Programowanie równolegle i rozproszone

Wykład 3
Programowanie współbieżne w Javie

Definicja i własności semaforów

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

Definicja: Semaforem S nazywamy zmienną przyjmująca 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, ze operacja signal() nie precyzuje, który z wątków ma być wznowiony. Najczęściej procesy oczekujące na wznowienie są kolejkowane.

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

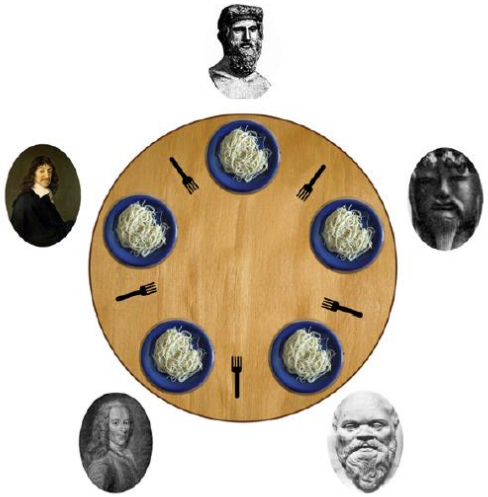
Semaphore (int permits , boolean fair)
// tworzy obiekt "semafor" o zadanej wartości początkowej z gwarancją tego, że wątki wstrzymywane są kolejkowane

void acquire ( ) throws InterruptedException
// operacja wait()

void acquireUninterruptibly ( )
// operacja wait() bez wyrzucania wyjątku

void release()
// operacja signal( )

Problem 5 filozofów



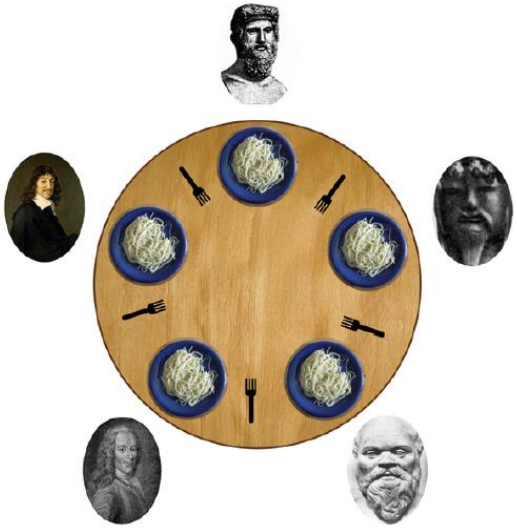
Przypuśćmy, ze przy stole ucztuje pięciu filozofów P0, P1, …, P4 którzy działają w pętlach nieskończonych wykonując na przemian:

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

Na stole umieszczono 5 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.**

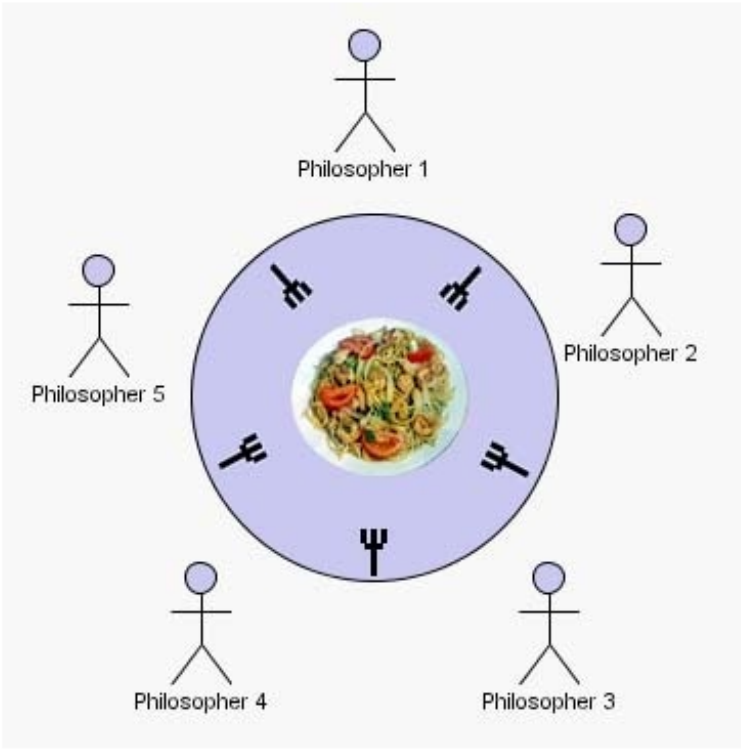
Każdy filozof myśli, a po pewnym czasie (gdy zgłodnieje) stara się jeść. Aby rozpocząć jedzenie, filozof musi podnieść oba widelce leżące obok jego talerza. Tym samym uniemożliwia jedzenie swoim sąsiadom. Po zaspokojeniu głodu filozof odkłada oba widelce na stół.

- dwóch filozofów nie korzystało jednocześnie z tego samego widelca (wzajemne wykluczanie),
- każdy filozof kończył jedzenie sam (niewywłaszczalność),
- nie nastąpiła sytuacja, w której każdy filozof trzyma jeden widelec i czeka na drugi (blokada),
- nie nastąpiło zagłodzenie filozofa czyli sytuacja, w której sąsiedzi filozofa zawsze będą korzystać z widelca.

Działanie filozofa przedstawia następujący program:

Proces Filozof(i)
powtarzaj cyklicznie:

1) myśli
2) siada przy stole na swoim miejscu
3) podnosi lewy widelec
4) podnosi prawy widelec
5) je
6) odkłada oba widelce
7) odchodzi od stołu



Program nie spełnia warunku bezpieczeństwa, gdyż zasoby dzielone (widelce) mogą być wykorzystywane jednocześnie przez dwóch filozofów.
Aby program spełniał warunek bezpieczeństwa, należy go uzupełnić o dodatkowe instrukcje oczekiwania.

Warunek bezpieczeństwa spełnia poniższy program :

Proces Filozof(i)
powtarzaj cyklicznie:

1) myśli
2) siada przy stole na swoim miejscu
3) czeka, aż lewy widelec będzie wolny
4) podnosi lewy widelec
5) czeka, aż prawy widelec będzie wolny
6) podnosi prawy widelec
7) je
8) odkłada oba widelce
9) odchodzi od stołu

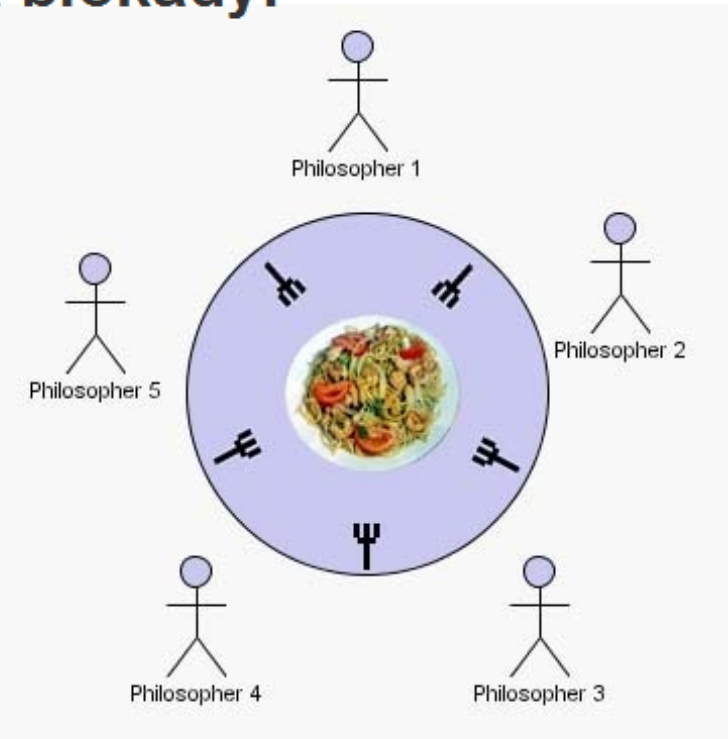
W programie występuje:
- wzajemne wykluczanie – widelec może być wykorzystywany w danej chwili tylko przez jednego filozofa,
- niewywłaszczalność – żaden filozof nie może odebrać widelca sąsiadowi,
- częściowy przydział – filozof podnosi najpierw jeden widelec, a następnie drugi widelec,
- jeśli każdy filozof podniesie lewy widelec w tym samym momencie, to wystąpi zamknięty łańcuch, w którym każdy filozof będzie oczekiwać na oddanie widelca przez poprzednika.

Program nie spełnia warunku żywotności, gdyż może wystąpić blokada.
Aby usunąć możliwość powstania blokady można wyeliminować częściowy przydział zasobów tzn. zapewnić, że filozof podnosi oba widelce jednocześnie.

Poniższy program spełnia warunek bezpieczeństwa oraz nie dopuszcza do powstania blokady:

Poniższy program spełnia warunek bezpieczeństwa oraz nie dopuszcza do powstania blokady:

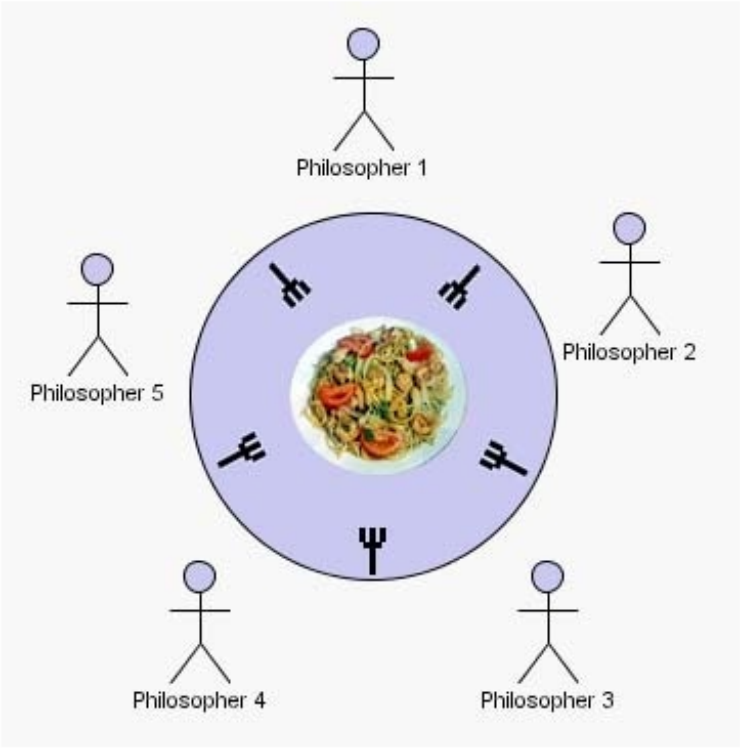
Proces Filozof(i)
powtarzaj cyklicznie:
1) myśli
2) siada przy stole na swoim miejscu
3) czeka, aż oba widelce będą wolne
4) podnosi oba widelce
5) je
6) odkłada oba widelce
7) odchodzi od stołu



Jeśli dwóch sąsiadów filozofa będzie na przemian trzymać widelce, to nastąpi zagłodzenie.
Program nie spełnia warunku żywotności.
Innym sposobem eliminacji możliwości powstania blokady jest niedopuszczenie do powstania zamkniętego łańcucha wzajemnego oczekiwania.
Można to osiągnąć poprzez ograniczenie liczby filozofów, którzy jednocześnie zasiadają przy stole.

W jadalni jest lokaj, który dba o to, by przy stole zawsze zasiadało nie więcej niż 4 filozofów.

Proces Filozof(i)
powtarzaj cyklicznie:
1) myśli
2) czeka, aż lokaj pozwoli dojść do stołu
3) siada przy stole na swoim miejscu
4) czeka, aż lewy widelec będzie wolny
5) podnosi lewy widelec
6) czeka, aż prawy widelec będzie wolny
7) podnosi prawy widelec
8) je
9) odkłada oba widelce
10) odchodzi od stołu



Program spełnia warunki bezpieczeństwa żywotności

```
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 ) (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 **semafory**.

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 ucztujących filozofów z niesymetrycznym sięganiem po widelce

Problem ucztujących 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 te 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();
        }

        }
}
```

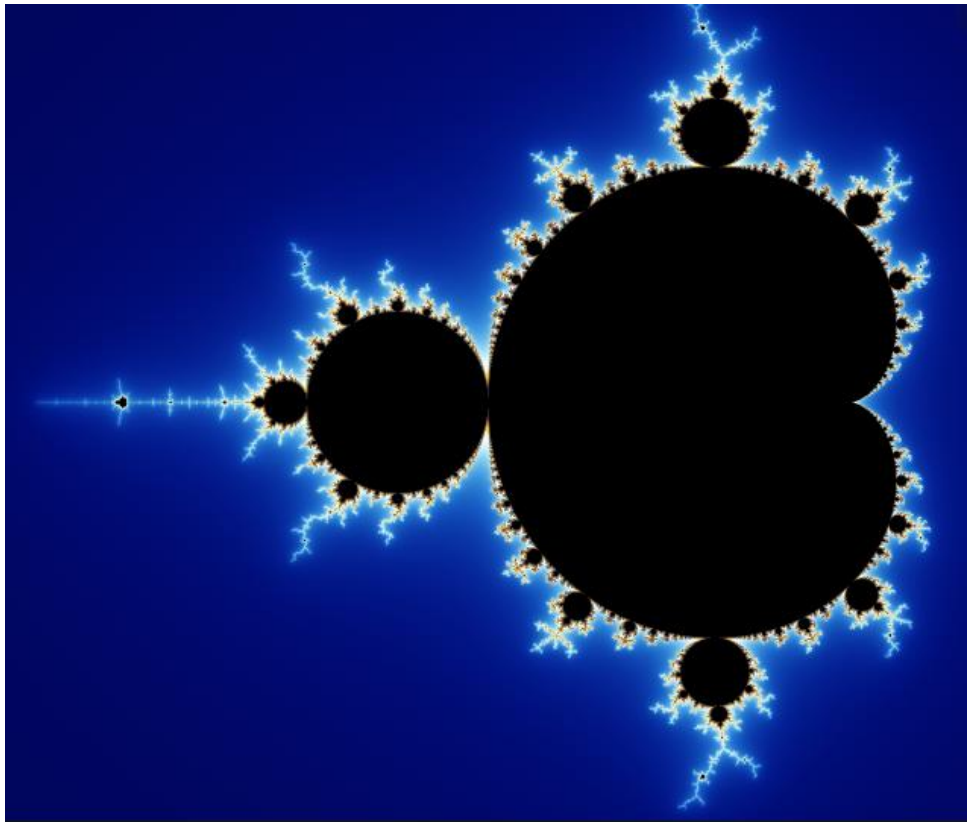
Rzut monety w rozwiązaniu problemu ucztujących Filozofów

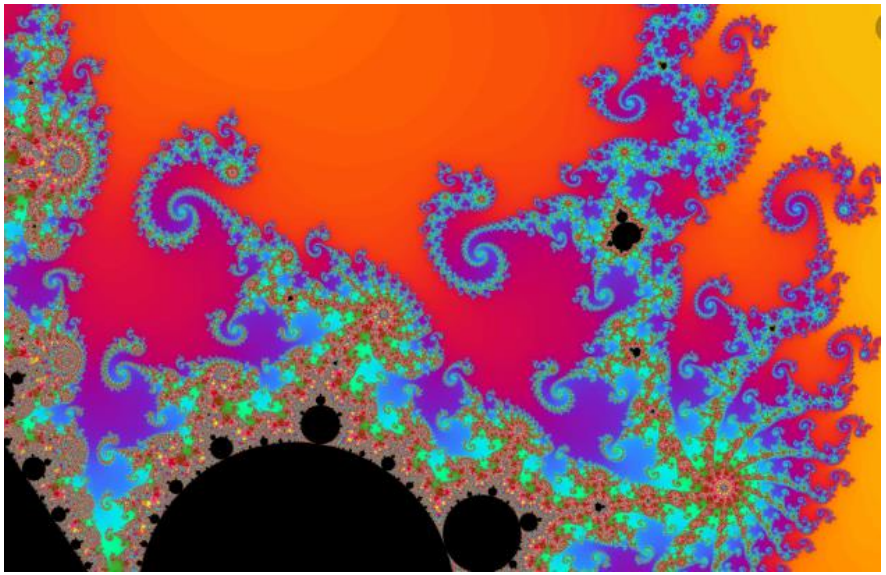
Problem ucztujących filozofów jest rozwiązany poprawnie, jeśli każdy filozof:

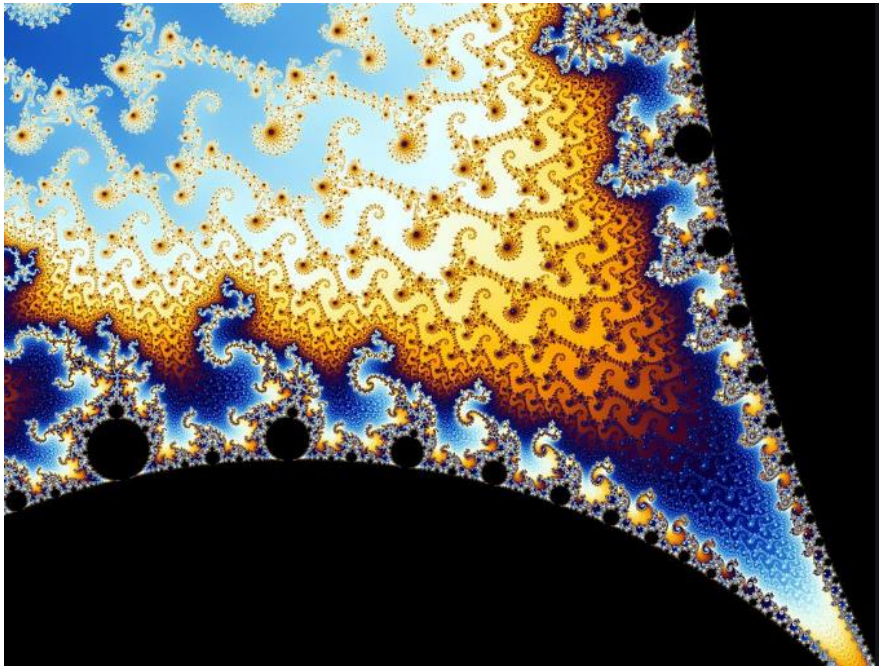
1. **O tym, który widelec podniesie jako pierwsza, zdecyduje rzutem monety,**
2. **Podniesie wylosowany widelec (jeśli nie jest wolny to zaczeka na niego),**
3. **Następnie sprawdzi czy drugi widelec jest wolny.**
   - **jeśli tak, to może jeść,**
   - **jeśli natomiast drugi widelec nie jest wolny to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

```
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 podniosłDwaWidelce = false ;
do {
 if ( strona == 0) {
 widelec [mojNum].acquireUninterruptibly ( ) ;

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

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

Zbiór Mandelbrota

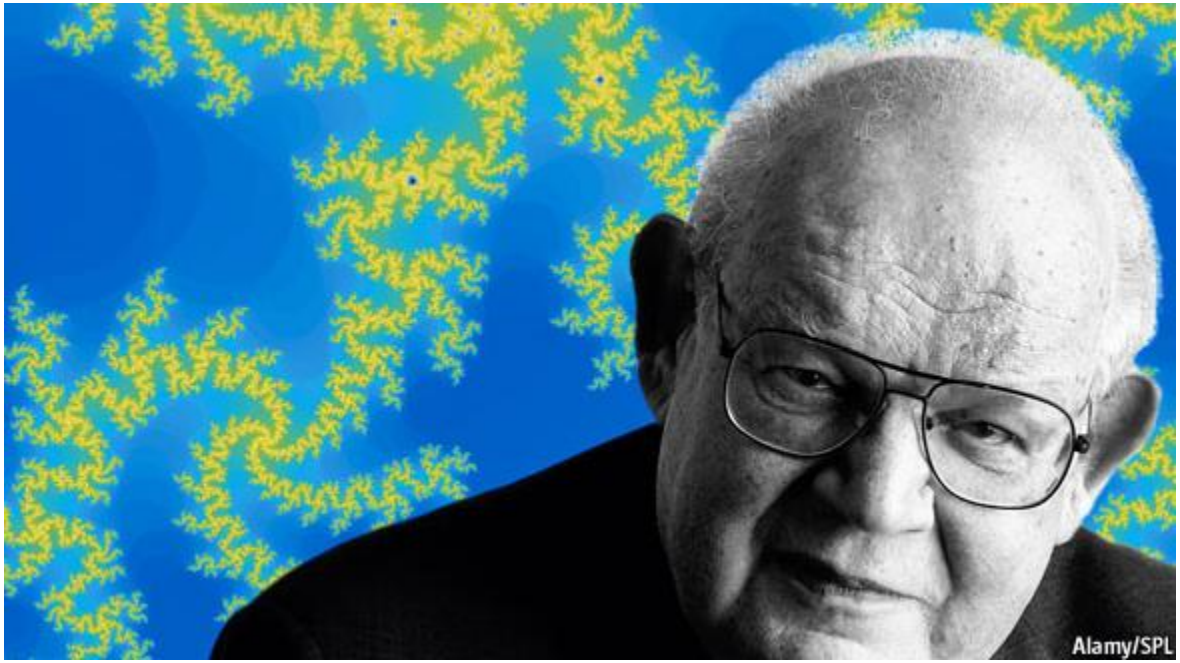






Po raz pierwszy pojęcie fraktala zostało użyte przez Benoit Mandelbrota w latach 70-tych XX wieku.

Zbiór Mandelbrota



- 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 pierwsze obrazy zbioru opublikował w roku 1980,

- Przykład https://www.youtube.com/watch?v=pCpLWbHVNhk

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, ... o wartościach zdefiniowanych następująco:

$$z_0=0$$

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

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:

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

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

  przyblizenie = iter;
end;
```

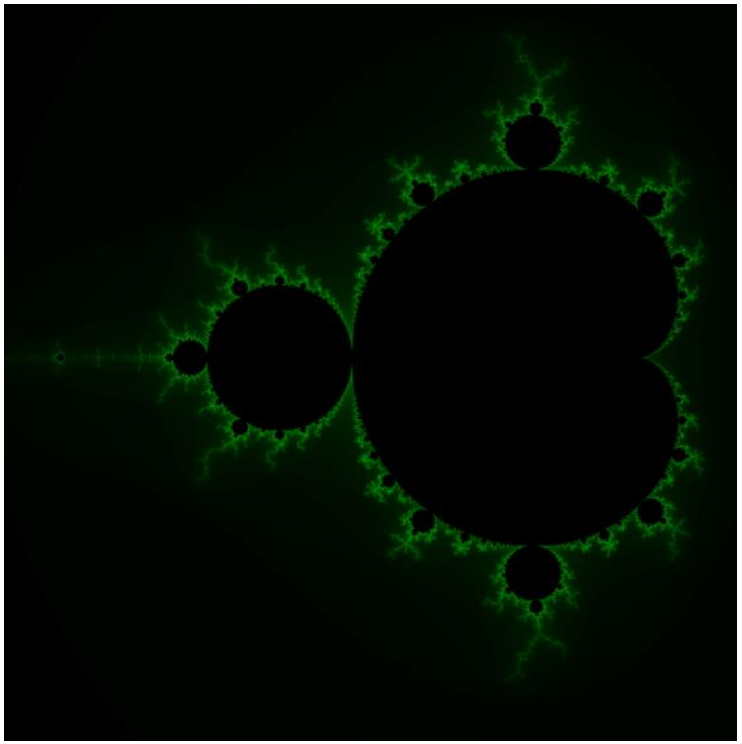
gdzie *maxIter* to maksymalne przybliżenie z jakim chcemy wyznaczać zbiór

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

$$z_0=0$$
$$z_{n+1}=z_n\text{^}2 + p$$

- 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;
//stala okreslająca czy szereg manderglora w aktualnym punkcje będzie nieskończony
final static int CUTOFF = 100;

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

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

   //ustawienie stopera liczącego czas obliczeń
   long startTime = System.currentTimeMillis();

   //ustawienie 4 wątków generujących fraktal w 4 ćwiartkach
   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();

   //czekanie głównego programu na zakończenie 4 wątków roboczych
   thread0.join();
   thread1.join();
   thread2.join();
   thread3.join();
```

```
    //zakończenie działania stopera i wyświetlenie czasu generowania fraktala
    long endTime = System.currentTimeMillis();
    System.out.println(„Obliczenia zakończone w czasie " + (endTime - startTime) + " millisekund");

    //ustawienie pliku graficznego, w którym zostanie wygenerowany fraktal
    BufferedImage img = new BufferedImage(N, N, BufferedImage.TYPE_INT_ARGB);

    //wstawianie pixeli do pliku graficznego
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {

            int k = set[i][j];

            float level;
            if (k < CUTOFF) {
                //pixel o wspolrzednych i,j należy do zbioru Manderbrota
                level = (float) k / CUTOFF;
            } else {
                //pixel o wspolrzednych i,j należy do zbioru Manderbrota
                level = 0;
            }
            //zapisywanie pixela (na zielono lub czarno)
            Color c = new Color(0, level, 0);  // zielony
            img.setRGB(i, j, c.getRGB());
        }
    }
    //zapis rysunku do pliku Mandelbrot.png
    ImageIO.write(img, "PNG", new File("Mandelbrot.png"));
}

int me;
//konstruktor, który ustawie numeracje wątków
public ParallelMandelbrot4(int me) {
    this.me = me;
}
```

```
//procedura wykonywana przez każdy z 4 wątków sprawdzająca czy dany punkt należy do zbioru Manderbrota
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++) {

            //przeskalowanie punktów cr i ci z dziedziny obrazka do
            //układu wspolrzednych kartezjanskich
            double cr = (4.0 * i - 2 * N) / N;
            double ci = (4.0 * j - 2 * N) / N;
            double zr = cr, zi = ci;

            int k = 0;
            //sprawdzanie czy szereg jest nieskonczony
            while (k < CUTOFF && zr * zr + zi * zi < 4.0) {
                // z = c + z * z
                double newr = cr + (zr*zr - zi*zi);
                double newi = ci + (zr*zi + zr*zi);

                zr = newr;
                zi = newi;

                k++;
            }

            set[i][j] = k;
        }
    }
}

}
```