Programowanie obiektowo-funkcyjne

Sprawozdanie

Szymon Wątorowski

Laboratorium 2a

Zadanie A – Hello i pętla:

Program wyświetla nam napis „Witaj!” i liczby od 1 do X, gdzie X to liczba (integer) podana jako parametr podczas uruchamiania programu.
W klasie *Hello* obsługujemy metodę *main* przyjmującą argumenty. W *main’*ie sprawdzamy czy liczba argumentów jest równa 1, w przeciwnym wypadku wyświetli nam na konsoli wiadomość z błędem.
W przypadku ominięcia błędu na ekranie wyświetli nam się „Witaj!”.
Tworzymy zmienną *myInt*, która zamienia nam nasz parametr na liczbę (*Integer*) i wykorzystujemy ją w naszej pętli do wyświetlenia liczb od 1 do liczby podanej jako nasz argument.

Kod:

```java
package pojava.lab2a.zadA;

public class Hello {  new *
    public static void main(String[] args) {  new *
        if (args.length != 1) {
            System.out.println("Error: podano złą ilość argumentów!");
            return;
        }

        System.out.println("Witaj");
        String myString = args[0];
        int myInt = Integer.parseInt(myString);
        for (int i = 1; i <= myInt; i++) {
            System.out.println(i);
        }
    }
}
```

Output:

```
Witaj
1
2
3
4
5
6
7
8
9
10

Process finished with exit code 0
```

Zadanie B – Tablica:

Program pobiera cztery argumenty z wiersza poleceń i zapisuje je do tablicy, a następnie wyświetla jej zawartość na ekranie.

W klasie Tablica obsługujemy metodę *main*, która przyjmuje argumenty. Na początku sprawdzamy, czy liczba argumentów jest mniejsza niż 4. Jeśli tak, program wyświetla komunikat błędu i kończy działanie.

Tworzymy tablicę *table* o rozmiarze 4 i przypisujemy do niej wartości z argumentów. Następnie wypisujemy jej zawartość na ekran przy użyciu *Arrays.toString()*.

Kod:

```
package pojava.lab2a.zadB;

import java.util.Arrays;

public class Tablica {  new *
    public static void main(String[] args) {  new *
        if (args.length < 4) {
            System.out.println("Error: Podano za mało argumentów!");
            return;
        }
        String[] table = new String[4];

        for (int i = 0; i <= args.length-1; i++) {
            table[i] = args[i];
        }

        System.out.println(Arrays.toString(table));
    }
}
```

Output:

```
[20, 50, 4, 7]

Process finished with exit code 0
```

Zadanie C – Dziedziczenie

Program symuluje działanie samochodu i taksówki, generując losowe wartości przebiegu oraz zarobków dla kolejnych 12 miesięcy, a następnie oblicza ich średnie wartości.
W klasie *Auto* definiujemy tablicę *przebieg*, w której przechowujemy miesięczny przebieg pojazdu. W konstruktorze klasy losujemy wartości przebiegu w zakresie od 500 do 2000 km. Następnie tworzymy metodę *srPrzebieg()*, która sumuje wartości z tablicy i zwraca ich średnią.
W klasie *Taxi*, która dziedziczy po *Auto*, definiujemy dodatkową tablicę *zarobki*, przechowującą miesięczne zarobki. W konstruktorze przypisujemy jej losowe wartości w przedziale od 3000 do 10000 zł. Dodajemy także metodę *srZarobki()*, która oblicza średnią miesięcznych zarobków.
W metodzie *main()* tworzymy obiekt klasy *Taxi*, a następnie wyświetlamy średni przebieg i średnie zarobki.

Kod *Auto*:

```java
package pojava.lab2a.zadC;

import java.util.Random;

public class Auto {  1 usage  1 inheritor  new *
    protected float[] przebieg;  4 usages

    public Auto() {  1 usage  new *
        przebieg = new float[12];
        Random random = new Random();
        for (int i = 0; i < 12; i++) {
            przebieg[i] = 500 + random.nextFloat() * 1500;
        }
    }

    public float srPrzebieg() {  1 usage  new *
        float suma = 0;
        for (float km : przebieg) {
            suma += km;
        }
        return suma / przebieg.length;
    }

}
```

Kod *Taxi*:

```java
package pojava.lab2a.zadC;

import java.util.Random;

public class Taxi extends Auto {  new *
    private float[] zarobki;  4 usages

    public Taxi() {  1 usage  new *
        super(); // Wywołanie konstruktora klasy Auto
        zarobki = new float[12];
        Random random = new Random();
        for (int i = 0; i < 12; i++) {
            zarobki[i] = 3000 + random.nextFloat() * 7000; // Losowe wartości od 3000 do 10000
        }
    }

    public float srZarobki() {  1 usage  new *
        float suma = 0;
        for (float zarobek : zarobki) {
            suma += zarobek;
        }
        return suma / zarobki.length;
    }

    public static void main(String[] args) {  new *
        Taxi taxi = new Taxi();
        System.out.println("Średni przebieg: " + taxi.srPrzebieg() + " km");
        System.out.println("Średnie zarobki: " + taxi.srZarobki() + " zł");
    }
}
```

Output:

```
Średni przebieg: 1156.1323 km
Średnie zarobki: 6543.129 zł

Process finished with exit code 0
```

Laboratorium 2b

Zadanie A – Dziedziczenie po JFrame

Program tworzy okno graficzne w Javie, które można zamknąć.
W klasie *CloseableFrame* rozszerzamy *JFrame* i definiujemy kilka konstruktorów, które pozwalają na utworzenie okna z różnymi parametrami:

- Bezargumentowy konstruktor ustawia domyślny rozmiar okna na 640x480 pikseli i ustawia opcję zamykania *DISPOSE_ON_CLOSE*.
- Konstruktor przyjmujący *GraphicsConfiguration* pozwala na określenie konfiguracji graficznej.
- Konstruktor przyjmujący *String title* umożliwia ustawienie tytułu okna.
- Konstruktor z parametrami *String title*, *GraphicsConfiguration gc* łączy możliwość ustawienia tytułu oraz konfiguracji graficznej.

W metodzie *main()* tworzymy obiekt *CloseableFrame*, ustawiamy jego widoczność na *true*, dzięki czemu okno pojawia się na ekranie.

Kod:

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

public class CloseableFrame extends JFrame {  new *
    public CloseableFrame() throws HeadlessException {  6 usages  new *
        this.setSize( width: 640, height: 480);
        setDefaultCloseOperation(DISPOSE_ON_CLOSE);
    }
    public CloseableFrame(GraphicsConfiguration gc) {  no usages  new *
        super(gc);
        this.setSize( width: 640, height: 480);
        setDefaultCloseOperation(DISPOSE_ON_CLOSE);
    }
    public CloseableFrame(String title) throws HeadlessException {  no usages  new *
        super(title);
        this.setSize( width: 640, height: 480);
        setDefaultCloseOperation(DISPOSE_ON_CLOSE);
    }
    public CloseableFrame(String title, GraphicsConfiguration gc) {  no usages  new *
        super(title, gc);
        this.setSize( width: 640, height: 480);
        setDefaultCloseOperation(DISPOSE_ON_CLOSE);
    }
    public static void main(String[] args) {  new *
        CloseableFrame cf = new CloseableFrame();
//        cf.setSize(640, 480);
        cf.setVisible(true);
    }
}
```

Output:

Zadanie B – Figury w losowych kolorach

Program tworzy interfejs graficzny z losowo kolorowanymi kształtami oraz interaktywnymi elementami.
Klasa *ThreeShapesPanel* rozszerza *JPanel* i definiuje tablicę *colorTable* przechowującą trzy losowe kolory. W konstruktorze klasy generujemy te kolory przy użyciu obiektu *Random*.
Metoda *paintComponent(Graphics g)* rysuje trzy kształty:

- Prostokąt w losowym kolorze,
- Koło w innym losowym kolorze,
- Trójkąt w trzecim kolorze.

W metodzie *main()* tworzymy okno *CloseableFrame*, ustawiamy jego układ na *GridLayout(1,2)*, a następnie dodajemy dwa panele:

- *ThreeShapesPanel*, który wyświetla kształty,
- *JPanel*, który zawiera dwa przyciski, etykietę tekstową oraz pole do wpisywania tekstu.

Kod:

```java
public class ThreeShapesPanel extends JPanel {  new *
    private final Color[] colorTable = new Color[3];  4 usages

    public ThreeShapesPanel() {  1 usage  new *
        Random random = new Random();
        for (int i = 0; i < 3; i++) {
            colorTable[i] = new Color(random.nextInt( bound: 256), random.nextInt( bound: 256), random.nextInt( bound: 256));
        }
    }
```

```java
    @Override  new *
    protected void paintComponent(Graphics g) {
        super.paintComponent(g);

        g.setColor(colorTable[0]);

        // Prostokat
        g.fillRect( x: 50,  y: 50,  width: 100,  height: 75);

        g.setColor(colorTable[1]);
        // Koło
        g.fillOval( x: 150,  y: 150,  width: 100,  height: 100);

        g.setColor(colorTable[2]);
        // Trójkat
        int[] xPoints = {50, 100, 150};
        int[] yPoints = {350, 250, 350};
        g.fillPolygon(xPoints, yPoints,  nPoints: 3);
    }
```

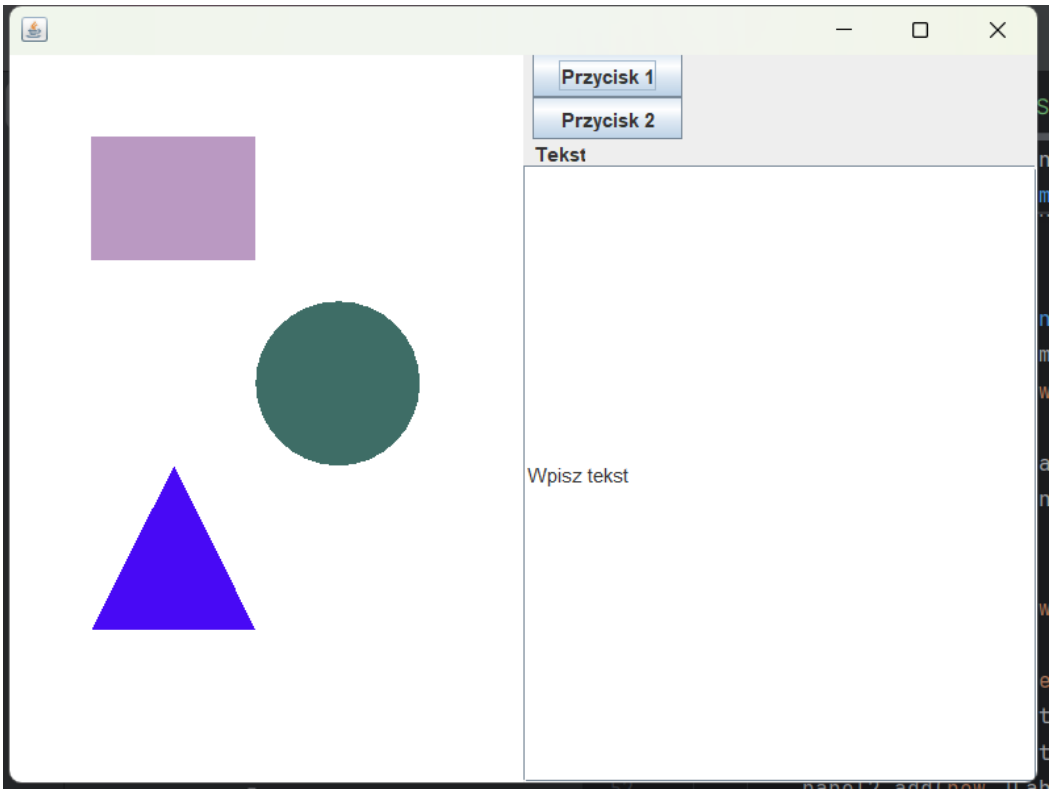
```java
    public static void main(String[] args) {  new *
        CloseableFrame frame = new CloseableFrame();
        frame.setLayout(new GridLayout( rows: 1, cols: 2));

        ThreeShapesPanel panel1 = new ThreeShapesPanel();
        panel1.setBackground(Color.white);
        frame.add(panel1);

        JPanel panel2 = new JPanel();
        frame.add(panel2);
        panel2.setLayout(new BoxLayout(panel2, BoxLayout.PAGE_AXIS));
        panel2.add(new JButton( text: "Przycisk 1"));
        panel2.add(new JButton( text: "Przycisk 2"));
        panel2.add(new JLabel( text: "Tekst"));
        panel2.add(new JTextField("Wpisz tekst"));

        frame.setVisible(true);
    }
}
```

Output:



Zadanie C – Okno z trzema przyciskami

Program tworzy interfejs graficzny z interaktywnymi przyciskami.

Klasa *ThreeButtonFrame* rozszerza *JFrame* i zawiera komponenty GUI, które pozwalają na interakcję użytkownika.

W konstruktorze tworzymy okno o wymiarach 640x480 pikseli i ustawiamy jego domyślną operację zamykania na *DISPOSE_ON_CLOSE*. Następnie dodajemy panel, który zawiera:

- Etykietę *counter* wyświetlającą wartość licznika,
- Przycisk +1, który zwiększa wartość licznika,
- Przycisk -1, który zmniejsza wartość licznika,
- Przycisk *Wyjdź*, który zamyka program,
- Etykietę *label*, której tekst można zmieniać,
- Przycisk „Zmień label”, który ustawia losową wartość w zakresie 0-99.

Panel jest rozmieszczony w układzie *GridLayout(2,3)*, co zapewnia równomierne rozmieszczenie elementów.

W metodzie *main()* tworzymy instancję *ThreeButtonFrame* i ustawiamy jej widoczność.

Kod:

```java
package pojava.lab2b.zadC;

import javax.swing.*;
import java.awt.*;
import java.util.Random;

public class ThreeButtonFrame extends JFrame {  new *
    private int a = 0;  5 usages

    public ThreeButtonFrame() throws HeadlessException {  1 usage  new *
        this.setSize( width: 640, height: 480);
        setDefaultCloseOperation(DISPOSE_ON_CLOSE);

        // Stworzenie panelu
        JPanel panel = new JPanel();
        this.add(panel);

        // Przycisk dodawania do counter
        JLabel counter = new JLabel( text: "Licznik: "+a);
        panel.add(counter);
        JButton additionButton = new JButton( text: "+1");

        additionButton.addActionListener( ActionEvent e -> { a += 1; counter.setText("Licznik: "+a); });
        panel.add(additionButton);
        JButton subtractionButton = new JButton( text: "-1");

        subtractionButton.addActionListener( ActionEvent e -> {a -= 1; counter.setText("Licznik: "+a);});
        panel.add(subtractionButton);

        // Przycisk wyjścia
        JButton exitButton = new JButton( text: "Wyjdź");

        exitButton.addActionListener( ActionEvent e -> { System.exit( status: 0);});
        panel.add(exitButton);

        // Zmiana labelu
        JLabel label = new JLabel( text: "Label");

        JButton labelButton = new JButton( text: "Zmień label");
        labelButton.addActionListener( ActionEvent e -> {Random random = new Random(); label.setText(String.valueOf(random.nextInt( bound: 100)));});

        panel.add(label);
        panel.add(labelButton);
        panel.setLayout(new GridLayout( rows: 2, cols: 3));
    }

    public static void main(String[] args) {  new *
        ThreeButtonFrame frame = new ThreeButtonFrame();
        frame.setVisible(true);
    }
}
```

Output:

Kalkulator

Program implementuje kalkulator, który obsługuje podstawowe operacje arytmetyczne i operacje na pamięci.

Zadanie 1

Dodano testy do operacji mnożenia i dodawania.

- **testAddNeg()** – tworzymy zmienną sut jako obiekt klasy Calculator, ustawiamy stan na 1, a następnie dodajemy wartość -1 za pomocą metody sut.add(-1). Sprawdzamy poprawność operacji wykorzystując assertEquals, porównując przewidywaną wartość (0) z aktualnym stanem (sut.getState()).
- **testMultOneByZero()** – tworzymy zmienną sut z wykorzystaniem klasy Calculator, ustawiamy stan na wartość 1, a następnie mnożymy go przez 0 za pomocą metody sut.mult(0). Sprawdzamy poprawność wykonania operacji wykorzystując assertEquals – porównujemy przewidywaną wartość (0) z aktualnym stanem (sut.getState()).

Zadanie 2

Dodano dwie nowe operacje wraz z testami: odejmowanie i dzielenie.

- *sub(int value)* – metoda wykonująca operację odejmowania poprzez odjęcie wartości podanej jako argument od stanu,
- *div(int value)* – metoda wykonująca operację dzielenia poprzez sprawdzenie czy wartość podana jako argument jest zerem, w przypadku w którym tak jest zmieniamy flagę błędu *err* na *true* i nie dokonujemy obliczeń (stan pozostaje bez zmian dla zapewnienia, że danej operacji nie da się wykonać). W przeciwnym wypadku stan jest skutecznie dzielony przez wartość podaną jako argument.

Testy:

- **testSubOne()** – tworzymy zmienną sut z klasy Calculator, ustawiamy stan na 1, następnie odejmujemy 1 przy użyciu metody sut.sub(1). Sprawdzamy poprawność operacji za pomocą assertEquals, porównując przewidywaną wartość (0) z aktualnym stanem (sut.getState()).

- **testSubNeg()** – inicjalizujemy sut jako obiekt klasy Calculator, ustawiamy stan na 1, po czym odejmujemy wartość -1 poprzez sut.sub(-1). Sprawdzamy poprawność operacji za pomocą assertEquals, porównując oczekiwaną wartość (2) z aktualnym stanem (sut.getState()).
- **testDivTenByTwo()** – tworzymy obiekt sut klasy Calculator, ustawiamy jego stan na 10, a następnie dzielimy przez 2 za pomocą sut.div(2). Weryfikujemy poprawność operacji poprzez assertEquals, porównując przewidywaną wartość (5) z aktualnym stanem (sut.getState()).
- **testDivTenByNeg()** – inicjalizujemy sut jako instancję klasy Calculator, ustawiamy stan na 10, następnie dzielimy przez -2 za pomocą sut.div(-2). Sprawdzamy poprawność operacji poprzez assertEquals, porównując oczekiwaną wartość (-5) z aktualnym stanem (sut.getState()).

Zadanie 3

Dodano możliwość przechowywania wyniku w pamięci oraz wykonywanie operacji z wykorzystaniem danych przechowywanych w pamięci.

- *memory* – zmienna przechowująca pamięć jako *int*,
- *saveMem()* – metoda pozwalająca na przechowanie obecnego stanu w pamięci poprzez przypisanie wartości zmiennej *state* do zmiennej *memory*,
- *useMem()* – metoda pozwalająca na wykorzystanie wartości zapisanej w pamięci poprzez dodanie jej do obecnej wartości stanu,
- *addMem()* – metoda dodająca wartość obecnego stanu do pamięci,
- *subMem()* – metoda odejmująca wartość obecnego stanu od pamięci,
- *getMem()* – metoda zwracająca obecną wartość pamięci.

Testy:

- **testSaveMemory()** – tworzymy zmienną sut z klasy Calculator, ustawiamy jej stan na 1, a następnie zapisujemy ten stan do pamięci za pomocą sut.saveMem(). Weryfikujemy poprawność operacji za pomocą assertEquals, porównując oczekiwaną wartość (1) z aktualnym stanem pamięci (sut.getMem()),
- **testUseMemory()** – tworzymy obiekt sut klasy Calculator, ustawiamy jego stan na 1 i zapisujemy tę wartość do pamięci (sut.saveMem()). Następnie dodajemy wartość zapisaną w pamięci do aktualnego stanu poprzez sut.useMem(). Sprawdzamy poprawność operacji wykorzystując assertEquals, porównując przewidywaną wartość (2) z aktualnym stanem (sut.getState()),
- **testAddMemory()** – tworzymy instancję sut klasy Calculator, ustawiamy stan na 1, a następnie dodajemy tę wartość do pamięci za pomocą sut.addMem(). Sprawdzamy poprawność operacji przy użyciu assertEquals, porównując przewidywaną wartość (1) z aktualnym stanem pamięci (sut.getMem()),
- **testSubMemory()** – inicjalizujemy sut jako obiekt klasy Calculator, ustawiamy jego stan na 1, następnie odejmujemy tę wartość od pamięci za pomocą sut.subMem(). Sprawdzamy poprawność działania operacji, używając assertEquals, porównując oczekiwaną wartość (-1) z aktualnym stanem pamięci (sut.getMem()).

Zadanie 4

Dodano obsługę błędów obliczeń.

- *err* – flaga błędu zapisana w formie *boolean* (domyślnie *false*),
- *getError()* – zwraca wartość zmiennej *err* w formie *boolean*.

Testy:

- **testDivByZero()** – tworzymy zmienną sut jako obiekt klasy Calculator, ustawiamy stan na 1 i wykonujemy dzielenie przez 0 (sut.div(0)). Sprawdzamy poprawność działania operacji wykorzystując assertTrue, weryfikując, czy flaga błędu (sut.getError()) została ustawiona na true.

Kod:

```
package kalkulator;

public class Calculator {  28 usages  szywat +1 *
    private int state = 0; // Wartość  10 usages
    private boolean err = false; // Błąd  2 usages
    private int memory = 0; // Pamięć  5 usages

    public int getState() { // Zwraca wartość  10 usages  Szymon Wąt
        return state;
    }

    public void setState(int state) { // Ustanawianie wartości  13
        this.state = state;
    }

    public void add(int value){ // Dodawanie  Szymon Wątorowski +1
        state += value;
    }

    public void mult(int value){ // Mnożenie  3 usages  Szymon Wątor
        state *= value;
    }

    public void sub(int value){ // Odejmowanie  2 usages  szywat
        state -= value;
    }
```

```
    public void div(int value){ // Dzielenie z uwzględnieniem błędu  3 usage
        if (value == 0) {
            err = true;
        } else {
            state /= value;
        }
    }

    public void saveMem(){ // Ustawia pamięć na aktualną wartość  2 usages
        memory = state;
    }

    public void useMem(){ // Dodaje zawartość pamięci do obecnej wartości
        state += memory;
    }

    public void addMem(){ // Dodaje wartość do pamięci  1 usage  szywat
        memory += state;
    }

    public void subMem(){ // Odejmuje wartość od pamięci  1 usage  szywat
        memory -= state;
    }

    public int getMem(){ // Zwraca pamięć  3 usages  szywat
        return memory;
    }
}
```

```
    public boolean getError(){  1 usage  new *
        return err;
    }
}
```

Testy:

```
package kalkulator;

import org.junit.*;

import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertTrue;

public class CalculatorTest {  szywat +1 *
    @Test  Szymon Wątorowski
    public void testAddOne(){
        // Arrange
        // sut = System Under Test
        Calculator sut = new Calculator();
        // Act
        sut.add(1);
        // Assert
        assertEquals( message: "0+1 = 1",  expected: 1, sut.getState());
    }

    @Test  Szymon Wątorowski
    public void testMultOneByTwo(){
        Calculator sut = new Calculator();
        sut.setState(1);
        sut.mult( value: 2);
        assertEquals( message: "1*2 = 2",  expected: 2, sut.getState());
    }
```

```
    @Test  szywat
    public void testAddNeg() {
        Calculator sut = new Calculator();
        sut.setState(1);
        sut.add(-1);
        assertEquals( message: "1+(-1) = 0",  expected: 0, sut.getState());
    }

    @Test  szywat
    public void testMultOneByNeg() {
        Calculator sut = new Calculator();
        sut.setState(1);
        sut.mult( value: -1);
        assertEquals( message: "1*(-1) = -1",  expected: -1, sut.getState());
    }

    @Test  szywat
    public void testMultOneByZero() {
        Calculator sut = new Calculator();
        sut.setState(1);
        sut.mult( value: 0);
        assertEquals( message: "1*0 = 0",  expected: 0, sut.getState());
    }

    @Test  szywat
    public void testSubOne() {
        Calculator sut = new Calculator();
        sut.setState(1);
        sut.sub( value: 1);
        assertEquals( message: "1-1 = 0",  expected: 0, sut.getState());
    }
```

```java
@Test  szywat
public void testSubNeg() {
    Calculator sut = new Calculator();
    sut.setState(1);
    sut.sub( value: -1);
    assertEquals( message: "1-(-1) = 2",  expected: 2, sut.getState());
}

@Test  szywat
public void testDivTenByTwo() {
    Calculator sut = new Calculator();
    sut.setState(10);
    sut.div( value: 2);
    assertEquals( message: "10 / 2 = 5",  expected: 5, sut.getState());
}

@Test  szywat
public void testDivTenByNeg() {
    Calculator sut = new Calculator();
    sut.setState(10);
    sut.div( value: -2);
    assertEquals( message: "10 / (-2) = -5",  expected: -5, sut.getState());
}

@Test  szywat
public void testSaveMemory() {
    Calculator sut = new Calculator();
    sut.setState(1);
    sut.saveMem(); // zawiera 1
    assertEquals( message: "Memory: 1",  expected: 1, sut.getMem());
}
```

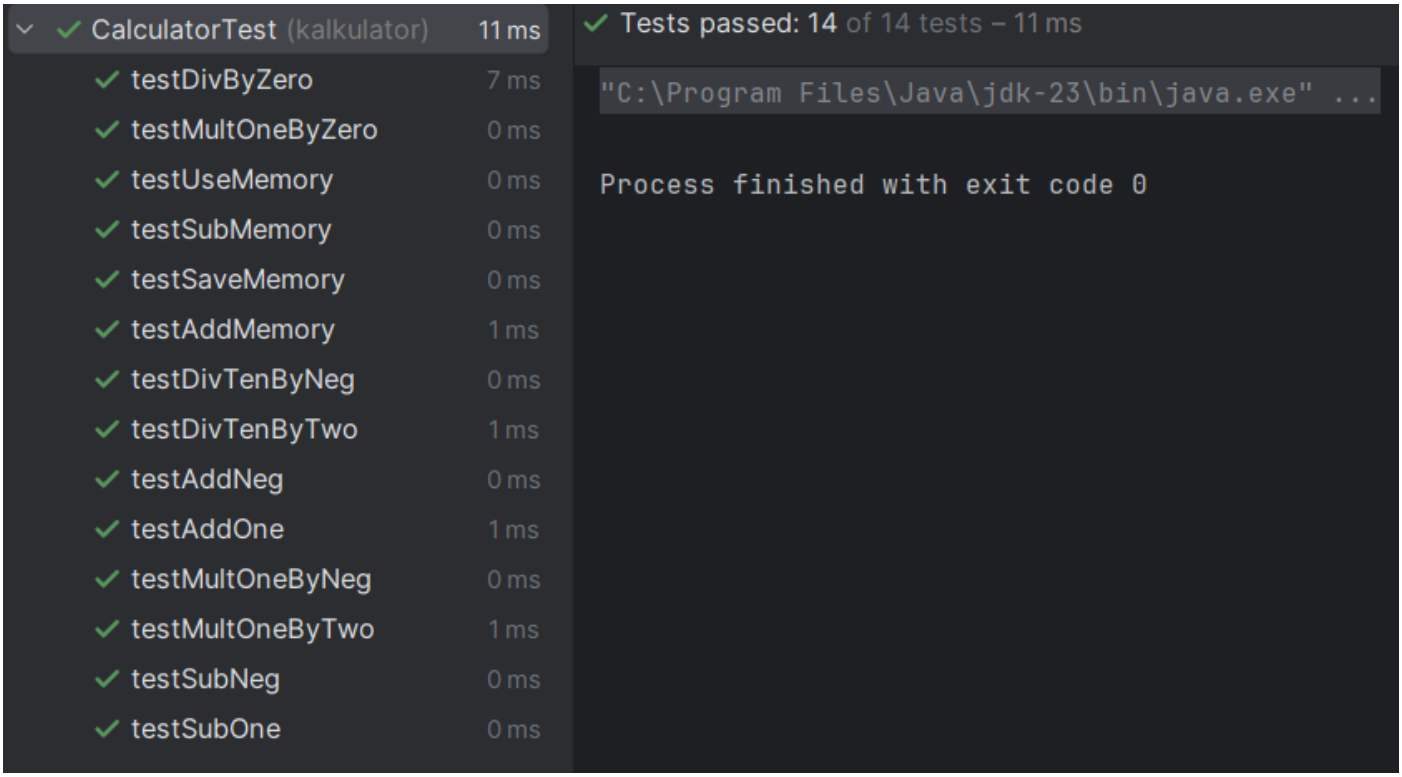
```java
@Test  szywat
public void testUseMemory() {
    Calculator sut = new Calculator();
    sut.setState(1);
    sut.saveMem(); // zawiera 1
    sut.useMem(); // dodaje 1
    assertEquals( message: "1 +(M 1) = 2",  expected: 2, sut.getState());
}

@Test  szywat
public void testAddMemory(){
    Calculator sut = new Calculator();
    sut.setState(1);
    sut.addMem();
    assertEquals( message: "(M 0)+1 = (M 1)",  expected: 1, sut.getMem());
}

@Test  szywat
public void testSubMemory(){
    Calculator sut = new Calculator();
    sut.setState(1);
    sut.subMem();
    assertEquals( message: "(M 0)-1 = (M -1)",  expected: -1, sut.getMem());
}

@Test  new *
public void testDivByZero() {
    Calculator sut = new Calculator();
    sut.setState(1);
    sut.div( value: 0);
    assertTrue( message: "1 / 0 = err", sut.getError());
```

Output:



Zadanie 5

Zadanie wykonałem na dwa sposoby – jeden z wykorzystaniem wcześniej wspomnianego kalkulatora i drugi poprzez napisanie nowego kodu programu. Poniżej przedstawiam obie metody:

Z wykorzystaniem kalkulatora obliczono wielomian:

$$\frac{x^5 + 4x^4 + 3x^3 - 2x^2 + 17}{x^2 - 7x + 1}$$

Dla wartości:

$$x \in \{-3, -2, -1, 1, 2, 3\}$$

Do obliczeń wykorzystano zaimplementowane już wcześniej metody, nową metodę *power(int value)* , która oblicza nam stan podniesiony do potęgi *value*, oraz kalkulator online do sprawdzenia poprawności wyników.

Wyniki kalkulatora online zamienione na *int*'y:

[0, 0, 1, -4, -14, -58]

Na początku tworzymy tablicę *int*'ów długości 6 (odpowiadającej ilości naszych x). Tworzymy nowy kalkulator *sut* oraz tablicę *x* zawierającą nasze wartości, dla których będziemy obliczać nasz wielomian. Tworzymy pętlę, która oblicza nam osobno *licznik* i *mianownik* z zapisywaniem danych w pamięci.

Cały kod został zapisany jako test do sprawdzenia poprawności działania.

Licznik:

```
@Test  new *
public void testWielomian(){
    int[] tablica = new int[6];
    Calculator sut = new Calculator();
    int[] x = {-3, -2, -1, 1, 2, 3};
    for (int i = 0; i < x.length; i++) {
        sut.setState(x[i]);
        sut.power( value: 5);
        sut.saveMem();

        sut.setState(x[i]);
        sut.power( value: 4);
        sut.mult( value: 4);
        sut.addMem();

        sut.setState(x[i]);
        sut.power( value: 3);
        sut.mult( value: 3);

        sut.addMem();

        sut.setState(x[i]);
        sut.power( value: 2);
        sut.mult( value: 2);
        sut.subMem();

        sut.setState(17);
        sut.addMem();

        int licznik = sut.getMem();
```

Ustawiamy stan na daną wartość x i podnosimy do potęgi, po czym przemnażamy uzyskaną wartość przez liczbę, która odpowiednio znajduje się przy danej potędze. Początkowo zapisujemy wartość w pamięci, ponieważ pamięć jest pusta. W każdym kolejnym etapie działania dodajemy nowy stan do pamięci i resetujemy wartość w stanie, żeby policzyć nową wartość dla kolejnej potęgi. Skumulowany wynik w pamięci przypisujemy do zmiennej *licznik* z użyciem metody *sut.getMem()*.

Mianownik:

```
sut.setState(x[i]);
sut.power( value: 2);
sut.saveMem();

sut.setState(x[i]);
sut.mult( value: -7);
sut.addMem();

sut.setState(1);
sut.addMem();

int mianownik = sut.getMem();
```

Mianownik obliczamy w sposób podobny do licznika – przypisujemy wartość x do stanu, podnosimy do potęgi, zapisujemy w pamięci (tym samym resetując wynik dla licznika w pamięci) i z każdym kolejnym działaniem resetujemy wartość stanu i dodajemy go do pamięci. Skumulowany wynik przypisujemy do zmiennej *mianownik*.

Dzielenie licznika przez mianownik:

```
sut.setState(licznik);
sut.div(mianownik);

int wynik = sut.getState();

tablica[i] = wynik;
```

Żeby sfinalizować nasze obliczenia dzielimy licznik przez mianownik. Ustawiamy stan na *licznik* i dzielimy go przez *mianownik*. Tym samym otrzymujemy końcowy wynik naszego wielomianu. Każdy pojedynczy wynik przypisujemy do tablicy.
Na zakończenie testu sprawdzamy czy wartości w tablicy zgadzają się z wartościami z kalkulatora online – daje to nam pozytywny wynik.

Dla tego samego wielomianu i tych samych wartości napisałem osobny program obliczający nasze zadanie.
Program składa się z:

- metody *obliczWielomian(int x)*, która oblicza nasz wielomian z pomocą *Math.pow*, które obliczyło potrzebne potęgi. Metoda zwraca *int*, przez co wymuszamy ten typ, ponieważ funkcja *Math.pow* zwraca wynik w formie *double*'a.
- metody *main*, w której wywołujemy metodę *obliczWielomian* dla podanych wcześniej wartości i z pomocą pętli zapisujemy wyniki w nowej tablicy *int*'ów.

Program wyświetla w terminalu tablicę z wynikami, które spełniają oczekiwania zadania.

Kod:

```
package kalkulator;

import java.util.Arrays;

public class Wielomian {  new *

    public static int obliczWielomian(int x) {  1 usage  new *
        return (int) (Math.pow(x, 5) + 4*(Math.pow(x, 4)) + 3*(Math.pow(x, 3)) -2*(Math.pow(x, 2)) + 17)
                / (int) (Math.pow(x, 2) -7*x + 1);
    }

    public static void main(String []args) {  new *
        int[] tablica = new int[6];
        int[] x = {-3, -2, -1, 1, 2, 3};
        for (int i = 0; i < x.length; i++) {

            tablica[i] = obliczWielomian(x[i]);
        }
        System.out.println(Arrays.toString(tablica));
    }
}
```

Output:

```
[0, 0, 1, -4, -14, -58]

Process finished with exit code 0
```