PAMIĘĆ MASZYNY WIRTUALNEJ JAVY
PROGRAMOWANIE WSPÓŁBIEŻNE

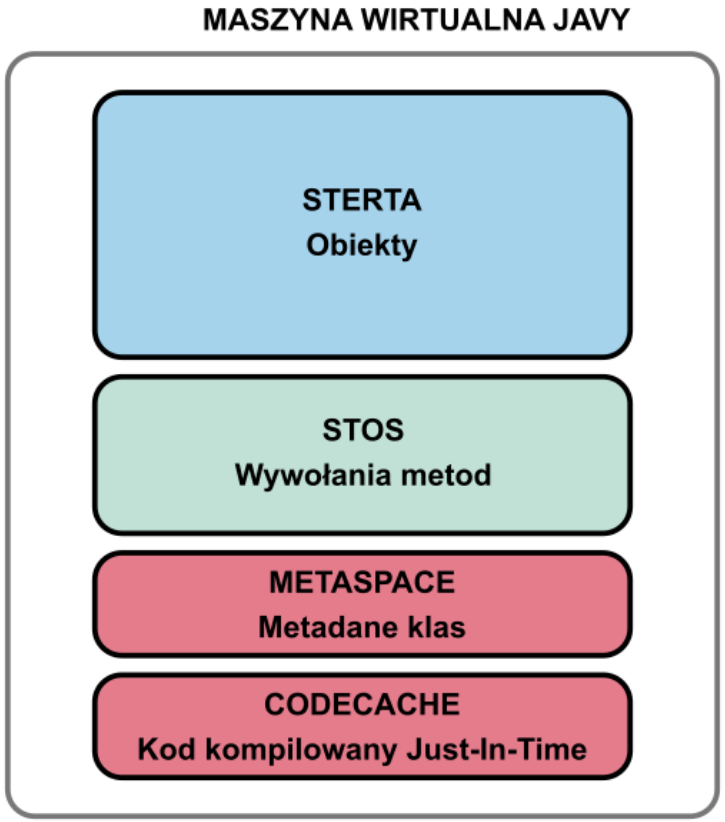
- Pamięć maszyny wirtualnej Javy: stos, sterta, Garbage Collector
- Współbieżność a równoległość
- Podstawy programowania współbieżnego: wątki, wyścig, synchronizacja

Wykład częściowo oparty na materiałach A. Jaskot

PAMIĘĆ MASZYNY WIRTUALNEJ JAVY

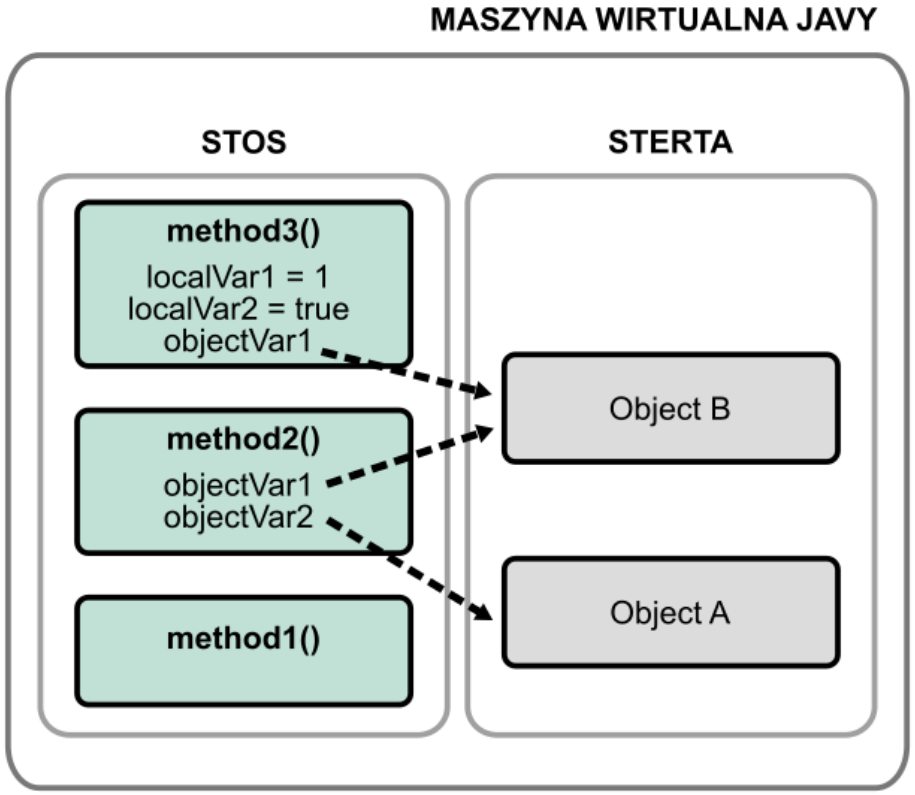
- maszyna wirtualna wykorzystuje określoną pamięć RAM, przydzieloną przez system operacyjny (ilość pamięci zależy od ustawień i systemu)

- maszyna wirtualna dynamicznie zarządza pamięcią w trakcie działania aplikacji, np. alokując miejsce na obiekty i usuwając nieużywane obiekty



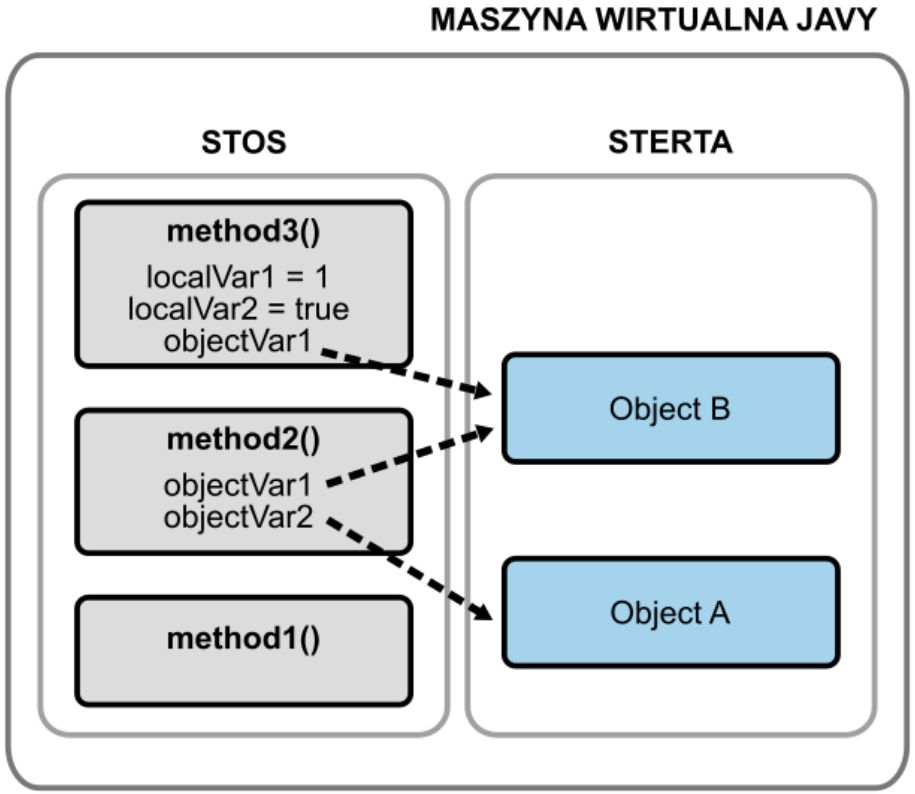
STOS (ang. Stack)

- obszar pamięci przechowujący wywołania metod w formie *ramek* (ang. frames)
- każda ramka zawiera zmienne lokalne, parametry danej metody oraz adres powrotu do miejsca, z którego metoda została wywołana
- dynamicznie alokowany i dealokowany
- działa na zasadzie kolejki LIFO (z ang. last in, first out)
- przekroczenie pamięci stosu powoduje wyjątek StackOverflowError



STERTA (ang. Heap)

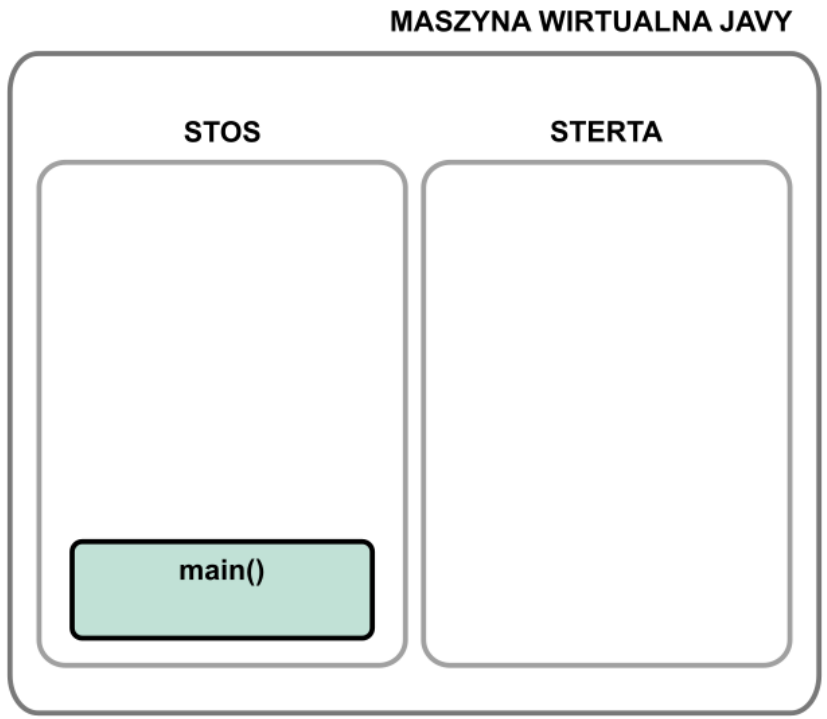
- obszar pamięci przechowujący obiekty
- zwykle znacznie większa niż stos
- składa się z kilku obszarów nazywanych *pokoleniami* (ang. generations)
- za dealokację nieużywanej pamięci odpowiada Garbage Collector
- przekroczenie pamięci sterty powoduje wyjątek OutOfMemoryError



STOS I STERTA W TRAKCIE DZIAŁANIA PROGRAMU

```
public class CalculatorTest {
    public static void main(String[] args) {
        Calculator calc = new Calculator();
        calc.sumAndShow(2, 3);
    }
}
```

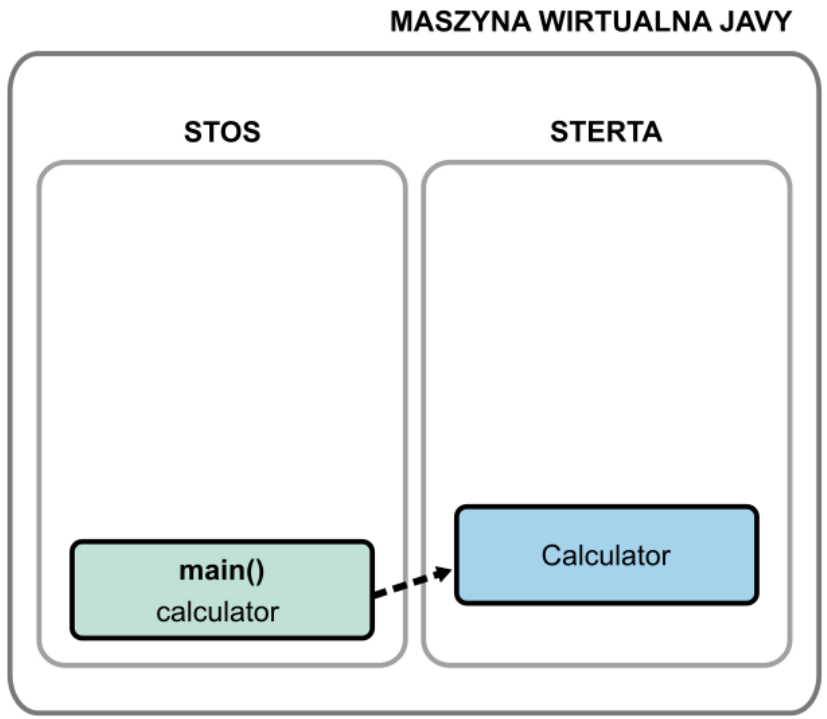
```
public class Calculator {

    public void sumAndShow(int a, int b) {
        int sum = sumAndReturn(a, b);
        System.out.println(sum);
    }

    public int sumAndReturn(int a, int b) {
        int sum = a + b;
        return sum;
    }
}
```



STOS I STERTA W TRAKCIE DZIAŁANIA PROGRAMU

```
public class CalculatorTest {
    public static void main(String[] args) {
        Calculator calc = new Calculator();
        calc.sumAndShow(2, 3);
    }
}
```

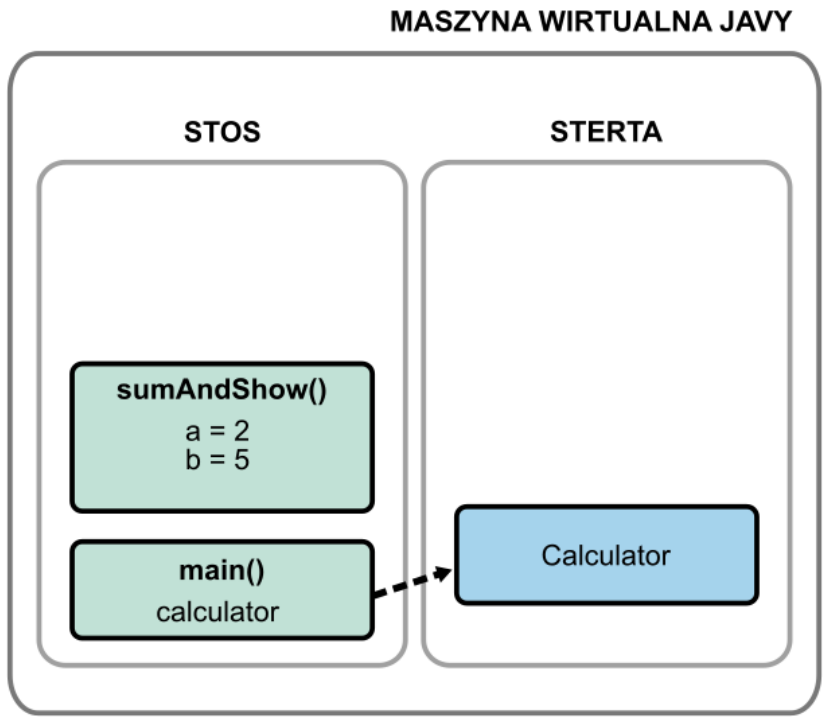
```
public class Calculator {

    public void sumAndShow(int a, int b) {
        int sum = sumAndReturn(a, b);
        System.out.println(sum);
    }

    public int sumAndReturn(int a, int b) {
        int sum = a + b;
        return sum;
    }
}
```



STOS I STERTA W TRAKCIE DZIAŁANIA PROGRAMU

```
public class CalculatorTest {
    public static void main(String[] args) {
        Calculator calc = new Calculator();
        calc.sumAndShow(2, 5);
    }
}
```

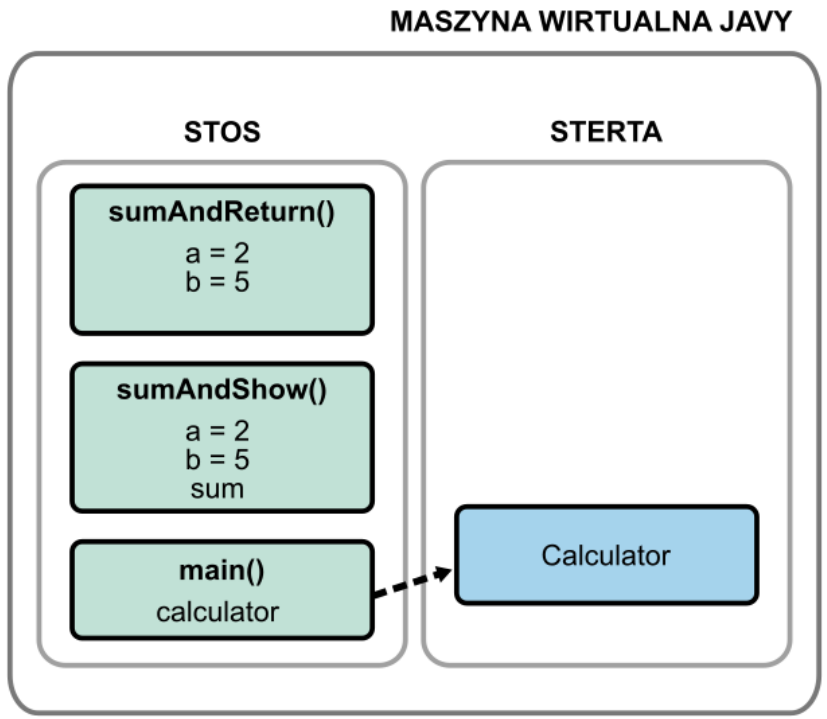
```
public class Calculator {

    public void sumAndShow(int a, int b) {
        int sum = sumAndReturn(a, b);
        System.out.println(sum);
    }

    public int sumAndReturn(int a, int b) {
        int sum = a + b;
        return sum;
    }
}
```



STOS I STERTA W TRAKCIE DZIAŁANIA PROGRAMU

```
public class CalculatorTest {
    public static void main(String[] args) {
        Calculator calc = new Calculator();
        calc.sumAndShow(2, 3);
    }
}
```

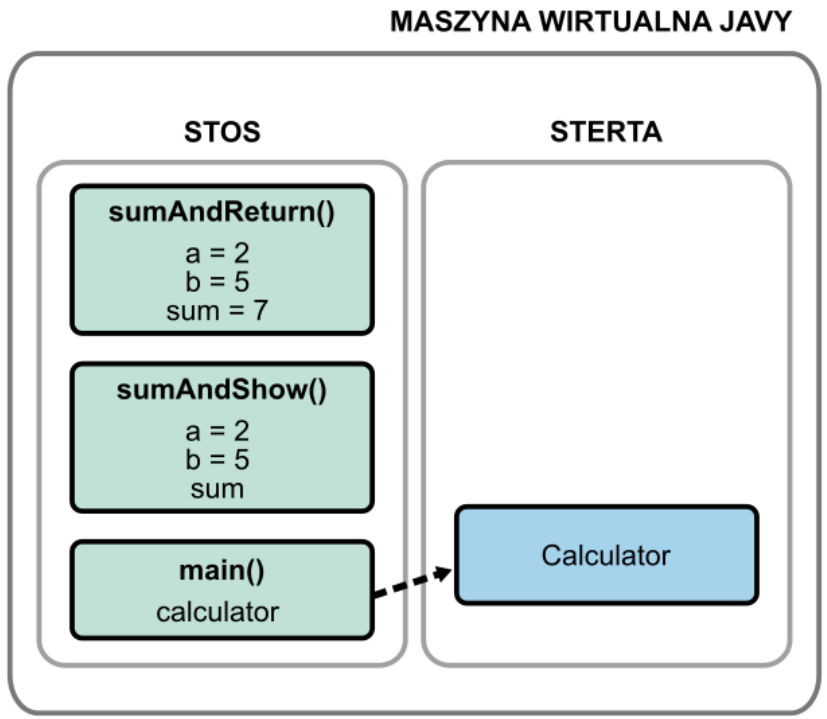
```
public class Calculator {

    public void sumAndShow(int a, int b) {
        int sum = sumAndReturn(a, b);
        System.out.println(sum);
    }

    public int sumAndReturn(int a, int b) {
        int sum = a + b;
        return sum;
    }
}
```



STOS I STERTA W TRAKCIE DZIAŁANIA PROGRAMU

```
public class CalculatorTest {
    public static void main(String[] args) {
        Calculator calc = new Calculator();
        calc.sumAndShow(2, 3);
    }
}
```

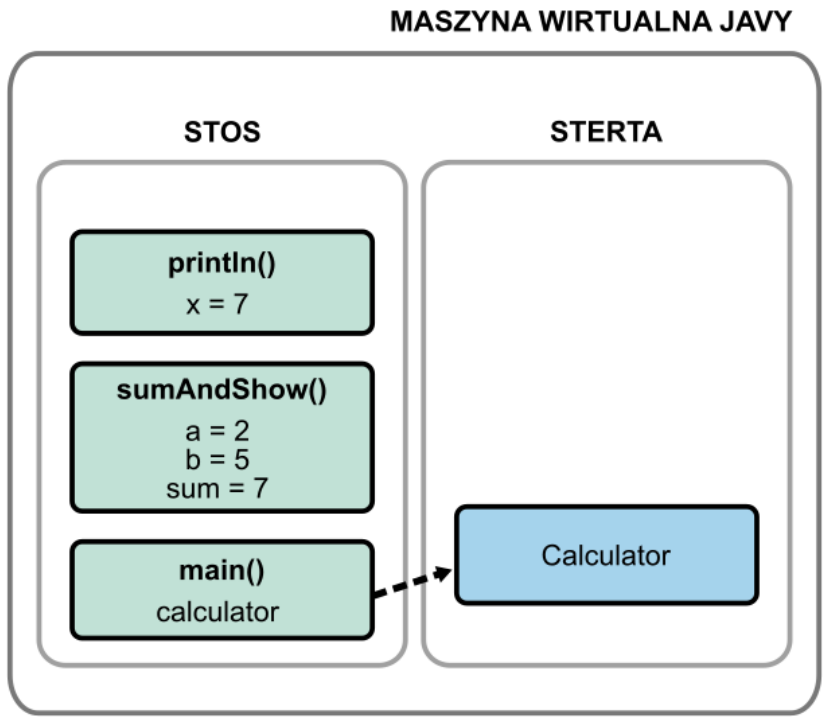
```
public class Calculator {

    public void sumAndShow(int a, int b) {
        int sum = sumAndReturn(a, b);
        System.out.println(sum);
    }

    public int sumAndReturn(int a, int b) {
        int sum = a + b;
        return sum;
    }
}
```



STOS I STERTA W TRAKCIE DZIAŁANIA PROGRAMU

```
public class CalculatorTest {
    public static void main(String[] args) {
        Calculator calc = new Calculator();
        calc.sumAndShow(2, 3);
    }
}
```

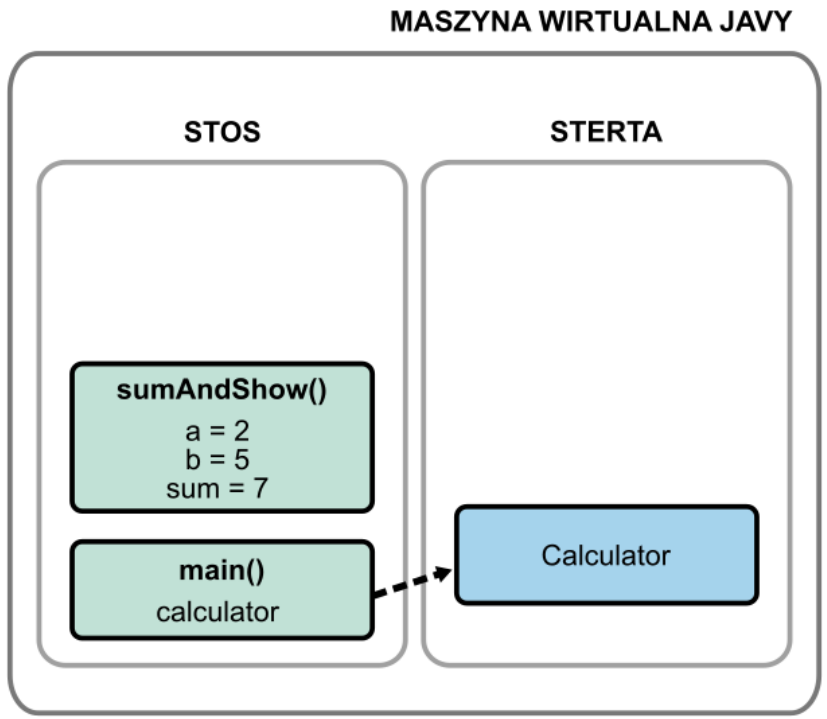
```
public class Calculator {

    public void sumAndShow(int a, int b) {
        int sum = sumAndReturn(a, b);
        System.out.println(sum);
    }

    public int sumAndReturn(int a, int b) {
        int sum = a + b;
        return sum;
    }
}
```



STOS I STERTA W TRAKCIE DZIAŁANIA PROGRAMU

```
public class CalculatorTest {
    public static void main(String[] args) {
        Calculator calc = new Calculator();
        calc.sumAndShow(2, 3);
    }
}
```

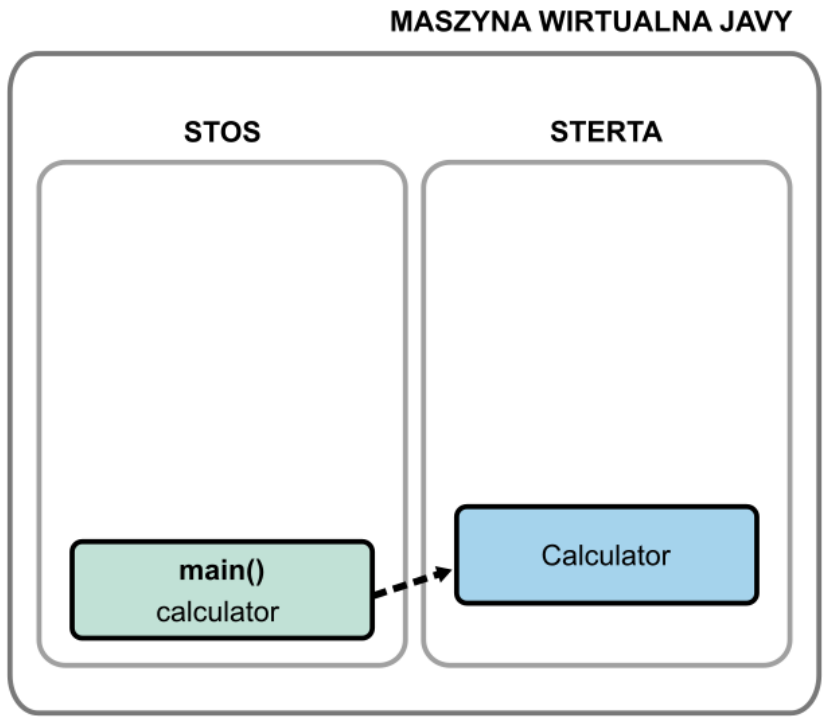
```
public class Calculator {

    public void sumAndShow(int a, int b) {
        int sum = sumAndReturn(a, b);
        System.out.println(sum);
    }

    public int sumAndReturn(int a, int b) {
        int sum = a + b;
        return sum;
    }
}
```



STOS I STERTA W TRAKCIE DZIAŁANIA PROGRAMU

```
public class CalculatorTest {
    public static void main(String[] args) {
        Calculator calc = new Calculator();
        calc.sumAndShow(2, 3);
    }
}
```

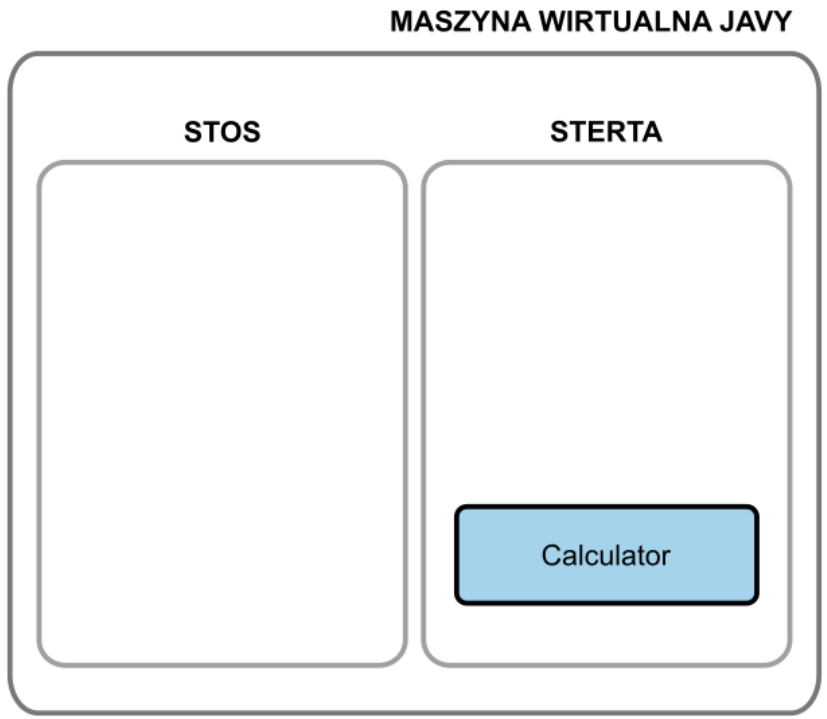
```
public class Calculator {

    public void sumAndShow(int a, int b) {
        int sum = sumAndReturn(a, b);
        System.out.println(sum);
    }

    public int sumAndReturn(int a, int b) {
        int sum = a + b;
        return sum;
    }
}
```



STOS I STERTA W TRAKCIE DZIAŁANIA PROGRAMU

```
public class CalculatorTest {
    public static void main(String[] args) {
        Calculator calc = new Calculator();
        calc.sumAndShow(2, 3);
    }
}
```

```
public class Calculator {

    public void sumAndShow(int a, int b) {
        int sum = sumAndReturn(a, b);
        System.out.println(sum);
    }

    public int sumAndReturn(int a, int b) {
        int sum = a + b;
        return sum;
    }
}
```

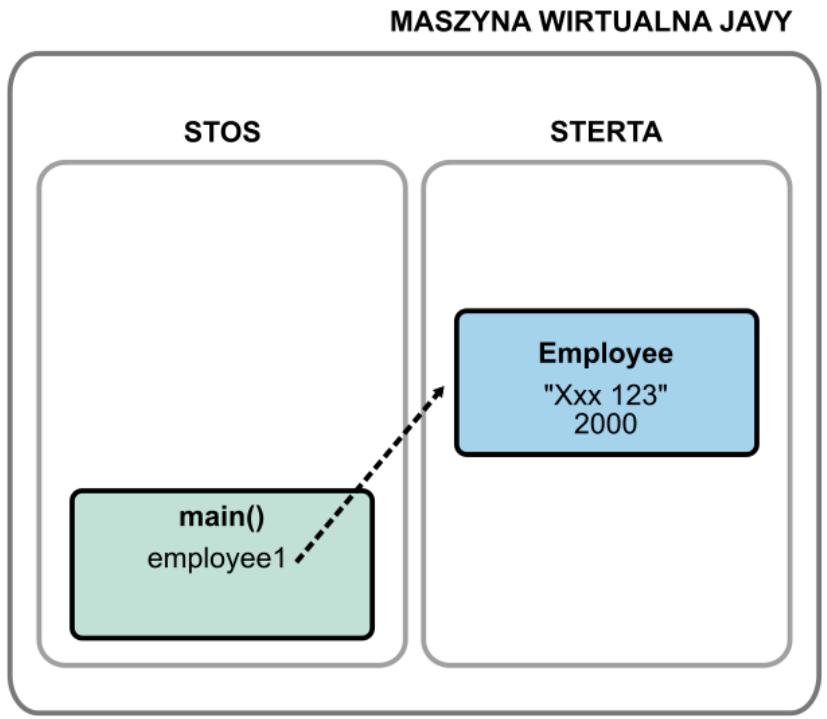


STOS I STERTA W TRAKCIE DZIAŁANIA PROGRAMU

```
public class SalaryBonusTest {
    public static void main(String[] args) {
        Employee employee1 = new Employee("Xxx 123", 2000);

        Company company = new Company();
        company.increaseSalary(employee1);

        System.out.println(employee1.salary);
} }
```

```
public class Employee {
    String ID;
    double salary;

    public Employee(String ID, double salary) {
        this.ID = ID;
        this.salary = salary;
} }
```

```
public class Company {
    public void increaseSalary(Employee emp) {
        emp.salary = emp.salary + 0.1 * emp.salary;
} }
```

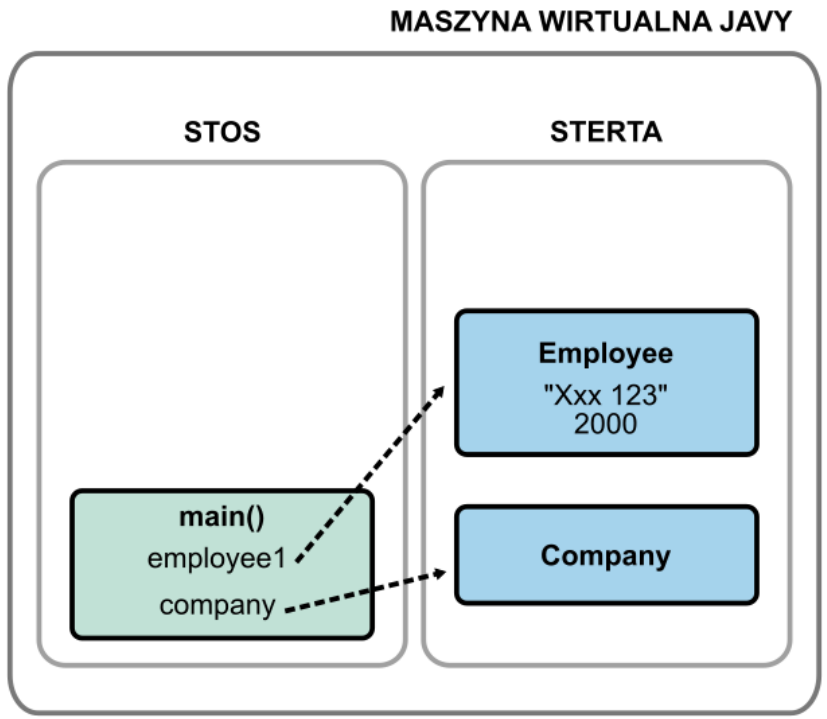


STOS I STERTA W TRAKCIE DZIAŁANIA PROGRAMU

```
public class SalaryBonusTest {
    public static void main(String[] args) {
        Employee employee1 = new Employee("Xxx 123", 2000);

        Company company = new Company();
        company.increaseSalary(employee1);

        System.out.println(employee1.salary);
} }
```

```
public class Employee {
    String ID;
    double salary;

    public Employee(String ID, double salary) {
        this.ID = ID;
        this.salary = salary;
} }
```

```
public class Company {
    public void increaseSalary(Employee emp) {
        emp.salary = emp.salary + 0.1 * emp.salary;
} }
```

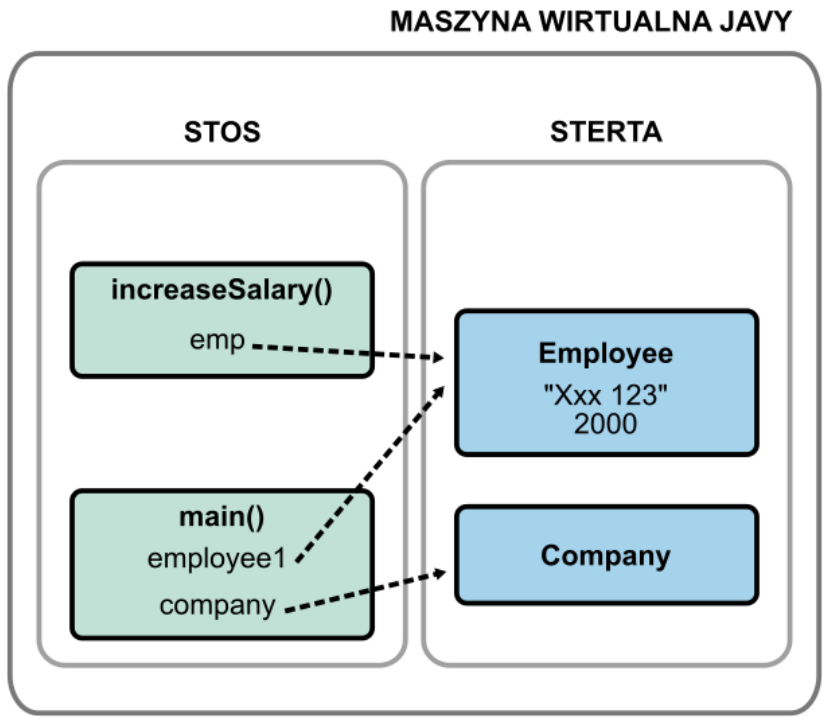


STOS I STERTA W TRAKCIE DZIAŁANIA PROGRAMU

```
public class SalaryBonusTest {
    public static void main(String[] args) {
        Employee employee1 = new Employee("Xxx 123", 2000);

        Company company = new Company();
        company.increaseSalary(employee1);

        System.out.println(employee1.salary);
} }
```

```
public class Employee {
    String ID;
    double salary;

    public Employee(String ID, double salary) {
        this.ID = ID;
        this.salary = salary;
} }
```

```
public class Company {
    public void increaseSalary(Employee emp) {
        emp.salary = emp.salary + 0.1 * emp.salary;
} }
```

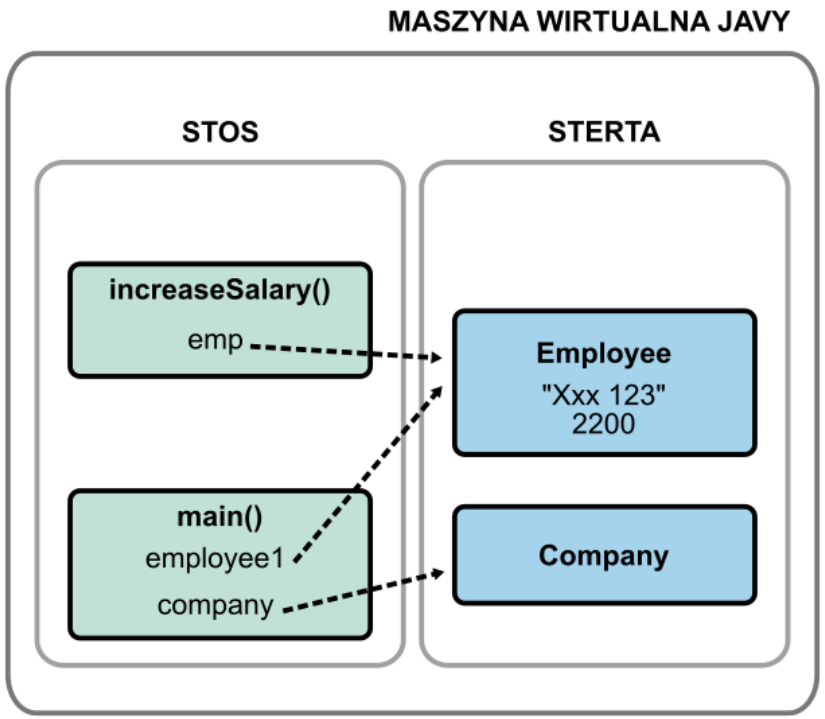


STOS I STERTA W TRAKCIE DZIAŁANIA PROGRAMU

```
public class SalaryBonusTest {
    public static void main(String[] args) {
        Employee employee1 = new Employee("Xxx 123", 2000);

        Company company = new Company();
        company.increaseSalary(employee1);

        System.out.println(employee1.salary);
} }
```

```
public class Employee {
    String ID;
    double salary;

    public Employee(String ID, double salary) {
        this.ID = ID;
        this.salary = salary;
} }
```

```
public class Company {
    public void increaseSalary(Employee emp) {
        emp.salary = emp.salary + 0.1 * emp.salary;
} }
```

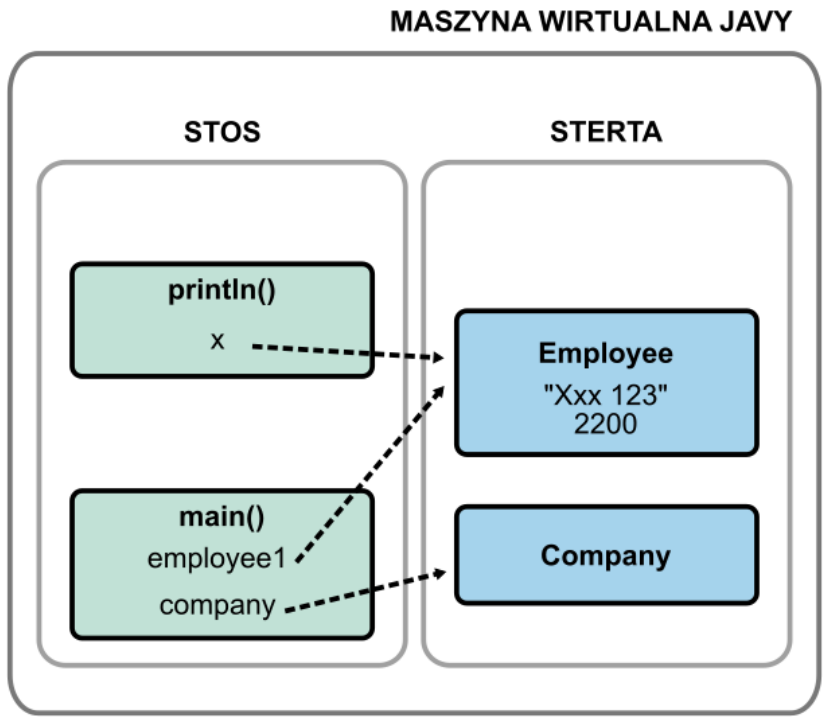


STOS I STERTA W TRAKCIE DZIAŁANIA PROGRAMU

```
public class SalaryBonusTest {
    public static void main(String[] args) {
        Employee employee1 = new Employee("Xxx 123", 2000);

        Company company = new Company();
        company.increaseSalary(employee1);

        System.out.println(employee1.salary);
} }
```

```
public class Employee {
    String ID;
    double salary;

    public Employee(String ID, double salary) {
        this.ID = ID;
        this.salary = salary;
} }
```

```
public class Company {
    public void increaseSalary(Employee emp) {
        emp.salary = emp.salary + 0.1 * emp.salary;
} }
```

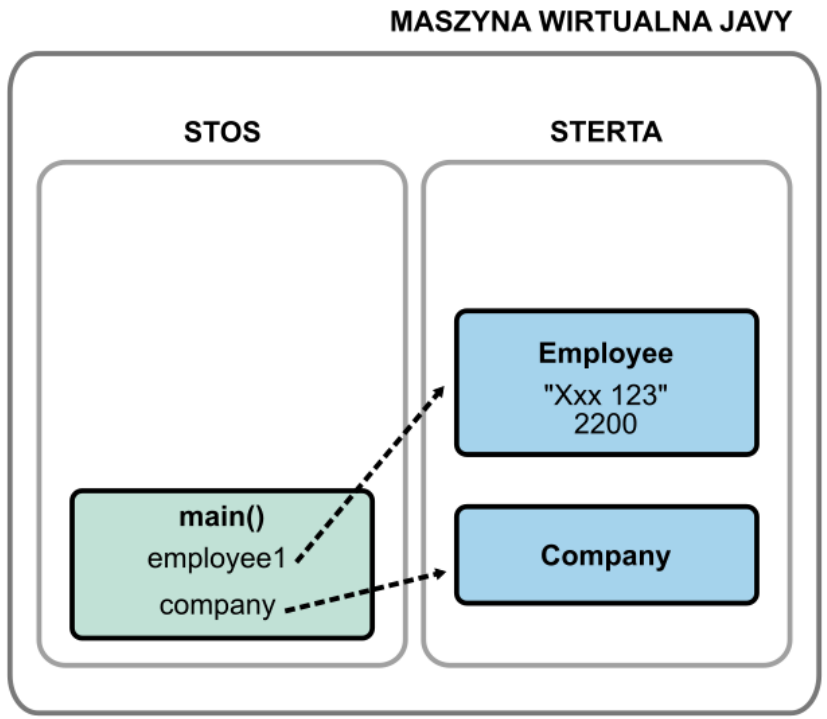


STOS I STERTA W TRAKCIE DZIAŁANIA PROGRAMU

```
public class SalaryBonusTest {
    public static void main(String[] args) {
        Employee employee1 = new Employee("Xxx 123", 2000);

        Company company = new Company();
        company.increaseSalary(employee1);

        System.out.println(employee1.salary);
} }
```

```
public class Employee {
    String ID;
    double salary;

    public Employee(String ID, double salary) {
        this.ID = ID;
        this.salary = salary;
} }
```

```
public class Company {
    public void increaseSalary(Employee emp) {
        emp.salary = emp.salary + 0.1 * emp.salary;
} }
```

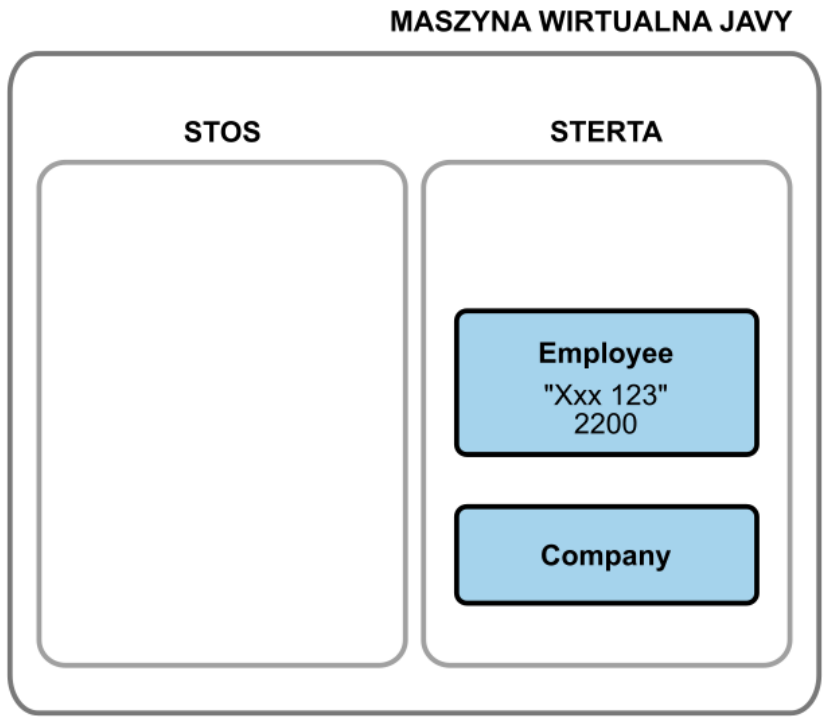


STOS I STERTA W TRAKCIE DZIAŁANIA PROGRAMU

```
public class SalaryBonusTest {
    public static void main(String[] args) {
        Employee employee1 = new Employee("Xxx 123", 2000);

        Company company = new Company();
        company.increaseSalary(employee1);

        System.out.println(employee1.salary);
} }
```

```
public class Employee {
    String ID;
    double salary;

    public Employee(String ID, double salary) {
        this.ID = ID;
        this.salary = salary;
} }
```

```
public class Company {
    public void increaseSalary(Employee emp) {
        emp.salary = emp.salary + 0.1 * emp.salary;
} }
```



STACK TRACE

```java
public class Calculator {
    public double divide(int a, int b) {
        if (b == 0) {
            throw new ArithmeticException("Illegal division by 0.");
        } else {
            return a / b;
        }
    }
}
```

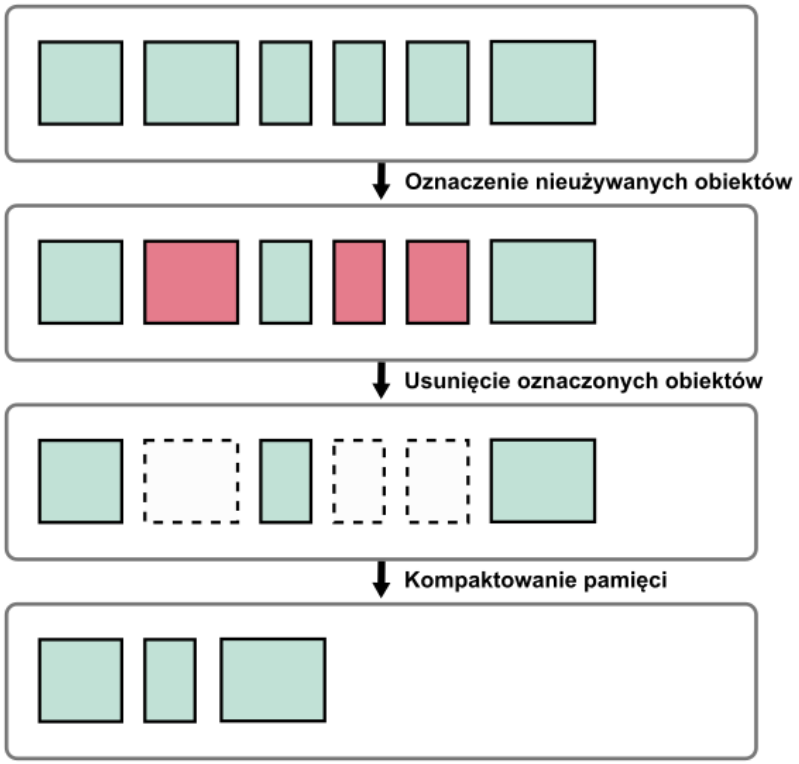
```java
public class Main {

    public static void main(String[] args) {
        Calculator c = new Calculator();
        divisionTest(c);
    }

    public static void divisionTest(Calculator c) {
        c.divide(2, 0);
    }
}
```

```
Exception in thread "main" java.lang.ArithmeticException Create breakpoint : Illegal division by 0.
    at lecture.Calculator.divide(Calculator.java:6)
    at lecture.Main.divisionTest(Main.java:18)
    at lecture.Main.main(Main.java:13)

Process finished with exit code 1
```

GARBAGE COLLECTOR (GC)

Pol. *odśmieczacz, zbieracz śmieci*

- wbudowany mechanizm odpowiedzialny za usuwanie nieużywanych obiektów (czyszczenie sterty)
- w Javie wywoływany automatycznie, działa w tle – maszyna wirtualna decyduje, kiedy go uruchomić (można zasugerować wywołanie GC, ale to wciąż nie gwarantuje jego uruchomienia)

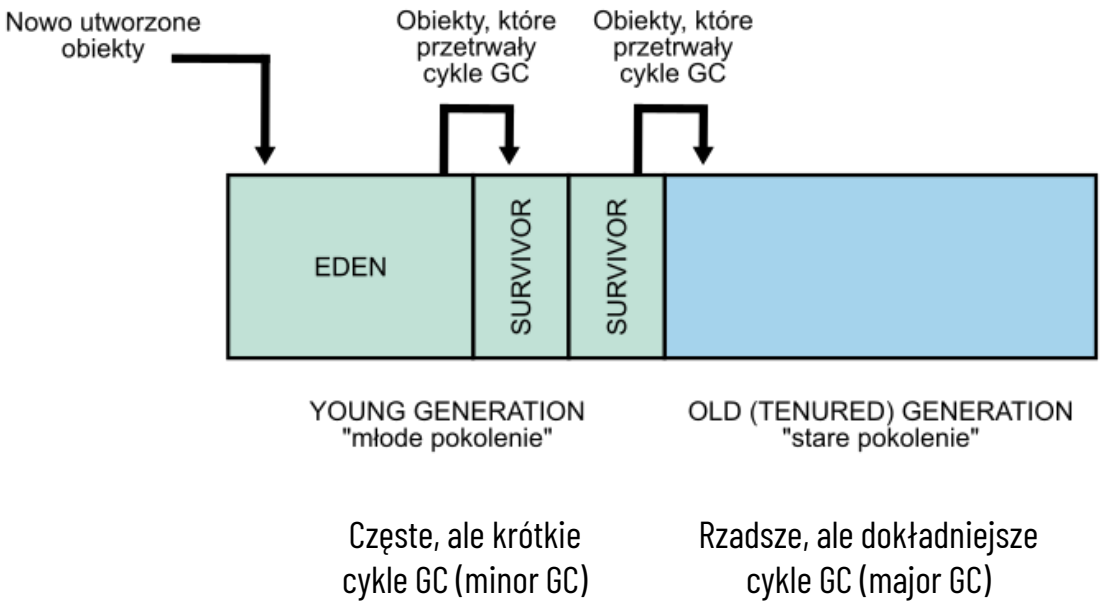


Schemat za: https://stackoverflow.com/questions/4813005/garbage-collection-java

GARBAGE COLLECTOR A STRUKTURA STERTY

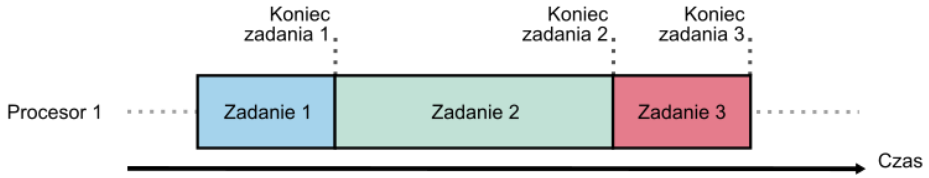
"Hipoteza generacyjna":

- większość obiektów jest używana tylko przez krótki czas ("młode obiekty umierają młodo")
- obiekty używane od dłuższego czasu prawdopodobnie będą używane dalej



Schemat za: https://docs.oracle.com/en/java/javase/21/gctuning/garbage-collector-implementation.html

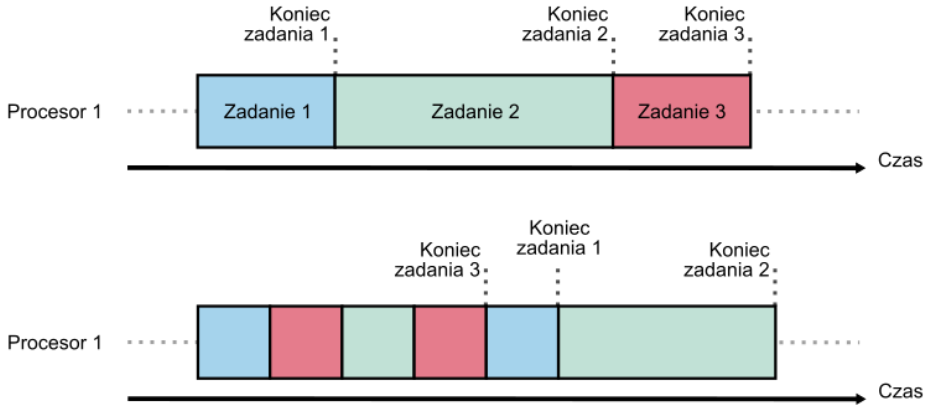
WSPÓŁBIEŻNOŚĆ A RÓWNOLEGŁOŚĆ



WSPÓŁBIEŻNOŚĆ A RÓWNOLEGŁOŚĆ

WSPÓŁBIEŻNOŚĆ (ang. Concurrency)

- operacje wykonywane w tym samym czasie (z perspektywy obserwatora), ale nie jednocześnie
- oparta na przełączaniu kontekstu w ramach jednego procesora (rdzenia)



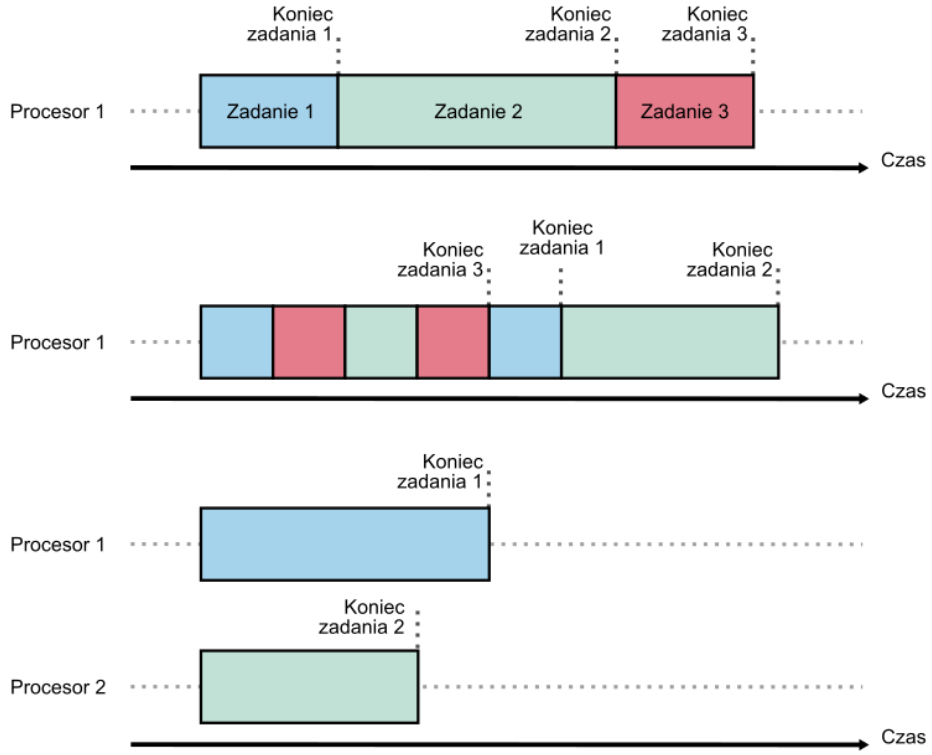
WSPÓŁBIEŻNOŚĆ A RÓWNOLEGŁOŚĆ

WSPÓŁBIEŻNOŚĆ (ang. Concurrency)

- operacje wykonywane w tym samym czasie (z perspektywy obserwatora), ale nie jednocześnie
- oparta na przełączaniu kontekstu w ramach jednego procesora (rdzenia)

RÓWNOLEGŁOŚĆ (ang. Parallelism)

- operacje wykonywane jednocześnie na kilku procesorach (rdzeniach)



WIELOWĄTKOWOŚĆ (ang. Multithreading)

- wątek (ang. thread) to wydzielona część programu, która może być wykonywana niezależnie, w tym samym czasie co inne operacje
- wątek jest wykonywany niezależnie, ale współdzieli zasoby w obrębie programu (procesu)
- wątki umożliwiają współbieżność (ale niekoniecznie równoległość)
- za przydzielanie wątkom zasobów procesora) odpowiada system operacyjny

TWORZENIE WĄTKÓW

```
public class MyThread extends Thread {

    @Override
    public void run() {
        // some operations here
    }
}
```

```
MyThread thread = new MyThread();
```

PODEJŚCIE 1:

stworzenie klasy dziedziczącej po klasie Thread i przesłonięcie metody *run()*

Metoda *run()* powinna zawierać wszystkie operacje, które ma wykonać dany wątek

TWORZENIE WĄTKÓW

```
public class MyThread extends Thread {

    @Override
    public void run() {
        // some operations here
    }
}
```

```
MyThread thread = new MyThread();
```

PODEJŚCIE 1:

stworzenie klasy dziedziczącej po klasie Thread i przesłonięcie metody *run()*

Metoda *run()* powinna zawierać wszystkie operacje, które ma wykonać dany wątek

```
public class MyRunnable implements Runnable {

    @Override
    public void run() {
        // some operations here
    }
}
```

```
MyRunnable runnable = new MyRunnable();
Thread thread = new Thread(runnable);
```

PODEJŚCIE 2:

stworzenie klasy implementującej interfejs Runnable i przesłonięcie metody *run()*, a później przekazanie jej do konstruktora klasy Thread

URUCHAMIANIE WĄTKÓW

```
public class MyRunnable implements Runnable {

    @Override
    public void run() {
        System.out.println(Thread.currentThread().getName()
            + " is running now.");
    }
}
```

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

    System.out.println("Starting program...");
    for (int i = 0; i < 5; i++) {
        MyRunnable runnable = new MyRunnable();
        Thread thread = new Thread(runnable);
        thread.start();
    }
    System.out.println("Program finished.");
}
```

```
Starting program...
Program finished.
Thread-3 is running now.
Thread-1 is running now.
Thread-0 is running now.
Thread-2 is running now.
Thread-4 is running now.
```

URUCHAMIANIE WĄTKÓW

```
public class MyRunnable implements Runnable {

    @Override
    public void run() {
        System.out.println(Thread.currentThread().getName()
            + " is running now.");
    }
}
```

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

    System.out.println("Starting program...");
    for (int i = 0; i < 5; i++) {
        MyRunnable runnable = new MyRunnable();
        Thread thread = new Thread(runnable);
        thread.start();
    }
    System.out.println("Program finished.");
}
```

```
Starting program...
Program finished.
Thread-3 is running now.
Thread-1 is running now.
Thread-0 is running now.
Thread-2 is running now.
Thread-4 is running now.
```

Ważne:

- do uruchomienia nowego wątku służy metoda *start()*, która z kolei wywołuje metodę *run()*
- wywołanie wprost metody run() spowoduje jej wykonanie w aktualnym wątku

```
Starting program...
main is running now.
main is running now.
main is running now.
main is running now.
main is running now.
Program finished.
```

WYŚCIG (ang. Race condition)

```
public class Counter {
    private long count = 0;

    public void increment() {
        count = count + 1; }

    public long getCount() {
        return count; }
}
```

```
public class MyRunnable implements Runnable {
    private Counter counter;

    public MyRunnable(Counter counter) {
        this.counter = counter; }

    @Override
    public void run() {
        for (int i = 0; i < 100000; i++) {
            counter.increment();
        }
    }
}
```

```
System.out.println("Starting program...");

Counter c = new Counter();
Thread thread1 = new Thread(new MyRunnable(c));
Thread thread2 = new Thread(new MyRunnable(c));
Thread thread3 = new Thread(new MyRunnable(c));

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

thread1.join();
thread2.join();
thread3.join();

System.out.println("Count: " + c.getCount());
System.out.println("Program finished.");
```

```
Starting program...
Count: 106220
Program finished.
```

```
Starting program...
Count: 120797
Program finished.
```

```
Starting program...
Count: 114123
Program finished.
```

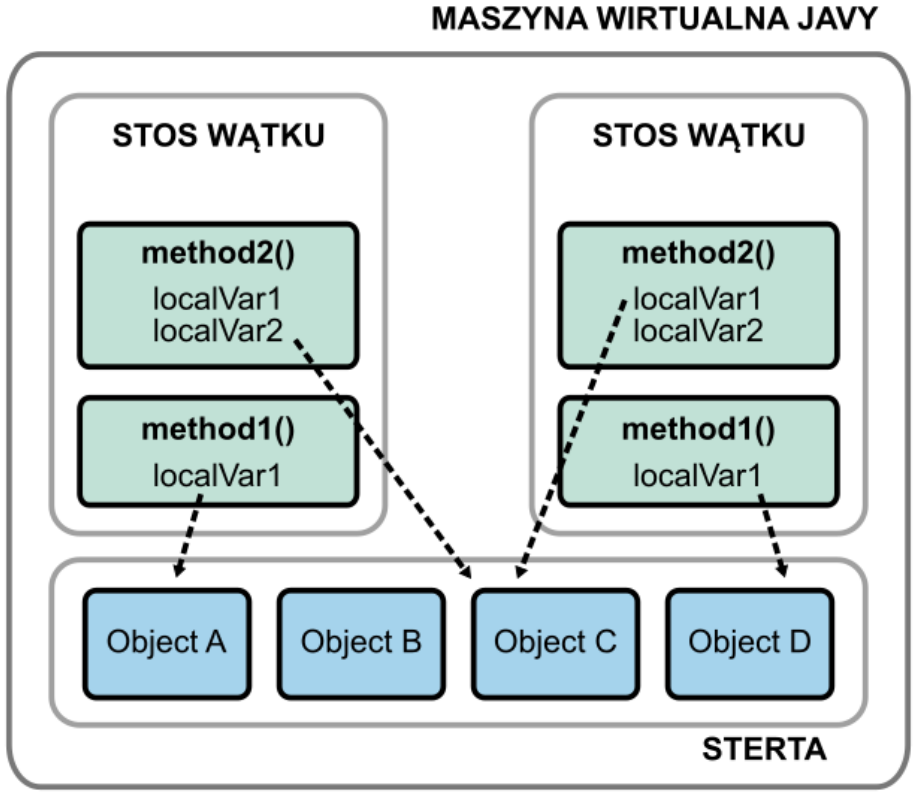
WYŚCIG (ang. Race condition)

```
public class Counter {
    private long count = 0;

    public void increment() {
        count = count + 1;
    }

    public long getCount() {
        return count; }
}
```

- instrukcja `count = count + 1` to trzy operacje: odczyt zmiennej z pamięci, dodanie wartości i zapis do pamięci
- pomiędzy operacjami może zajść przełączenie wątku
- operacje, których nie da się podzielić na części nazywa się *atomowymi* (ang. atomic)

WYŚCIG (ang. Race condition)

- sytuacja, w której kilka wątków jednocześnie modyfikuje daną zmienną, przez co kolejność wykonania wpływa na końcowy wynik
- fragment kodu, w którym może zajść wyścig nazywa się *sekcją krytyczną* (ang. critical section)
- sekcje krytyczne zwykle dotyczą sytuacji, w których następuje:
  - odczyt, modyfikacja i zwrócenie wartości
  - sprawdzenie, a potem zmiana warunku



SYNCHRONIZACJA (ang. Synchronization)

```
public class Counter {
    private long count = 0;

    public void increment() {
        synchronized (this) {
            count = count + 1;
        }
    }
}
```

```
public class Counter {
    private long count = 0;

    public synchronized void increment() {
        count = count + 1;
    }
}
```

SŁOWO KLUCZOWE SYNCHRONIZED

- oznacza fragment kodu, który może być wykonywany w danym momencie tylko przez jeden wątek
- może być zastosowane do bloku kodu lub metody
- najbardziej podstawowy sposób synchronizacji – inne to np. blokady (ang. locks) i semafory (ang. semaphore)

KOMUNIKACJA POMIĘDZY WĄTKAMI

```
public class ProducerThread implements Runnable {
    private Queue<Double> queue;
    private int itemCount;

    public ProducerThread(Queue queue, int itemCount) {
        this.queue = queue;
        this.itemCount = itemCount;
    }

    @Override
    public void run() {
        for (int i = 1; i < itemCount; i++) {
            synchronized (queue) {
                queue.add(Math.random());
            }
            System.out.println("Sent item no " + i);
        }
    }
}
```

```
public class ConsumerThread implements Runnable {
    private Queue<Double> queue;
    private int itemCount;

    public ConsumerThread(Queue queue, int itemCount) {
        this.queue = queue;
        this.itemCount = itemCount;
    }

    @Override
    public void run() {
        int itemsReceived = 0;
        while (itemsReceived < itemCount) {
            Double item;
            synchronized (queue) {
                if (queue.isEmpty()) continue;
                item = queue.poll();
            }
            itemsReceived++;
        }
    }
}
```

KOMUNIKACJA POMIĘDZY WĄTKAMI

```
public class ProducerThread implements Runnable {
    private Queue<Double> queue;
    private int itemCount;

    @Override
    public void run() {
        for (int i = 1; i < itemCount; i++) {
            synchronized (queue) {
                queue.add(Math.random());
                queue.notify();
            }
            System.out.println("Sent item no " + i);
        }
    }
}
```

```
public class ConsumerThread implements Runnable {
    private Queue<Double> queue;
    private int itemCount;

    @Override
    public void run() {
        int itemsReceived = 0;
        while (itemsReceived < itemCount) {
            Double item;
            synchronized (queue) {
                while (queue.isEmpty()) {
                    try {
                        queue.wait();
                    } catch (InterruptedException e) {
                        throw new RuntimeException(e);
                    }
                }
                item = queue.poll();
            }
            itemsReceived++;
        }
    }
}
```

CYKL ŻYCIA WĄTKU