

Języki Programowania

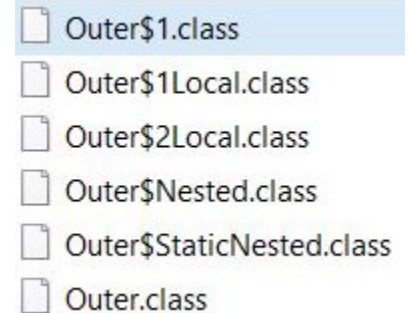
dr inż. Tomasz Kubik

tomasz.kubik.staff.iiar.pwr.edu.pl

Pliki źródłowe .java oraz pliki z kodem bajtowym .class

- W jednym pliku źródłowym może znajdować się **wiele klas**, ale **tylko jedna** z nich może być **klasą publiczną**.
- Istnieje jednak **wyjątek od powyższej reguły**. Polecenie `java` może być wykorzystane do bezpośredniego uruchamiania jakiegoś kodu źródłowego (bez etapu jawnej jego kompilacji), pod warunkiem, że pierwsza zamieszczona w tym kodzie klasa jest publiczna i zawiera metodę `main()`, od której zacząć się ma wykonywanie programu (pozostałe klasy w tym kodzie źródłowym mogą być publiczne).
- Nazwa pliku źródłowego powinna odpowiadać nazwie klasy publicznej. Jeśli w pliku źródłowym brak klasy publicznej, to jego nazwa może być dowolna (jest to możliwe, ale nie jest zalecane).
- Nie ma specjalnych ograniczeń co do lokalizacji plików źródłowych, zaleca się jednak umieszczać je w strukturze katalogów odpowiadającej pakietom.
- Dla każdej klasy występującej w kodzie źródłowym po kompilacji wygenerowany zostanie kod bajtowy. Nazwa pliku z kodem bajtowym odpowiadać będzie nazwie klasy.
- W szczególnym przypadku, gdy w kodzie źródłowym wystąpiły klasy wewnętrzne, zagnieżdżone lub anonimowe, nazwa pliku z kodem bajtowym będzie kombinacją klasy zewnętrznej, znaku „\$” i ewentualnie kolejnych numerów (w przypadku klasy anonimowej) oraz nazwy klasy wewnętrznej lub zagnieżdżonej.

<https://www.baeldung.com/java-class-file-naming>



- Outer\$1.class
- Outer\$1Local.class
- Outer\$2Local.class
- Outer\$Nested.class
- Outer\$StaticNested.class
- Outer.class

<https://www.infoq.com/articles/single-file-execution-java11/>

JAR

- Kody bajtowe klas można spakować do jednego archiwum *.jar
 - Format tego archiwum został ustandaryzowany
 - jest to archiwum zip
 - pliki znajdują się w zadanej strukturze katalogów

<https://docs.oracle.com/javase/9/docs/specs/jar/jar.html>
 - Do tworzenia archiwów *.jar służy narzędzie jar
 - jest to jedno z wielu narzędzi dostępnych w JDK

<https://docs.oracle.com/en/java/javase/11/tools/tools-and-command-reference.html>

 - przykład użycia

```
jar --create --file classes.jar Foo.class Bar.class
jar --create --file classes.jar --manifest mymanifest -C foo/
jar --create --file my.jar @classes.list
```
- W archiwum *.jar w katalogu META-INF powinien znaleźć się plik MANIFEST.MF
- Manifest uruchamialnego *.jar powinien zawierać deklarację klasy głównej

```
Main-Class: com.foo.Hello
```
- Aby odpalić klasę z uruchamialnego *.jar należy wydać komendę

```
java.exe -jar .\hello.jar
```

Czytelny kod bajtowy klasy

```
// Compiled from Hello.java (version 11 : 55.0, super bit)
public class Hello {
```

```
    // Method descriptor #6 ()V
    // Stack: 1, Locals: 1
    public Hello();
        0   aload_0 [this]
        1   invokespecial java.lang.Object() [8]
        4   return
```

Line numbers:

[pc: 0, line: 2]

Local variable table:

[pc: 0, pc: 5] local: this index: 0 type: Hello

```
    // Method descriptor #15 ([Ljava/lang/String;)V
    // Stack: 2, Locals: 1
    public static void main(java.lang.String[] args);
```

```
        0   getstatic java.lang.System.out : java.io.PrintStream [16]
```

```
        3   ldc <String "Hello"> [22]
```

```
        5   invokevirtual java.io.PrintStream.println(java.lang.String) : void [23]
```

```
        8   return
```

Line numbers:

[pc: 0, line: 4]

[pc: 8, line: 5]

Local variable table:

[pc: 0, pc: 9] local: args index: 0 type: java.lang.String[]

```
}
```

```
public class Hello {
    public static void main(String[] args) {
        System.out.println("Hello");
    }
}
```

```
> javac Hello.java
> javap -c Hello
```

Instrukcje kodu bajtowego

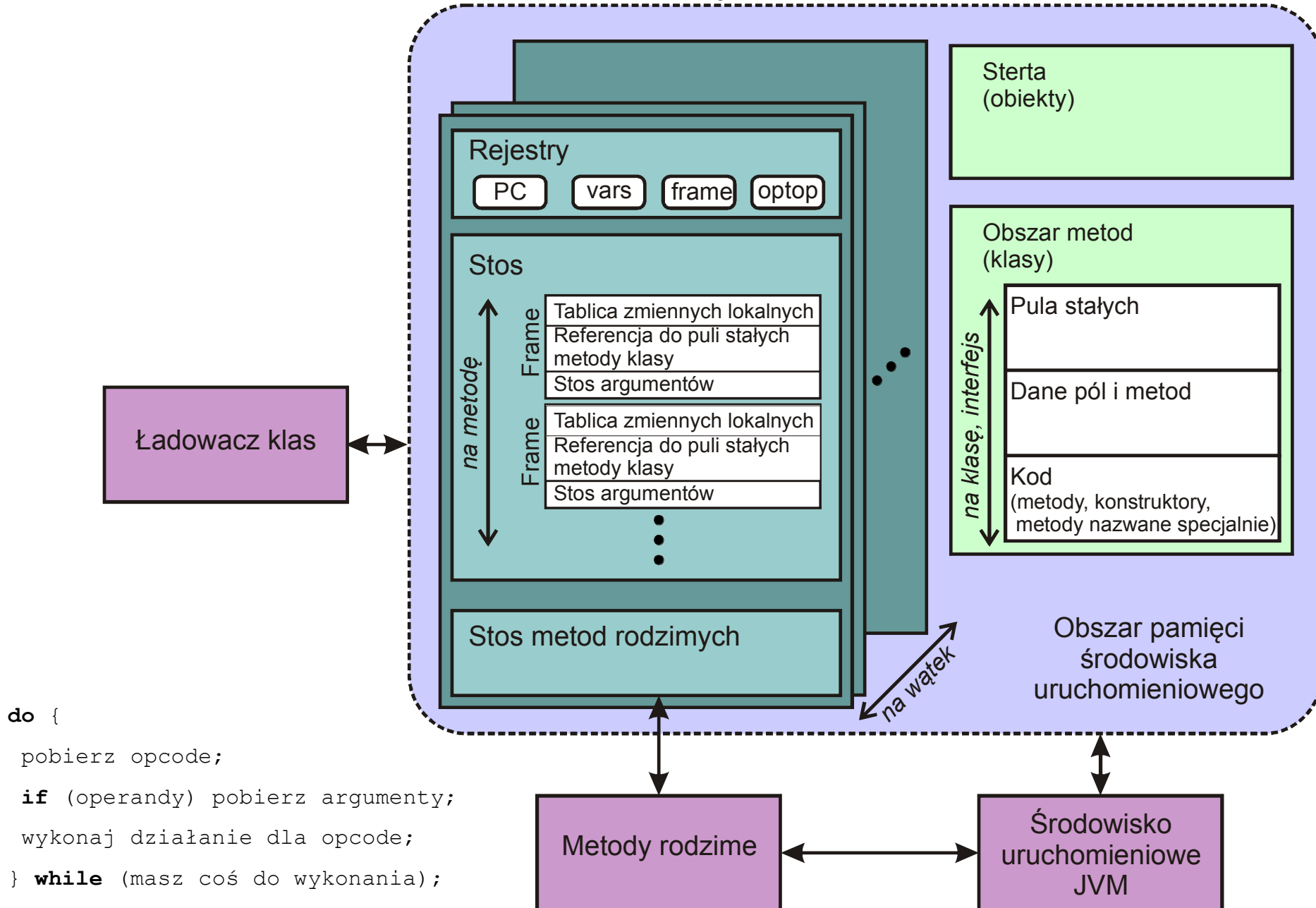
- 212 instrukcje
 - opcode (8bit) + 0 lub więcej argumentów
- 44 instrukcje zarezerwowane
 - przyszłe rozszerzenia lub pośrednia optymalizacja JVM
- mnemoniki
 - **a**... : manipulacja referencjami (Class, Interface, Array)
 - **s**... : operacje dla typu short
 - **i**... : operacje dla typu integer (boolean)
 - **l**... : operacje dla typu long,
 - **b**... : operacje dla typu byte,
 - **c**... : operacje dla typu char,
 - **f**... : operacje dla typu float
 - **d**... : operacje dla typu double

```
// Bytecode stream:  
// 03 3b 84 00 01 1a 05 68 3b a7 ff f9  
// Disassembly:  
iconst_0    // 03  
istore_0    // 3b  
iinc 0, 1   // 84 00 01  
iload_0     // 1a  
iconst_2    // 05  
imul        // 68  
istore_0    // 3b  
goto -7     // a7 ff f9
```

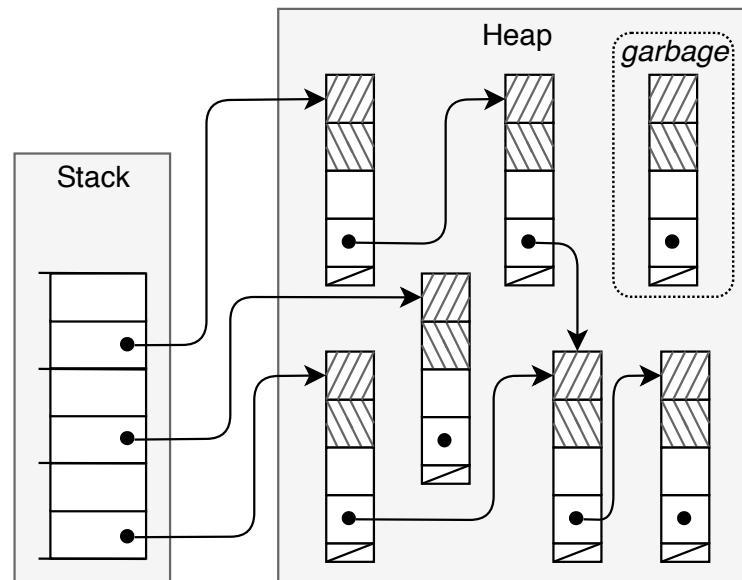
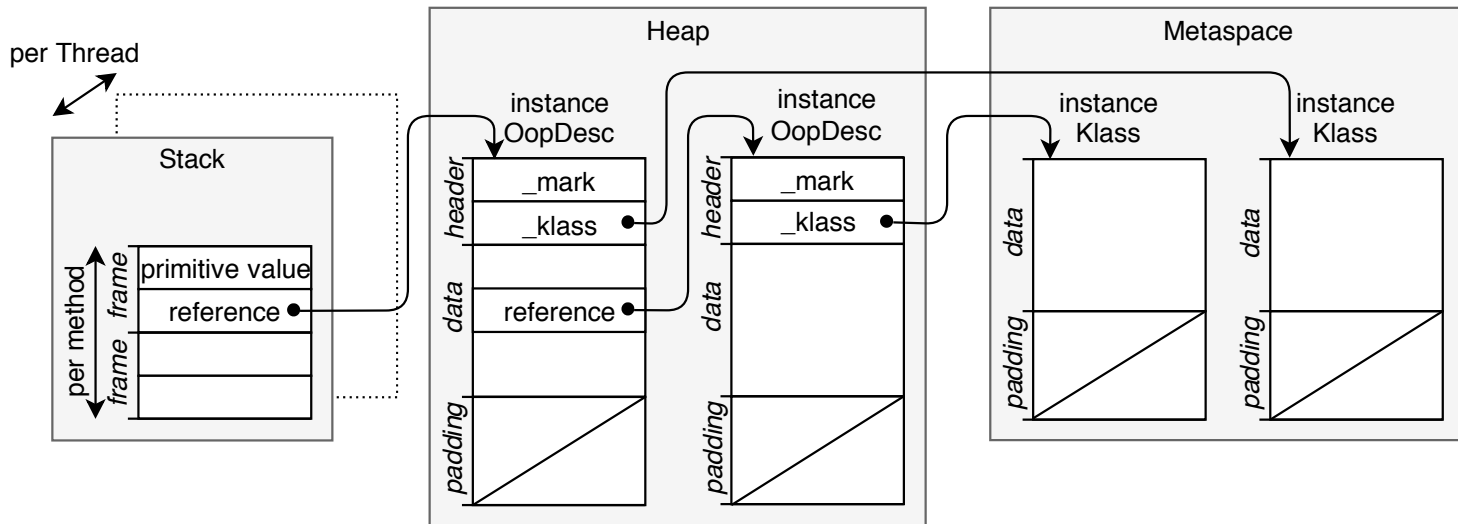
Instrukcje kodu bajtowego

- instrukcje w podziale na manipulowane nimi elementy architektury JVM
 - Stos: `iconst, iload, bipush, istore, pop, dup`
 - PC: `goto, ifeq, ifgt, return, athrow`
 - Sterta: `new, newarray`
 - Pola: `getstatic, putstatic, getfield, putfield`
 - Metody: `invokestatic, invokevirtual`
- instrukcje w podziale na realizowane funkcje
 - Przerzucanie: `pop, swap, dup, ...`
 - Obliczanie: `iadd, isub, imul, idiv, ineg, ...`
 - Konwersja: `d2i, i2b, d2f, i2z, ...`
 - Operacje na pamięci lokalnej: `iload, istore, ...`
 - Operacje na tablicach: `arraylength, newarray, ...`
 - Zarządzanie obiektami: `get/putfield, invokevirtual, new`
 - Operacje typu push: `acost_null, iconst_m1, ...`
 - Strumień sterowania: `nop, goto, jsr, ret, tableswitch, ...`
 - Wielowątkowość: `monitorenter, monitorexit, ...`

Wirtualna maszyna JAVA



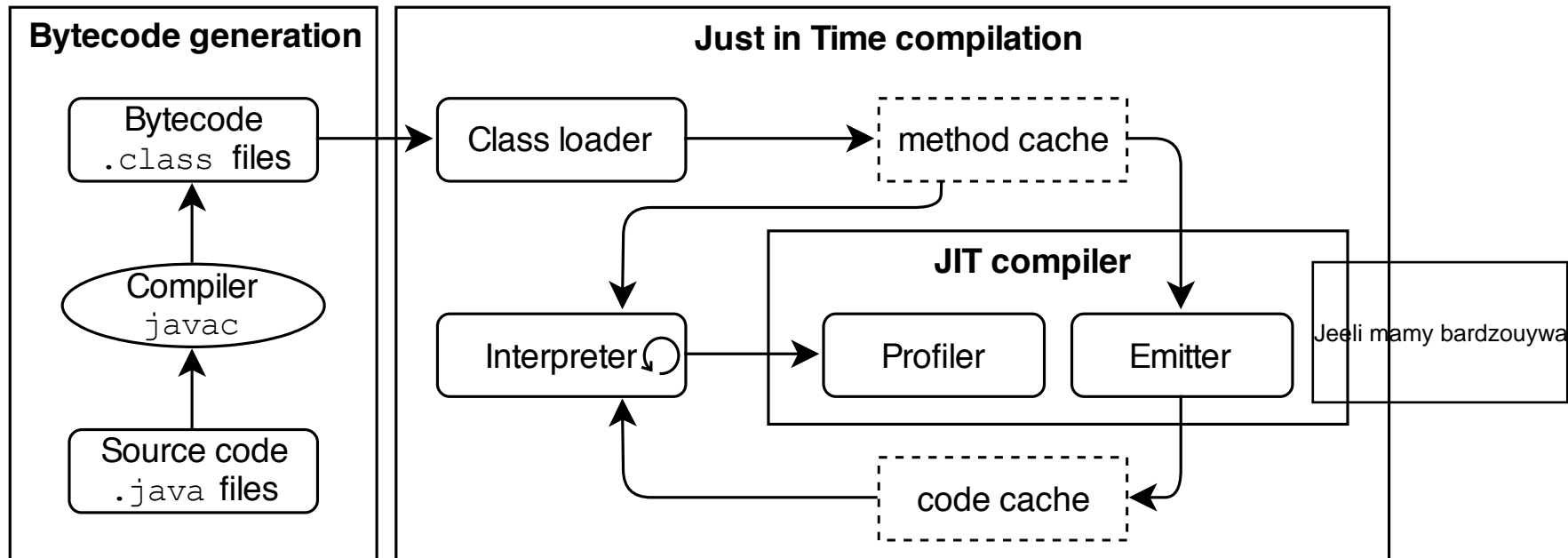
Stos, sarta, odsmieanie



Stos, sterta, odśmiecanie

<https://stuefe.de/posts/metaspacer/what-is-metaspacer/>

Wirtualna maszyna



<https://www.ibm.com/docs/en/ztpf/1.1.0.15?topic=reference-jit-compiler>

<https://softwareskill.pl/podsumowanie-jvm-jit>

Języki kompilowane do kodu bajtowego JVM

- [PHP](#) , z Quercus
- [Clojure](#), dialekt [Lisp](#)
- [Groovy](#), język skryptowy
- [JRuby](#), implementacja [Ruby](#)
- [Jython](#), implementacja [Python](#)
- [Rhino](#), implementacja [JavaScript](#)
- [Scala](#), język programowania obiektowego i funkcjonalnego
- istnieją kompilatory [Ada](#) oraz [COBOL](#).



Konwencja nazewnicza i deklaracje klas, metod, pól i zmiennych

- Deklaracja klas
 - ciało klasy ujęte jest w nawiasy klamrowe,
 - klasa może dziedziczyć z jednej klasy bazowej oraz implementować wiele interfejsów,
 - w ciele klasy pojawiają się: bloki inicjalizacji (statycznej i instancyjnej), deklaracje metod (w tym konstruktorów) i pól,
 - metody:
 - mogą pobierać atrybuty, zwracać wartości, korzystać ze zmiennych,
 - można przeciążać (deklarować o tych samych nazwach, ale innej liście atrybutów),
 - są nierozróżnialne przez kompilator, jeśli posiadają takie same listy atrybutów, nawet jeśli zwracają wartości różnego typu (takie metody są jednak rozróżnialne w kodzie bajtowym).
- Deklaracja zmiennych
 - *typ nazwamiennnej*; lub *typ nazwazmiennnej1, nazwazmiennnej2*; (mogą być z inicjalizatorami *typ nazwazmiennnej=wartość*; lub *typ nazwazmiennnej=metoda()*);
`int i; int j=10, k;`
 - typy całkowite: `byte` | `short` | `int` | `long`
 - typy zmiennoprzecinkowe: `float` | `double`
 - typy inne: `char` | `boolean`
 - łańcuchy znaków: `String`, `StringBuffer` (z metodą `append()`)
 - tablice:
`int [] taba = {1,2,3,4}; // tablica czterech elementów typu int`
`int[] tabb = new int[10]; // choć można i tak: int tab[] = new int[10]`
`MyClass[] tabc = new MyClass[5]; // tablica 5 referencji`
`for(int i=0; i<5; i++) tabc[i] = new MyClass(); // utworzenie referencji`
- Zasady:
 - nazwy pól i zmiennych z małej litery,
 - `np.int toJestZmienna`
 - nazwy klas – z dużej litery, notacja wielbłądzia,
 - `np.class ToJestKlasa`
 - stosowane są ciągi znaków Unicode o dowolnej długości, zaczynające się literą, nie pokrywające się ze słowami kluczowymi,
 - nazwy zmiennych nie mogą powtarzać się w zasięgu obszaru ich obowiązywania

modyfikatory

- `static`
 - metody i pola klasy: stanowią część klasy, a nie instancji

```
public class A{  
    public static int i;  
    public static void m();}
```
- `final`
 - zmienne i pola klasy: mogą być zmodyfikowane tylko raz

```
final int aBlankFinal; // deklaracja  
aBlankFinal = 0; // modyfikacja
```
 - metody: nie można nadpisywać w klasach potomnych

```
final void metoda{}
```
 - klasy: nie mogą mieć klas potomnych

```
public final class A { }
```
- `abstract`
 - klasa: nie można utworzyć instancji takiej klasy (choć klasa może być bez metod abstrakcyjnych)
 - metoda: nie posiada implementacji

```
abstract class A {  
    abstract void metoda(); }
```
- `native`
 - metoda: wygląda jak metoda abstrakcyjna, ale jej implementacja jest wykonana w kodzie natywnym

```
native void metoda();
```
- `synchronized`
 - przy metodzie: czyni z metody monitor (stosowany w programowaniu aplikacji wielowątkowych)

```
synchronized void metoda() {...}
```
 - na obiekcie: definiuje sekcję krytyczną (stosowane w programowaniu aplikacji wielowątkowych)

```
synchronized(mutex) {...}
```
- `var`
 - skraca zapis typu (nie mylić z deklaracją dynamicznie typowanych zmiennych)

```
var fileName = "input.txt";
```
- `strictfp`
 - do deklaracji dokładności obliczeń zgodnie z IEEE's 754 standard:
 - można używać do klas, interfejsów i nieabstrakcyjnych metod
 - nie można używać do zmiennych, konstruktorów i abstrakcyjnych klas

```
strictfp public class A { }
```

Deklaracja klas

```
package pakiecik;           // Deklaracja pakietu (obowiązkowa dla projektów modułowych Java).

                               // Sekcja importów, w której pojawić się mogą:
import java.util.List;      // deklaracja importu indywidualnej klasy (zalecane)
import java.io.*;           // deklaracja importu całych pakietów klas (niezalecane, bo importuje się za dużo)
import static pakiet.B.m;   // deklaracja importu statycznej metody jakiejś klasy (mocno niezalecane)
                               // (w podobny sposób można też importować statyczne pola).

                               // Sekcja z deklaracjami/implementacjami klas:
                               // w jednym pliku ze źródłem kodu może pojawić się
                               // tylko jedna klasa publiczna oraz dowolna liczba klas niepublicznych.
                               // Nazwa pliku ze źródłem kodu (pomijając rozszerzenie .java)
                               // musi być taka, jak nazwa klasy publicznej (jeśli zdefiniowano taką klasę).

public class A {             // Tu zadeklarowano klasę publiczną (stąd modyfikator public).
                               // Jeśli nie zdefiniowano jawnie żadnego dziedziczenia słowem extends,
                               // to klasa dziedziczyć będzie bezpośrednio z klasy Object.

    {                       // Tu zaczyna się opcjonalna, instancyjna sekcja inicjalizacji
                               // mogąca zawierać kod, który wygląda jak ciało metody.
                               // Kod ten jest uruchamiany podczas tworzenia obiektu przed konstruktorem.

        int i = 1;
        System.out.println("Instancyjna sekcja inicjalizacji" + k);
                               // W sekcji tej można odwoływać się do pól lub metod statycznych,
                               // nawet jeśli będą one zadeklarowane później.
                               // Odwołania do pól lub metod instancyjnych będą możliwe,
                               // jeśli te pola i metody będą zadeklarowane wcześniej.

    }

    static {                 // Tu zaczyna się opcjonalna, statyczna sekcja inicjalizacji,
                               // mogąca zawierać kod, który wygląda jak ciało metody.
                               // Kod ten jest uruchamiana podczas ładowania klasy.

        System.out.println("Statyczna seekcja inicjalizacji");
                               // W sekcji tej można odwoływać się do pól lub metod statycznych,
                               // pod warunkiem, że będą one zadeklarowane wcześniej.
                               // W sekcji tej nie można odwoływać się do pól lub metod instancyjnych.

    }

                               // Deklaracje pól klasy mogą pojawiać się w dowolnym miejscu,
                               // jednak jeśli pola inicjalizowane są wartościami pobieranymi
                               // z innych pól, wtedy kolejność deklaracji pól musi być adekwatna.
                               // Ponadto o kolejności może decydować użycie tych pól w sekcjach inicjalizacji.

    public List<A> l = null;
    private int i = 0;
    int j = i + 1;           // Inicjalizacja pola instancji odbywa się przed uruchomieniem jej konstruktora.
    static int k = 0;         // Inicjalizacja pola statycznego odbywa się podczas ładowania klasy
                               // Wszystkie instancje klasy będą współdzielić pola statyczne.

                               // Metody można deklarować w dowolnej kolejności,
                               // podobnie w dowolnej kolejności można deklarować konstruktory.
```

```
public static void main(String[] args) {
    // W metodzie statycznej można odwoływać się bezpośrednio do pól statycznych,
    // jednak bezpośrednie odwołania do pól instancyjnych są niemożliwe.
    // Aby skorzystać z pola instancyjnego najpierw trzeba utworzyć obiekt.

    A.k = 10;                // Dostęp do pola statycznego poprzez użycie klasy (zalecane).
    A a = new A(10);         // Utworzenie obiektu zawsze poprzez new.
    a.j = 10;                // Dostęp do pola instancyjnego istniejącego obiektu.
    a.k = 20;                // Dostęp do pola statycznego poprzez istniejący obiekt (niezalecane).
    a.n();                   // Metodę prywatną można używać tylko w obrębie klasy.

}

private void n() {

    m(); // Tu wywołano zaimportowaną metodę statyczną (budzi niejasności).

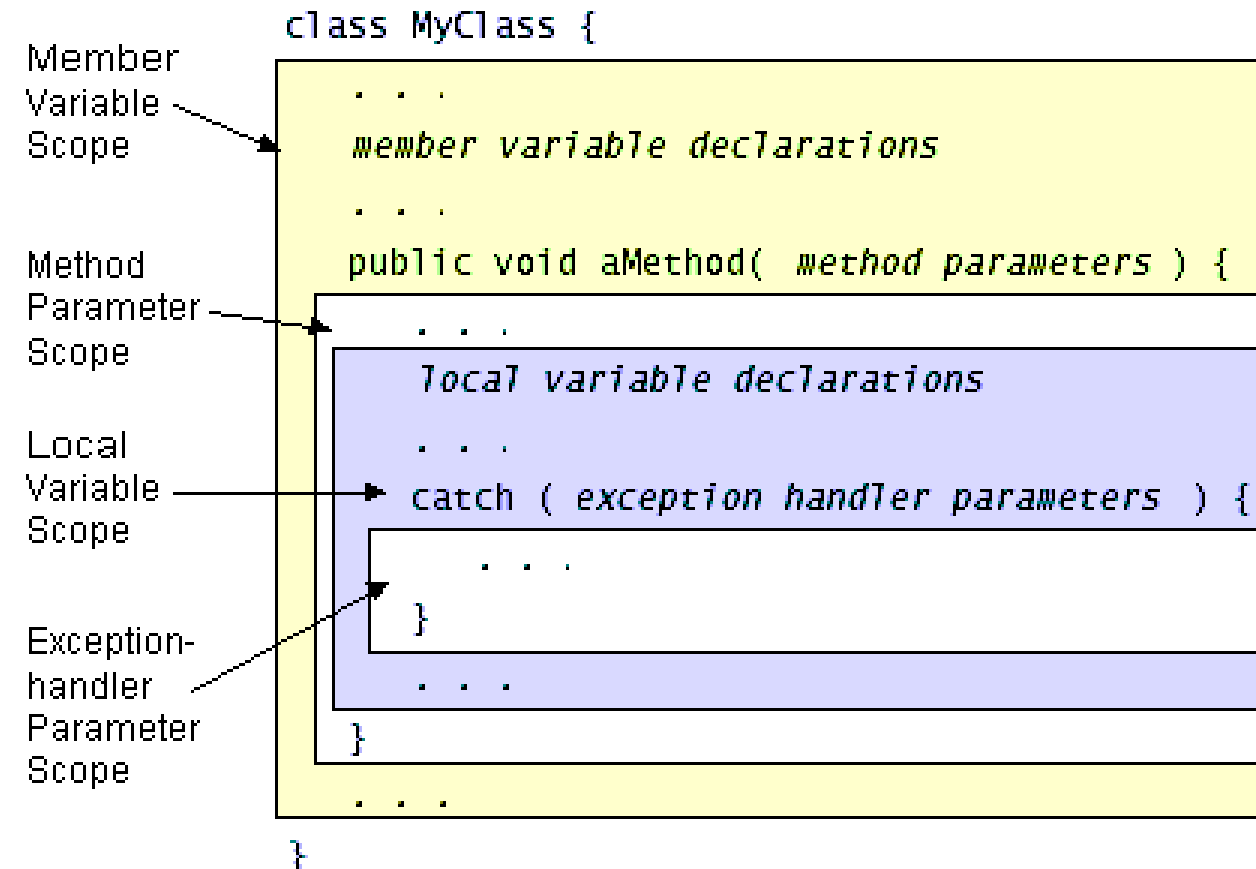
    // Konstruktor bezargumentowy o pustej implementacji:
    //    A(){
    // jest dostarczany domyślnie jedynie wtedy,
    // gdy w kodzie nie zdefiniowano żadnego konstruktora.

    A(int i){                // Tutaj mamy konstruktor z jednym argumentem (dlatego bezargumentowy się nie pojawi).
                               // W pierwszej linii konstruktora, choć tego jawnie nie zadeklarowano,
                               // zawsze odpala się bezargumentowy konstruktor klasy nadrzędnej.
                               // Zachowanie to można zmienić, jawnie wywołując w pierwszej linii
                               // wybrany (tylko jeden) konstruktor, przy czym może to być:
                               // - konstruktor klasy nadrzędnej, np. super(),
                               // - inny konstruktor klasy bieżącej, np. this().
                               // Wywołanie konstruktora nie może być wywołaniem rekurencyjnym.
                               // Jeśli wywoływany jest konstruktor z argumentami, należy te argumenty
                               // umieścić w nawiasach, np. super(10), this(10).
                               // Przez this można wnieź odwołać się do instancji.

        this.i = i;
    }

}
```

Zasięg zmiennych i modyfikatory dostępu



- Dostęp:
 - **public**,
 - **private**,
 - **protected**
 - **package**

- Zasięg:
 - parametr klasy,
 - zmienna lokalna,
 - parametr metody,
 - obsługa wyjątku,
 - blok {}

Typ wyliczeniowy

- Typ wyliczeniowy deklaruje się podobnie jak klasę, tyle że z wykorzystaniem słowa `enum` zamiast `class`

```
package other;
```

```
public enum Size {  
    S, M, L, X, XL, XXL  
}
```

- Można w tym typie tworzyć metody, pola oraz implementować interfejsy
- Konstruktor typu wyliczeniowego zawsze jest **prywatny** (w deklaracji można pominąć modyfikator dostępu, ale użycie innego modyfikatora to błąd)
- Typy wyliczeniowe nie mogą dziedziczyć po sobie
- Elementy typu wyliczeniowego są uporządkowane
- Elementy typu wyliczeniowego można wykorzystywać w instrukcji `switch`
- Każda zmienna typu wyliczeniowego
 - posiada metody klasy `Object`.
 - implementuje interfejsy `Comparable` oraz `Serializable`,
- Do typów wyliczeniowych można stosować metody statyczne klas pomocniczych (`EnumSet`, `EnumMap` z `java.util`), jak na przykład `EnumSet.range()`, która zwraca tablicę zawierającą wszystkie wartości typu `enum` w porządku ich zadeklarowania.

```
for (Size d : EnumSet.range(Size.S, Size.XXL))  
    System.out.println(d);
```

- Typ wyliczeniowy dostarcza metody statyczne `values()` oraz `valueOf()`

Typ wyliczeniowy

- Typ wyliczeniowy kompilowany jest do klasy z finalnymi polami statycznymi. Można się o tym przekonać kompilując klasę, a potem ją dekompilując:

```
> javac -d bin src\other\Size.java
> javap -cp bin other.Size
Compiled from "Size.java"
public final class other.Size extends
java.lang.Enum<other.Size> {
    public static final other.Size S;
    public static final other.Size M;
    public static final other.Size L;
    public static final other.Size X;
    public static final other.Size XL;
    public static final other.Size XXL;
    public static other.Size[] values();
    public static other.Size valueOf(java.lang.String);
    static {};
}
```

Typ wyliczeniowy

- Jeśli typ wyliczeniowy ma metody i pola, to lista elementów typu wyliczeniowego musi kończyć się średnikiem

```
public enum Planet {
    MERCURY (3.303e+23, 2.4397e6),
    VENUS (4.869e+24, 6.0518e6),
    EARTH (5.976e+24, 6.37814e6),
    MARS (6.421e+23, 3.3972e6),
    JUPITER (1.9e+27, 7.1492e7),
    SATURN (5.688e+26, 6.0268e7),
    URANUS (8.686e+25, 2.5559e7),
    NEPTUNE (1.024e+26, 2.4746e7),
    PLUTO (1.27e+22, 1.137e6);
    private final double mass; //in kilograms
    private final double radius; //in meters
    Planet(double mass, double radius) {
        this.mass = mass;
        this.radius = radius;
    }
    public double mass() { return mass; }
    public double radius() { return radius; }
    public static final double G = 6.67300E-11; //universal gravitational constant
    (m3 kg-1 s-2)
    public double surfaceGravity() {
        return G * mass / (radius * radius);
    }
    public double surfaceWeight (double otherMass) {
        return otherMass * surfaceGravity();
    }
}
```

Instrukcja switch

- lista wartości w warunku (*multiple values per case*)(od JDK 13)

```
Size d = Size.X;
switch (d) {
    case S,M,L: System.out.println(d + " - less than X"); break;
    case X:
    case XL:
    case XXL: System.out.println(d + " - greather or equal X"); break;
    default: System.out.println("Will never happen");
}
```

- wartość wyliczana w warunku (*switch expression*)(w JDK 12)

```
int i = 1;
char out = switch(i){
    case 0 -> '0';
    case 1 -> '1';
}
```

- wartość zwracana z warunku (*switch yield*)(od JDK 13)

```
String token = "123";

int tokenType = switch(token) {
    case "123" : yield 0;
    case "abc" : yield 1;
    default : yield -1;
};
```

Rekordy i zapieczętowane klasy

- Rekordy
 - skrócony zapis klas z polami finalnymi

<https://docs.oracle.com/en/java/javase/16/language/records.html>
- Zapieczętowane klasy
 - klasy, w których deklaracji użyto wyrażeń pozwalających ograniczyć sposób ich wykorzystania (dziedziczenia)

<https://docs.oracle.com/en/java/javase/16/language/sealed-classes-and-interfaces.html>

<https://javastart.pl/baza-wiedzy/slownik/sealed-classes>

Java records

```
package other;
```

```
public final class A {  
    private final int x;  
    private final int y;
```

```
    public A(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }
```

```
    public int x() {  
        return this.x;  
    }
```

```
    public int y() {  
        return this.y;  
    }
```

```
@Override  
    public String toString() {  
        return "A[x=" + x + ", y=" + y + "];"  
    }
```

```
@Override  
    public int hashCode() {  
        return super.hashCode();  
    }
```

```
@Override  
    public boolean equals(Object obj) {  
        if (obj instanceof A) {  
            return x == ((A) obj).x() && y == ((A) obj).y();  
        }  
        return false;  
    }  
}
```

```
package other;
```

```
record B(int x, int y) {  
}
```

prawie równoważne, z dokładnością do:
hashCode(), equals()

```
> javap B.class  
Compiled from "B.java"  
final class other.B extends java.lang.Record {  
    other.B(int, int);  
    public int x();  
    public int y();  
    public final java.lang.String toString();  
    public final int hashCode();  
    public final boolean equals(java.lang.Object);  
}
```

<https://docs.oracle.com/en/java/javase/16/language/records.html>

[https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/Record.html#hashCode\(\)](https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/Record.html#hashCode())