Metoda zwraca pierwiastek sześcienny z arg.
- **double log(double arg)**
  Metoda zwraca logarytm naturalny z arg.
- **double log1p(double arg)**
  Metoda zwraca logarytm naturalny z arg+1.
- **double log10(double arg)**
  Metoda zwraca logarytm dziesiętny z arg.

- **double pow(double arg, double n)**
  Metoda zwraca arg podniesiony do potęgi n.

- **double sqrt(double arg)**
  Metoda zwraca pierwiastek drugiego stopnia z arg.

Klasa Math definiuje ponadto dwie stałe:

- **PI** = 3.141592653589793

- **E** = 2.718281828459045

Klasa StrictMath definiuje identyczne metody co klasa Math, z tą tylko różnicą, że otrzymane wyniki będą dokładnie takie same we wszystkich implementacjach Javy.

Zdarzenia myszy

W javie aby obsłużyć zdarzenia myszy należy zaimplementować interfejs *MouseListener* z pakietu *java.awt.event.**.
Interfejs ten wymusza dodanie następujących metod:

- **public void mouseClicked(MouseEvent e)**
  Reaguje na kliknięcie myszy.

- **public void mouseEntered(MouseEvent e)**
  Reaguje na wejście kursora myszy w obszar komponentu.

- **public void mouseExited(MouseEvent e)**
  Reaguje na wyjście kursora myszy z obszaru komponentu.

- **public void mousePressed(MouseEvent e)**
  Reaguje na naciśnięcie przycisku myszy.

- **public void mouseReleased(MouseEvent e)**
  Reaguje na zwolnienie przycisku myszy.

Wszystkie te metody przyjmują jako argument klase *MouseEvent*
Oto jej ważniejsze metody:

- **Point getPoint( )**
  Zwraca położenie kursora myszy.
  Alternatywne metody to: *int getX( )* i *int getY( ).*

- **int getButton( )**
  Zwraca wartość przycisku myszy odpowiedzialnego za wygenerowanie zdarzenia.
  Wartości te odpowiadają stałym *NOBUTTON*, *BUTTON1*, *BUTTON2* i *BUTTON3.*

Aby zarejestrować klasę nasłuchującą zdarzenie (implementującą interfejs MouseListener) należy wywołać metodę *addMouseListener* jako parametr podając klasę implementującą to zdarzenie.

A teraz przykład obsługi zdarzeń myszy.
Napiszemy prostą grę, która będzie mierzyć czas kolejnych kliknięć gracza na losowo wygenerowany kwadrat:

```
import java.awt.*;
import java.applet.*;
import java.util.*;
import java.awt.event.*;

public class fajer extends Applet implements MouseListener
{

        int pozX, pozY;
        Random r = new Random( ); // 1
```

```
	float czas;
	Date czas1=new Date( ), czas2= new Date( ); // 2

	public void init( )
	{
		addMouseListener(this);
		losuj( );
	}

	public void paint(Graphics g)
	{
		g.drawString("Czas reakcji: " + czas + " s.", 20, 20);
		g.fillRect(pozX, pozY, 20, 20);
	}

	void losuj( )
	{
		pozY = r.nextInt(this.getHeight( )-20);
		pozX = r.nextInt(this.getWidth( )-20);
	}

	public void mouseClicked(MouseEvent e)
	{
		if(e.getX( )>pozX && e.getX( )<pozX+20 && e.getY( )>pozY && e.getY(
		)<pozY+20)
		{
			losuj( );
			czas2 = new Date( );
			czas = (float)(czas2.getTime( ) - czas1.getTime( ))/1000;
			czas1 = new Date( );
			repaint( );
		}
	}

	public void mouseEntered(MouseEvent e) { }
	public void mouseExited(MouseEvent e) { }
	public void mousePressed(MouseEvent e) { }
	public void mouseReleased(MouseEvent e) { }
}
```

Kod wygenerowany za pomocą programu: [Code Generator]

1. Klasa Random służy do generowania liczb pseudolosowych (patrz: Klasa Random).
2. Klasa Date służy nam do mierzenia czasu. Klasę tą szerzej opiszę niebawem.

Zdarzenia klawiatury

Do obsługi zdarzeń generowanych przez klawiaturę implementujemy interfejs *KeyListener* z pakietu *java.awt.event.**.
Interfejs ten wymusza dodanie następujących metod:

- **public void keyPressed(KeyEvent e)**
  Reaguje na naciśnięcie klawisza.

- **public void keyReleased(KeyEvent e)**
  Reaguje na zwolnienie klawisza.

- **public void keyTyped(KeyEvent e)**
  Reaguje gdy wciśnięty klawisz (albo klawisze) spowodują wygenerowanie jakiegoś znaku.

Wszystkie te metody przyjmują jako argument klasę *KeyEvent*
Oto jej ważniejsze metody:

- **char getKeyChar( )**
  Zwraca znak klawisza, który wygenerował akcję.

- **int getKeyCode( )**
  Zwraca identyfikator klawisza, który wygenerował akcję.
  Wartości te odpowiadają różnym stałym zaczynającym się od *VK_*, np. VK_1, VK_A, VK_ENTER.

Aby zarejestrować klasę nasłuchującą zdarzenie należy wywołać metodę *addKeyListener*, jako parametr podając klasę implementującą to zdarzenie.
Musimy także wywołać metodę *requestFocus( ).*

A teraz jakiś przykład.
Napiszemy aplet który będzie mierzył naszą szybkość pisania na klawiaturze.

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;

public class Klawiatura extends Applet
implements KeyListener
{

        private int stan = 1;
        private String teksty[ ] = {
                "\nW hackingu nie możliwe są tylko 2 rzeczy: polubić XP-ka i nakarmić mysz
                serem.",
                "\nSztuczna inteligencja lepsza jest od naturalnej głupoty.",
```

```
        "\nLudzie dzielą się na 10 grupy, na tych co rozumieją kod dwójkowy i na tych
        co go nie rozumieją.",
        "\nPrawdziwy informatyk wiesza się razem ze swoim programem.",
        "\nKomputer służy do tego, aby ułatwić ci pracę, której bez niego w ogóle byś
        nie miał."
};
private int nr_tekstu, max=4;
private String tekst = new String( );
private Date data1, data2;

private int policz( )
{
        long liter = 0;
        for (String str : teksty)
        {
                liter += str.length( );
        }
        data2 = new Date( );

        return (int)(liter/(((float)(data2.getTime( )-data1.getTime( ))/1000)/60));
}

public void init( )
{
        addKeyListener(this);
        requestFocus( );
        setBackground(Color.black);
}

public void paint(Graphics g)
{
        g.setColor(Color.white);

        switch(stan)
        {
                case 1:
                g.drawString("Program do mierzenia szybkości pisania na
                klawiaturze", 10, 20);
                g.drawString("Aby rozpocząć wciśnij ENTER", 10, 40);
                break;

                case 2:
                g.drawString(teksty[nr_tekstu], 10, 20);
                g.setColor(Color.orange);
                g.drawString(tekst, 10, 40);
                break;

                case 3:
                g.drawString("Koniec gry " , 10, 20);
                g.drawString("Twój wynik to: " + policz( ) + " znaków na minutę.", 10,
```

```
                40);
                g.drawString("Jeśli chcesz spróbować jeszcze raz wciśnij ENTER." ,
                10, 60);
                break;
        }
}

public void keyPressed(KeyEvent e)
{
        if(e.getKeyCode( ) == e.VK_ENTER)
        {
                switch(stan)
                {
                        case 1:
                        nr_tekstu = 0;
                        stan=2;
                        data1 = new Date( );
                        break;

                        case 2:
                        if(tekst.equals(teksty[nr_tekstu]) == true)
                        {
                                if(nr_tekstu==max) stan=3;
                                nr_tekstu++;
                        }
                        else
                        {
                                Graphics g = getGraphics( );
                                g.setColor(Color.white);
                                g.drawString("Źle, popraw się!", 500, 60);
                                try
                                {
                                        Thread.sleep(2000);
                                } catch (InterruptedException ex){ }
                        }
                        tekst = new String( );
                        break;

                        case 3:
                        stan = 1;
                        break;
                }
                repaint( );
        }
}

public void keyTyped(KeyEvent e)
{
        if(stan==2)
        {
```

```
                    if(e.getKeyChar( ) == '\b')
                    tekst = tekst.substring(0, tekst.length( )-1); else tekst = tekst +
                    e.getKeyChar( );
                    repaint( );
            }
    }

    public void keyReleased(KeyEvent e) { }
}
```