

Programowanie równoległe i rozproszone

Wykład 3

Programowanie współbieżne w Javie

Komunikacja poprzez przerwania

W języku Java istnieje możliwość komunikowania się wątków poprzez przerwania.

Wątek na listingu poniżej powtarza dziesięciokrotnie czekanie przez dwie sekundy (statyczna metoda `sleep()` klasy `Thread`).

Jeśli w czasie „drzemki” otrzyma sygnał `interrupt`, wówczas wykonanie metody `sleep(2000)` kończy się wyrzuceniem wyjątku `InterruptedException`, którego obsługa polega tutaj na wyświetleniu stosownego komunikatu oraz zakończeniu wykonywania pętli (instrukcja `break`).

```
public class Watek extends Thread {  
  
    public void run ( ) {  
  
        for ( int i =0; i <10; i++){  
            try{  
                Thread.sleep (2000) ;  
                System.out.println ( "Spalem 2 sekundy" ) ;  
            }  
            catch( InterruptedException e ) {  
                System.out.println ( "Dostalem sygnał interrupt" ) ;  
                break ;  
            } } } }  
}
```

Komunikacja poprzez przerwania

Listing demonstruje możliwości wysyłania przerwań.

W programie wątek metody main() po upływie pięciu sekund wysyła sygnał interrupt. Następnie czeka za zakończenie działania wątku reprezentowanego w zmiennej w (wywołanie metody `join()`) i wyświetla komunikat o zakończeniu pracy.

```
public class Main {  
  
    public static void main ( String [ ] args )  
        throws Exception {  
        Thread w = new Watek ( ) ;  
        w.start ( ) ;  
        Thread.sleep (5000) ;  
        w.interrupt ( ) ;  
        w.join ( ) ;  
        System.out.println ( "KONIEC" ) ;  
    }  
}
```

Definicja i własności semaforów

Rozważmy następującą definicję semaforów, która została sformułowana przez Dijkstre w odniesieniu do procesów współbieżnych.

Definicja: Semaforem S nazywamy zmienną przyjmującą wartości całkowite nieujemne. Jedynymi dopuszczalnymi dla semaforów operacjami są:

- $S.init(n)$: dopuszczalne jedynie przed pierwszym użyciem, jednokrotne nadanie semaforowi wartości początkowej n ,
- $S.wait()$: jeśli $S > 0$ wówczas $S := S - 1$, w przeciwnym razie wstrzymaj wykonanie procesu, który wywołał te operacje,
- $S.signal()$: w razie gdy są jakieś procesy wstrzymane w wyniku wykonania operacji $S.wait()$ na tym semaforze, wówczas wznów wykonanie jednego z nich, w przeciwnym razie $S := S + 1$.

Operacje $wait()$ i $signal()$ są operacjami atomowymi, czyli ich wykonania na danym semaforze nie mogą być ze sobą przeplatane.

Warto zaznaczyć, że operacja $signal()$ nie precyzuje, który z wątków ma być wznowiony. Najczęściej procesy oczekujące na wznowienie są kolejgowane.

Poza operacjami $wait()$ i $signal()$ nie są dozwolone żadne inne operacje. W szczególności nie ma możliwości testowania wartości semafora.

Semafory w Javie

Dla synchronizacji współbieżnych wątków w języku Java dysponujemy klasą **Semaphore** dostępną w pakiecie **java.util.concurrent**

Podstawowe funkcjonalności klasy **Semaphore**

Semaphore (int permits)

// tworzy obiekt "semafor" o zadanej wartości początkowej

Semaphore (int permits , boolean fair)

// tworzy obiekt "semafor" o zadanej wartości początkowej z gwarancją tego, że wątki wstrzymywane są kolejgowane

void acquire () throws InterruptedException

// operacja wait()

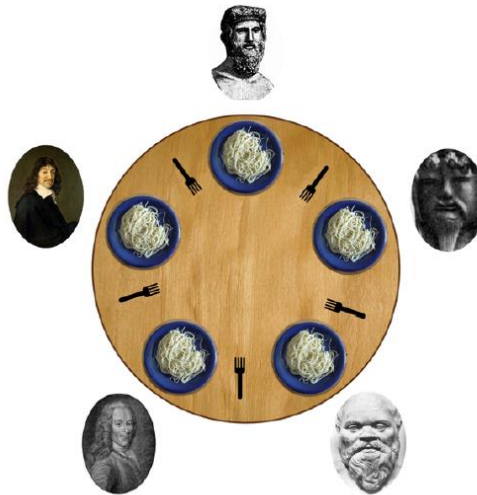
void acquireUninterruptibly ()

// operacja wait() bez wyrzucania wyjątku

void release()

// operacja signal()

Problem 5 filozofów



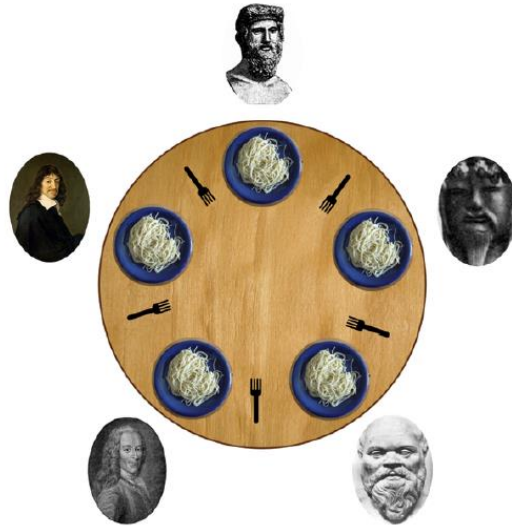
Przypuśćmy, że przy stole ucztuje pięciu filozofów **P0, P1, ..., P5** którzy działają w pętlach nieskończonych wykonując na przemian:

- sekcję lokalną **myślenie**
- sekcję krytyczną **jedzenie** do czego potrzebne są dwa widelce

Na stole umieszczono pięć widelców **f0, f1, ..., f4**, z których każdy leży po lewej stronie filozofa.

Filozof w chwili gdy ma rozpocząć jedzenie podnosi najpierw jeden widelec (po swojej lewej albo prawej stronie), a następnie drugi widelec.

Problem 5 filozofów



Problem polega takim zaprojektowaniu korzystania przez filozofów z widelców, aby spełnione były następujące własności:

1. filozof je wtedy i tylko wtedy, gdy ma dwa widelce,
2. dwóch filozofów nie może w tym samym czasie korzystać z tego samego widelca,
3. nie wystąpi zakleszczenie,
4. żaden filozof nie będzie zagłodzony,
5. rozwiązanie działa w przypadku braku współzawodnictwa.

```
import java.util.concurrent.Semaphore ;

public class Filozof extends Thread {

    static final int MAX=5;
    static Semaphore [] widelec = new Semaphore [MAX] ;
    int mojNum;

    public Filozof ( int nr ) {
        mojNum=nr ;
    }

    public void run ( ) {
        while ( true ) {
            // myślenie
            System.out.println ( "Mysle | " + mojNum) ;
            try {
                Thread.sleep ( ( long ) (7000 * Math.random( ) ) ) ;
            } catch ( InterruptedException e ) {
            }
            widelec [mojNum].acquireUninterruptibly ( ) ; //przechwycenie L widelca
            widelec [ (mojNum+1)%MAX].acquireUninterruptibly ( ) ; //przechwycenie P widelca
            // jedzenie
            System.out.println ( "Zaczyna jesc "+mojNum) ;
            try {
                Thread.sleep ( ( long ) (5000 * Math.random( ) ) ) ;
            } catch ( InterruptedException e ) {
            }
            System.out.println ( "Konczy jesc "+mojNum) ;

            widelec [mojNum].release ( ) ; //zwolnienie L widelca
            widelec [ (mojNum+1)%MAX].release ( ) ; //zwolnienie P widelca
        }
    }
}
```

Na listingu przedstawiono próbę rozwiązania problemu. Każdemu filozofowi odpowiada jeden wątek, zaś rolę **widelców** pełnią **semafony**.

Żaden widelec nie jest nigdy trzymany jednocześnie przez dwóch filozofów.

W rozwiązaniu może jednak wystąpić zakleszczenie. Istotnie, w sytuacji, gdy każdy z filozofów chwyci jednocześnie swój widelec (po lewej stronie), żaden z filozofów nie będzie mógł rozpocząć jedzenia.

Rozwiązaniem tego problemu będzie ograniczenie do czterech liczby filozofów trzymających jednocześnie widelce.

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

    for ( int i =0; i<MAX; i++) {
        widelec [ i ]=new Semaphore ( 1 ) ;
    }
    for ( int i =0; i<MAX; i++) {
        new Filozof(i).start();
    }
}
```


Problem uczujących filozofów z niesymetrycznym sięganiem po widelce

Problem uczujących filozofów można rozwiązać również zamieniając kolejność sięgania po widelce jednego z filozofów.

Czterech spośród pięciu filozofów, najpierw sięga po widelec z lewej strony, a potem te z prawej, natomiast jeden filozof czynność tę wykonuje odwrotnie.

Rozwiązanie to przedstawiono na kolejnym slajdzie

```
import java.util.concurrent.Semaphore ;

public class Filozof extends Thread {

    static final int MAX=5;
    static Semaphore [] widelec = new Semaphore [MAX] ;

    int mojNum;

    public Filozof ( int nr ) {
        mojNum=nr ;
    }

    public void run ( ) {

        while ( true ) {

// myslenie
System.out.println ( "Mysle | " + mojNum) ;

            try {
                Thread.sleep ( ( long ) (5000 * Math.random( ) ) ) ;
            } catch ( InterruptedException e ) {
            }

            if (mojNum == 0) {
                widelec [ (mojNum+1)%MAX].acquireUninterruptibly ( ) ;
                widelec [mojNum].acquireUninterruptibly ( ) ;
            } else {
                widelec [mojNum].acquireUninterruptibly ( ) ;
                widelec [ (mojNum+1)%MAX].acquireUninterruptibly ( ) ;
            }
        }
    }
}
```

```
// jedzenie
System.out.println ( "Zaczyna jesc "+mojNum) ;
try {
    Thread.sleep ( ( long ) (3000 * Math.random( ) ) ) ;
} catch ( InterruptedException e ) {
}
System.out.println ( "Konczy jesc "+mojNum) ;

        widelec [mojNum].release ( ) ;
        widelec [ (mojNum+1)%MAX].release ( ) ;

    }
}

public static void main ( String [] args ) {

        for ( int i =0; i<MAX; i++) {
            widelec [ i ]=new Semaphore ( 1 ) ;
        }

        for ( int i =0; i<MAX; i++) {
            new Filozof(i).start();
        }

    }
}
```

Rzut monety w rozwiązaniu problemu uczujących Filozofów

Problem uczących filozofów jest rozwiązany poprawnie, jeśli każdy filozof:

- 1. O tym, który widelec podniesie jako pierwsza, zdecyduje rzutem monety,**
- 2. Podniesie wylosowany widelec (jeśli nie jest wolny to zaczeka na niego),**
- 3. Następnie sprawdzi czy drugi widelec jest wolny.**
 - jeśli tak, to może jeść,**
 - jeśli natomiast drugi widelec nie jest wolny to odkłada widelec, który już trzyma i podejmuje kolejną próbę jedzenia, ponownie rzucając monetą.**

Programowanie równoległe i rozproszone

```
import java.util.Random;
import java.util.concurrent.Semaphore ;

public class Filozof extends Thread {

    static final int MAX=5;
    static Semaphore [] widelec = new Semaphore [MAX] ;
    int mojNum;
    Random losuj ;
    public Filozof ( int nr ) {
        mojNum=nr ;
        losuj = new Random(mojNum) ;
    }
    public void run ( ) {
        while ( true ) {
            // myslenie
            System.out.println ( "Mysle ! " + mojNum) ;

            try {
                Thread.sleep ( ( long ) (5000 * Math.random( ) ) ) ;
            } catch ( InterruptedException e ) {
            }

            int strona = losuj.nextInt ( 2 ) ;
            boolean podnioslDwaWidelce = false ;
            do {
                if ( strona == 0) {
                    widelec [mojNum].acquireUninterruptibly ( ) ;

                    if( ! ( widelec [ (mojNum+1)%MAX].tryAcquire ( ) ) ) {
                        widelec[mojNum].release ( ) ;
                    } else {
                        podnioslDwaWidelce = true ;
                    }
                }
            }
```

```
                } else {
                    widelec[(mojNum+1)%MAX].acquireUninterruptibly ( ) ;

                    if ( ! (widelec[mojNum].tryAcquire ( ) ) ) {
                        widelec[(mojNum+1)%MAX].release ( ) ;
                    } else {
                        podnioslDwaWidelce = true ;
                    }
                } while ( podnioslDwaWidelce == false ) ;

                System.out.println ( "Zaczyna jesc "+mojNum) ;
                try {
                    Thread.sleep ( ( long ) (3000 * Math.random( ) ) ) ;
                } catch ( InterruptedException e ) {
                }
                System.out.println ( "Konczy jesc "+mojNum) ;

                widelec [mojNum].release ( ) ;
                widelec [ (mojNum+1)%MAX].release ( ) ;

            }
        }

        public static void main ( String [] args ) {

            for ( int i =0; i<MAX; i++) {
                widelec [ i ]=new Semaphore ( 1 ) ;
            }
            for ( int i =0; i<MAX; i++) {
                new Filozof(i).start();
            }
        }
    }
}
```

Symulacja lotniska

```
import java.util.Random;

public class Samolot extends Thread {
    //definicja stanu samolotu
    static int LOTNISKO=1;
    static int START=2;
    static int LOT=3;
    static int KONIEC_LOTU=4;
    static int KATASTROFA=5;
    static int TANKUJ=1000;
    static int REZERWA=500;

    //zmienne pomocnicze
    int numer;
    int paliwo;
    int stan;
    Lotnisko l;
    Random rand;

    public Samolot(int numer, int paliwo, Lotnisko l){
        this.numer=numer;
        this.paliwo=paliwo;
        this.stan=LOT;
        this.l=l;
        rand=new Random();
    }
}
```

Programowanie równoległe i rozproszone

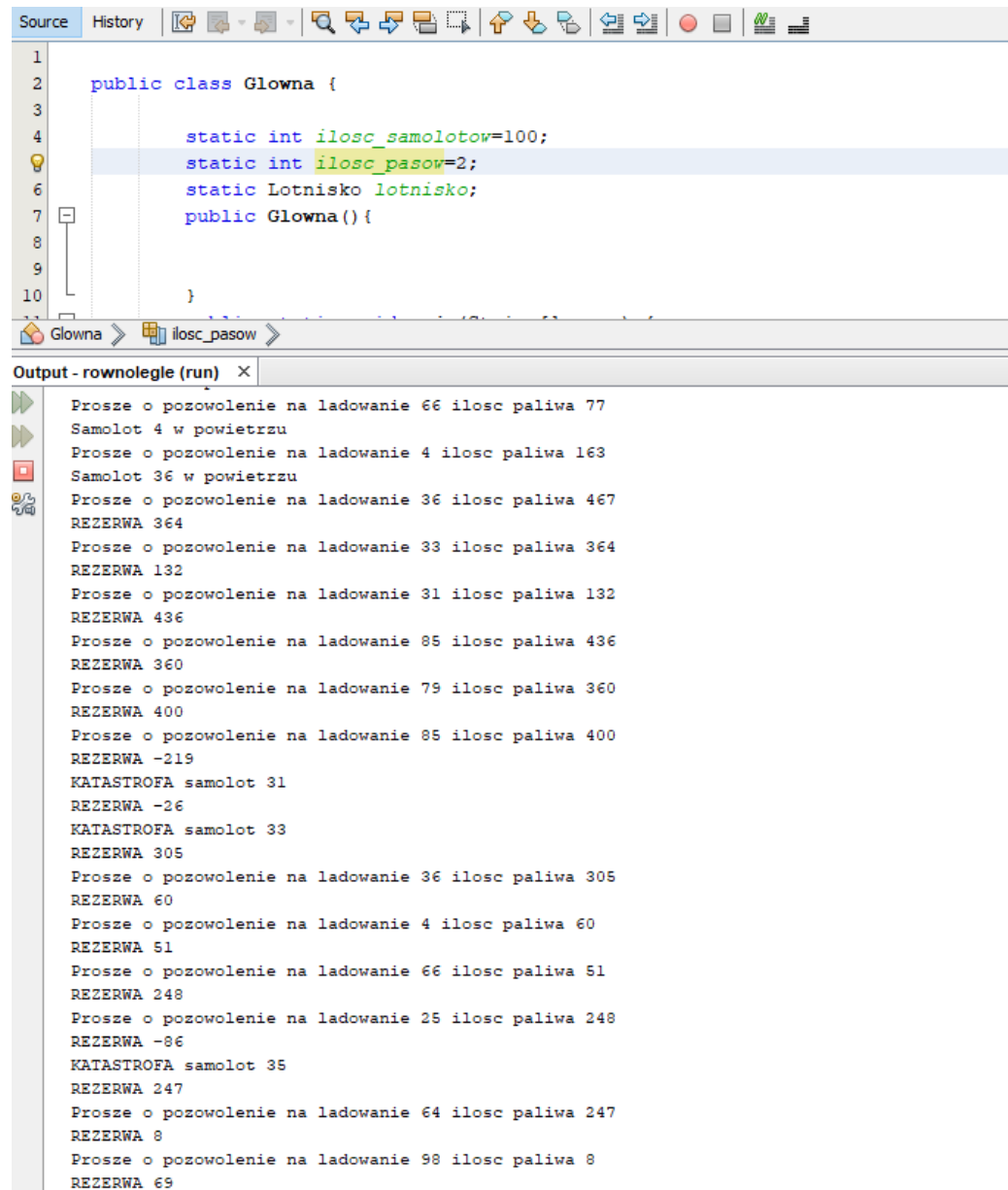
```
public void run(){
    while(true){
        if(stan==LOTNISKO){
            if(rand.nextInt(2)==1){
                stan=START;
                paliwo=TANKUJ;
                System.out.println(„Na lotnisku prosze o pozwolenie na start, samolot "+numer);
                stan=l.start(numer);
            }
            else{
                System.out.println("Postoje sobie jeszcze troche");
            }
        }
        else if(stan==START){
            System.out.println("Wystartowalem, samolot "+numer);
            stan=LOT;
        }
        else if(stan==LOT){
            paliwo=rand.nextInt(500);
            System.out.println("Samolot "+numer+" w powietrzu");
            if(paliwo<=REZERWA){
                stan=KONIEC_LOTU;
            }
            else try{
                sleep(rand.nextInt(1000));
            }
            catch (Exception e){}
        }
        else if(stan==KONIEC_LOTU){
            System.out.println("Prosze o pozowolenie na ladowanie "+numer+" ilosc paliwa "+paliwo);
            stan=l.laduj();
            if(stan==KONIEC_LOTU){
                paliwo=rand.nextInt(500);
                System.out.println("REZERWA "+paliwo);
                if(paliwo<=0) stan=KATASTROFA;
            }
        }
        else if(stan==KATASTROFA){
            System.out.println("KATASTROFA samolot "+numer);
            l.zmniejsz();
        }
    }
}
```

```
public class Lotnisko {
    static int LOTNISKO=1;
    static int START=2;
    static int LOT=3;
    static int KONIEC_LOTU=4;
    static int KATASTROFA=5;

    int ilosc_pasow;
    int ilosc_zajetych;
    int ilosc_samolotow;
    Lotnisko(int ilosc_pasow,int ilosc_samolotow){
        this.ilosc_pasow=ilosc_pasow;
        this.ilosc_samolotow=ilosc_samolotow;
        this.ilosc_zajetych=0;
    }
    synchronized int start(int numer){
        ilosc_zajetych--;
        System.out.println("Pozwolenie na start samolotowi "+numer);
        return START;
    }
    synchronized int laduj(){
        try{
            Thread.currentThread().sleep(1000);
        }
        catch(Exception ie){
        }
        if(ilosc_zajetych<ilosc_pasow){
            ilosc_zajetych++;
            System.out.println("Pozwolenie ladowanie na pasie "+ilosc_zajetych);
            return LOTNISKO;
        }
        else
            {return KONIEC_LOTU;}
    }
    synchronized void zmniejsz(){
        ilosc_samolotow--;
        System.out.println("ZABILEM");
        if(ilosc_samolotow==ilosc_pasow) System.out.println(„Ilosc samolotow jaka sama jak pasow");
    }
}
```

Programowanie równoległe i rozproszone

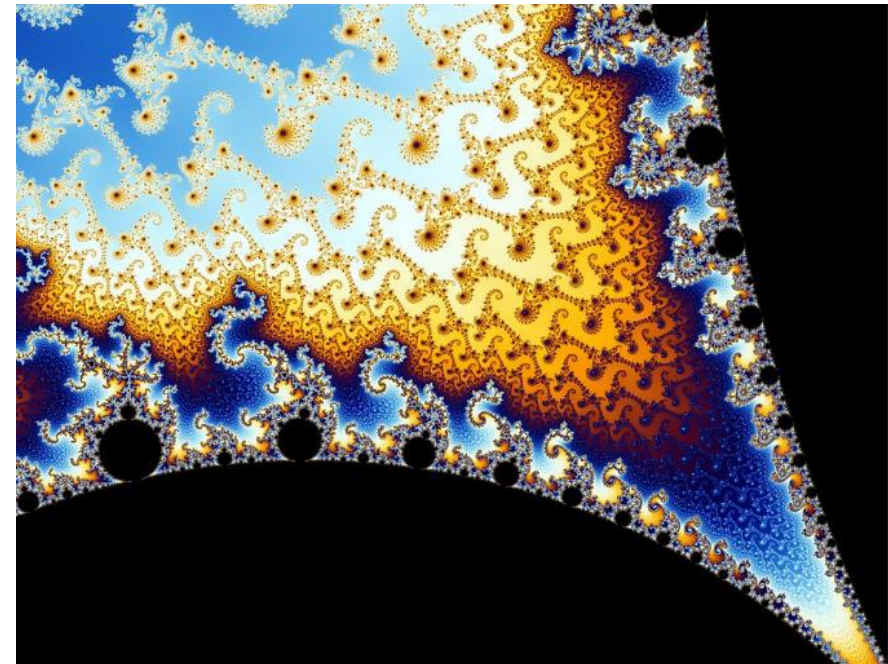
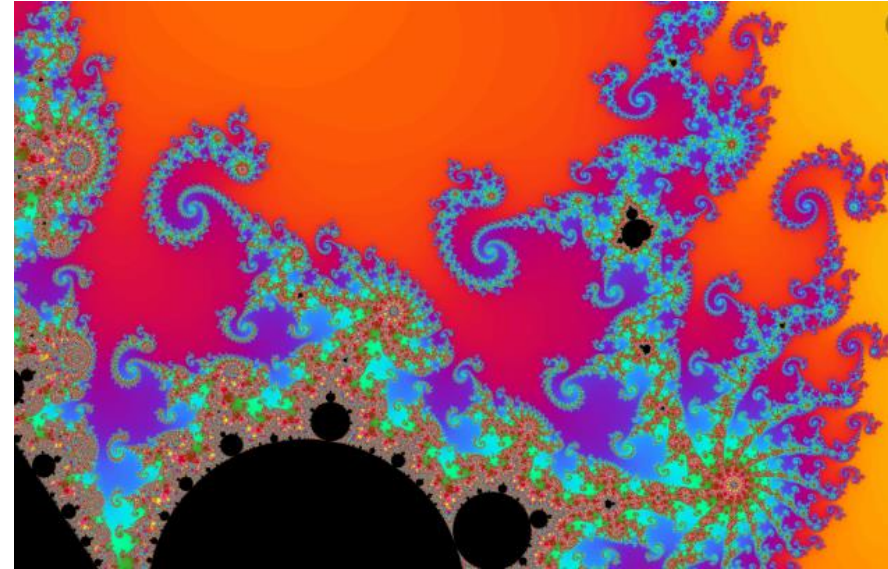
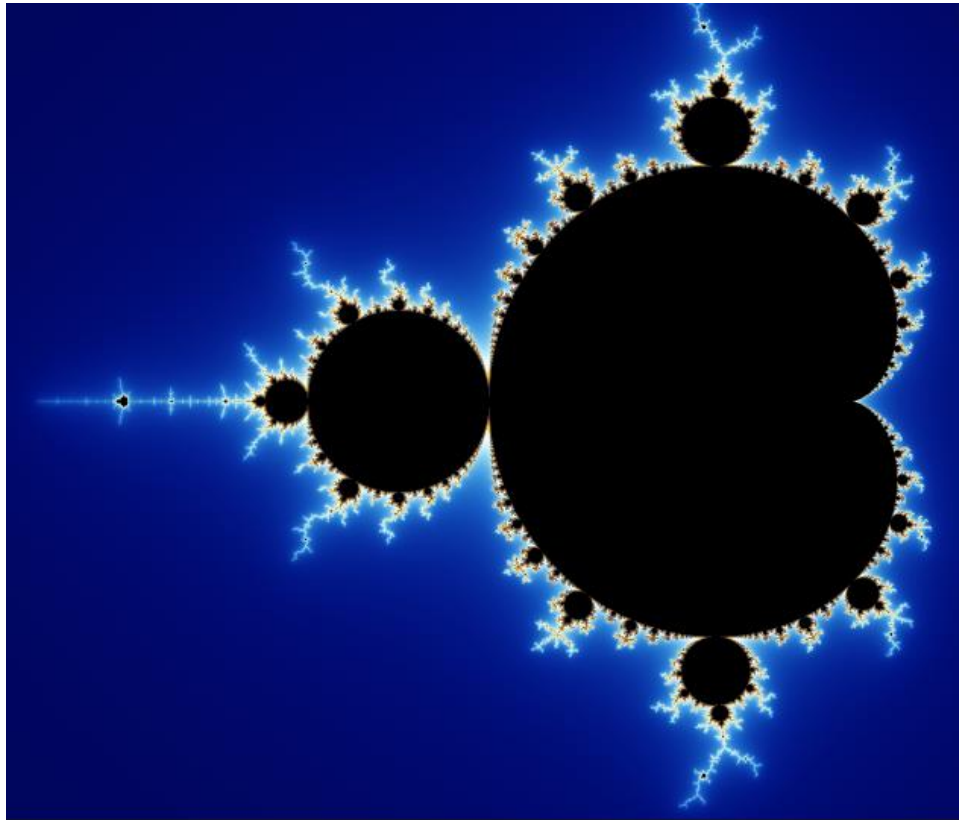
```
public class Glowna {  
  
    static int ilosc_samolotow=10;  
    static int ilosc_pasow=5;  
    static Lotnisko lotnisko;  
  
    public Glowna(){ }  
    public static void main(String[] args) {  
        lotnisko=new Lotnisko(ilosc_pasow, ilosc_samolotow);  
        for(int i=0;i<ilosc_samolotow;i++)  
            new Samolot(i,2000,lotnisko).start();  
    }  
}
```



The screenshot shows an IDE with a Java source file and its output. The source code defines a `Glowna` class with static variables for the number of planes and passengers, and a `main` method that creates a `Lotnisko` object and starts a loop of `Samolot` objects. The output window shows the execution of these objects, displaying messages like "Proszę o pozwolenie na ladowanie" and "REZERWA" with associated fuel and passenger counts.

```
Source History [Icons]
1
2 public class Glowna {
3
4     static int ilosc_samolotow=10;
5     static int ilosc_pasow=2;
6     static Lotnisko lotnisko;
7     public Glowna() {
8
9
10    }
11
12 }
13
14 Glowna
15 ilosc_pasow
16
Output - rownolegle (run) X
Proszę o pozwolenie na ladowanie 66 ilosc paliwa 77
Samolot 4 w powietrzu
Proszę o pozwolenie na ladowanie 4 ilosc paliwa 163
Samolot 36 w powietrzu
Proszę o pozwolenie na ladowanie 36 ilosc paliwa 467
REZERWA 364
Proszę o pozwolenie na ladowanie 33 ilosc paliwa 364
REZERWA 132
Proszę o pozwolenie na ladowanie 31 ilosc paliwa 132
REZERWA 436
Proszę o pozwolenie na ladowanie 85 ilosc paliwa 436
REZERWA 360
Proszę o pozwolenie na ladowanie 79 ilosc paliwa 360
REZERWA 400
Proszę o pozwolenie na ladowanie 85 ilosc paliwa 400
REZERWA -219
KATASTROFA samolot 31
REZERWA -26
KATASTROFA samolot 33
REZERWA 305
Proszę o pozwolenie na ladowanie 36 ilosc paliwa 305
REZERWA 60
Proszę o pozwolenie na ladowanie 4 ilosc paliwa 60
REZERWA 51
Proszę o pozwolenie na ladowanie 66 ilosc paliwa 51
REZERWA 248
Proszę o pozwolenie na ladowanie 25 ilosc paliwa 248
REZERWA -86
KATASTROFA samolot 35
REZERWA 247
Proszę o pozwolenie na ladowanie 64 ilosc paliwa 247
REZERWA 8
Proszę o pozwolenie na ladowanie 98 ilosc paliwa 8
REZERWA 69
```


Zbiór Mandelbrota



Po raz pierwszy pojęcie fraktala zostało użyte przez Benoit Mandelbrota w latach 70-tych XX wieku.

Zbiór Mandelbrota

Po raz pierwszy pojęcie fraktala zostało użyte przez Benoit Mandelbrota w latach 70-tych XX wieku.



Po raz pierwszy pojęcie fraktala zostało użyte przez Benoit Mandelbrota w latach 70-tych XX wieku.

Po łacinie **fractus** oznacza podzielny, ułamkowy, cząstkowy. Nazwa ta nie ma ścisłej matematycznej definicji.

Oznacza ona obiekty, które mają nietrywialną strukturę w każdej skali oraz są samopodobne - czyli każda ich część przypomina całość.

Mandelbrot prowadził badania przy pomocy komputera. Pierwsze obrazy zbioru opublikował w roku 1980.

Zbiór Mandelbrota

By zdefiniować zbiór Mandelbrota, zdefiniujemy najpierw dla danego punktu p na płaszczyźnie zespolonej nieskończony ciąg liczb zespolonych z_0, z_1, z_2, \dots o wartościach zdefiniowanych następująco:

$$z_0=0$$

$$z_{n+1}=z_n^2 + p$$

Definiujemy jako zbiór liczb zespolonych p takich, że zdefiniowany powyżej ciąg nie dąży do nieskończoności.

Fraktalem jest brzeg tego zbioru.

Zbiór Mandelbrota

W praktyce by narysować fraktale oblicza się kolejne przybliżenia zbioru, które oznacza się różnymi kolorami.

I tak kolejne przybliżenia zdefiniujemy jako zbiór liczb zespolonych p takich, że:

- 1 przybliżenie: wszystkie punkty
- 2 przybliżenie: $|z_1| < 2$
- 3 przybliżenie: $|z_1| < 2$ oraz $|z_2| < 2$
- 4 przybliżenie: $|z_1| < 2$ oraz $|z_2| < 2$ oraz $|z_3| < 2$
- ...
- n-te przybliżenie: $|z_1| < 2$ oraz $|z_2| < 2$, ... $|z_{n-1}| < 2$

Zbiór Mandelbrota

Zatem funkcję obliczającą z jakim maksymalnym przybliżeniem dany punkt p należy do zbioru Mandelbrota możemy zdefiniować następująco
(gdzie *maxIter* to maksymalne przybliżenie z jakim chcemy wyznaczać zbiór):

```
przyblizenie(p)
begin
  iter := 0;
  z := 0;

  repeat
    iter := iter + 1;
    z = z^2 + p;
  until (|z| < 2) and (iter < maxIter)

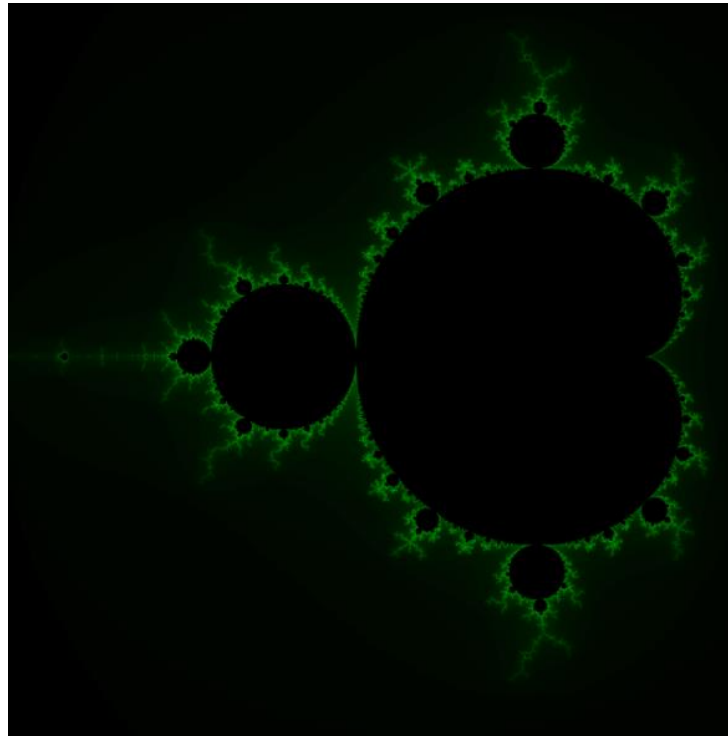
  przyblizenie = iter;
end;
```

Zbiór Mandelbrota

Dla kolejnych punktów na płaszczyźnie, obliczamy przybliżenia zgodnie z podanym algorytmem i wzorami.

Oś X oznacza wartości rzeczywiste, natomiast os Y wartości urojone.

Przedstawiając kolejne przybliżenia na płaszczyźnie (lewy górny róg ma współrzędne - $2.0 + -1.25i$, dolny prawy róg ma współrzędne $0.5 + 1.25i$) i oznaczając je różnymi kolorami otrzymujemy wynik - zbiór Mandelbrota,



- obliczenia, jakie należy wykonać w celu wygenerowania takiego zbioru, sprowadzają się do iteracyjnego rozwiązywania równania, którego parametrami są punkty płaszczyzny zespolonej

$$z_0=0$$

$$z_{n+1}=z_n^2+p$$

- aby obliczyć jedną iterację powyższego równania, należy bieżącą wartość podnieść do kwadratu i dodać stałą C

- punkty, dla których ciąg rozwiązań równania dąży do nieskończoności nie należą do zbioru

- natomiast punkty, dla których ciąg rozwiązań równania nie dąży do nieskończoności, należą do zbioru


```
import java.awt.Color;
import java.awt.image.BufferedImage;

import javax.imageio.ImageIO;

import java.io.File;

public class ParallelMandelbrot extends Thread {

    final static int N = 4096;
    final static int CUTOFF = 100;

    static int[][] set = new int[N][N];

    public static void main(String[] args) throws Exception {

        // Calculate set
        long startTime = System.currentTimeMillis();

        ParallelMandelbrot thread0 = new ParallelMandelbrot(0);
        ParallelMandelbrot thread1 = new ParallelMandelbrot(1);
        ParallelMandelbrot thread2 = new ParallelMandelbrot(2);
        ParallelMandelbrot thread3 = new ParallelMandelbrot(3);

        thread0.start();
        thread1.start();
        thread2.start();
        thread3.start();

        thread0.join();
        thread1.join();
        thread2.join();
        thread3.join();
    }
}
```

```
long endTime = System.currentTimeMillis();
```

```
System.out.println(„Obliczenia zakończone w czasie " + (endTime - startTime) + " milisekund");
```

```
// wyświetlanie rusunku
```

```
BufferedImage img = new BufferedImage(N, N, BufferedImage.TYPE_INT_ARGB);
```

```
// Rysowanie pixeli
```

```
for (int i = 0; i < N; i++) {
```

```
    for (int j = 0; j < N; j++) {
```

```
        int k = set[i][j];
```

```
        float level;
```

```
        if (k < CUTOFF) {
```

```
            level = (float) k / CUTOFF;
```

```
        } else {
```

```
            level = 0;
```

```
        }
```

```
        Color c = new Color(0, level, 0); // zielony
```

```
        img.setRGB(i, j, c.getRGB());
```

```
    }
```

```
}
```

```
// zapis do pliku
```

```
ImageIO.write(img, "PNG", new File("Mandelbrot.png"));
```

```
}
```

```
int me;
```

```
public ParallelMandelbrot4(int me) {
```

```
    this.me = me;
```

```
}
```

```
public void run() {  
  
    int begin = 0, end = 0;  
  
    if (me == 0) {  
        begin = 0;  
        end = (N / 4) * 1;  
    }  
    else if (me == 1) {  
        begin = (N / 4) * 1;  
        end = (N / 4) * 2;  
    }  
    else if (me == 2) {  
        begin = (N / 4) * 2;  
        end = (N / 4) * 3;  
    }  
    else if (me == 3) {  
        begin = (N / 4) * 3;  
        end = N;  
    }  
}
```

```
for (int i = begin; i < end; i++) {  
    for (int j = 0; j < N; j++) {  
  
        double cr = (4.0 * i - 2 * N) / N;  
        double ci = (4.0 * j - 2 * N) / N;  
        double zr = cr, zi = ci;  
  
        int k = 0;  
        while (k < CUTOFF && zr * zr + zi * zi < 4.0) {  
            // z = c + z * z  
            double newr = cr + zr * zr - zi * zi;  
            double newi = ci + 2 * zr * zi;  
  
            zr = newr;  
            zi = newi;  
  
            k++;  
        }  
  
        set[i][j] = k;  
    }  
}
```