METODY

Metoda (funkcja) to blok kodu, który ma nazwę.
Do metody możemy przekazywać dane (argumenty), a metoda może zwracać wynik.

Zmienne zadeklarowane w nagłówku metody nazywają sie parametrami.
Konkretne wartości (np.liczby) przekazywane do metody w momencie jej wywołania to argumenty.

Przykłady deklaracji metod:

```
void metoda1() {
// słowo "void" oznacza, że nie zwraca wyniku
// pusty nawias - nie ma parametrów
}
 void metoda2(int liczba){
// nie zwraca wyniku, ma jeden parametr
}
int metoda3(){
// zwraca wynik, nie ma parametrów
return 10 ;
}
int metoda4(int liczba, int liczba2){
// zwraca wynik i ma dwa parametry
return 10;
}
```

Aby metoda została wykonana musimy podać jej nazwę oraz argumenty, jeśli są wymagane.
np. metoda4(2, 6);

Każda metoda musi mieć inną nazwę.
Dopuszczalne jest, żeby metoda wystepowała w kilku wariantach pod tą samą nazwą,
ale wtedy każdy wariant musi mieć inne parametry.
Mówimy wtedy, że metoda jest przeciążona.

Metod wykonują pewne działania, więc ich nazwy najczęściej są czasownikami.

```
public static void main(String[] args) {
    System.out.println(obliczSume(2, 2));
    System.out.println(obliczSume(2.5f, 2.5f));
}

public static int obliczSume(int liczba1, int liczba2) {
    return liczba1 + liczba2;
}

public static float obliczSume(float liczba1, float liczba2) {
    return liczba1 + liczba2;
}
```

Klasy - wstęp

Na klasę składają się dane i działania.
Dane (atrybuty obiektu) opisują obiekt,
działania określają co robi obiekt.

Klasa jest szablonem, według którego powstają obiekty.
Obiekty tworzymy przy użyciu słowa kluczowego "new".
Z jednej klasy można utworzyć wiele obiektów np. z klasy String tworzymy wiele łańcuchów.

Każda klasa zajmuje się jednym rodzajem zadań i zapisywana jest w oddzielnym pliku.
Tworzymy więc klasy Prostokat, Trojkat, Kolo, a nie jedną klasę ze wszystkimi figurami.

Program składa się z wielu obiektów, które ze sobą współpracują, aby wykonać określone zadanie.

Klasy - konstruktory

```
public class Nauka10Klasy_Konstruktory {

    public static void main(String[] args) {
        new Klasa2(5);
        new Klasa2();
    }
}

class Klasa2 {

    - Aby powstał obiekt potrzebny jest konstruktor, a więc specjalna metoda,
    która jest wywoływana przy tworzeniu obiektu.
    - Konstruktor ma taką samą nazwę jak klasa.
    - Jeśli nie zdefiniujemy konstruktora, to automatycznie zostanie utworzony konstruktor bezargumentowy.
    - Klasa może mieć kilka konstruktorów i wtedy użyjemy tego, który najlepiej nam odpowiada.
     */
    // Pierwszy konstruktor
    public Klasa2(int liczba) {
        System.out.println("Konstruktor 1 " + liczba);
    }

    // Drugi konstruktor
    public Klasa2() {
        this(0);  // wywołanie innego konstruktora - tylko w pierwszej linii
        System.out.println("Konstruktor 2 " + 0);
    }
}
```

Klasy - kwantyfikatory

```
    /*
       Zmienne i metody w klasie mogą mieć kwantyfikatory określające gdzie można ich użyć.
       Mamy 4 możliwości:
       private, public, protected lub brak jakiegokolwiek kwantyfikatora.
     */
    public static void main(String[] args) {
       Klasa3 k3 = new Klasa3();
       // klasa2.liczba1; // pole prywatne nie jest dostępne w innej klasie
       k3.liczba2 = 5; // pole publiczne jest dostępne w innej klassie
       System.out.println(k3.liczba2);

       // klasa3.metodaPrywatna(); metoda prywatna nie jest dostępna w innej klasie
       k3.metodaPubliczna(); // metoda publiczna jest dostępna w innej klassie
    }
}

class Klasa3 {

    private int liczba1 = 1;    // wszystkie zmienne należy deklarować jako prywatne
    public int liczba2 = 2;     // zmienne publiczne mogą prowadzić do problemów

    // Metoda prywatna używana jest tylko wewnątrz danej klasy
    private void metodaPrywatna() {
       System.out.println("Metoda prywatana ");
    }

    // Metoda publiczna może być używana w innych klasach
    public void metodaPubliczna() {
       System.out.println("Metoda publiczna ");
    }
}
```

Klasy – zmienne statyczne

```
/*
   Każdy obiekt ma swoje własne zmienne. Jeśli jednak chcemy, aby jakaś zmienna
   była wspólna (tylko jedna) dla wszystkich obiektów, to oznaczamy ją słowem kluczowym
"static".
   Mówimy, że jest to zmienna klasowa lub statyczna.
   Zmienna lub metoda statyczna może być użyta bez konieczności tworzenia obiektów.
   Gdy wywołujemy taką zmienną lub metodę to piszemy nazwę klasy, a nie obiektu np.
   Klasa4.metodaStatyczna() zamiast obiekt.metodaStatyczna().
 */
public class Nauka10Klasy_Statyczne {

   public static void main(String[] args) {
      Klasa4.metodaStatyczna(); // Metody statycznej można użyć nawet, gdy nie ma obiektów
      Klasa4 k4 = new Klasa4();
      Klasa4.zwiekszIlosc(); // piszemy Klasa4.zwiekszIlosc() a nie k4.zwiekszIlosc()
      System.out.println(Klasa4.podajIlosc());
   }
}

class Klasa4 {

   private static int iloscObiektow = 0;
   // Metody statycznej można użyć bez tworzenia obiektu

   public static void metodaStatyczna() {
      System.out.println("Metoda statyczna");
   }

   public static void zwiekszIlosc() {
      iloscObiektow++;
   }

   public static int podajIlosc() {
      return iloscObiektow;
   }
}
```

Klasy – dziedziczenie

```
    public static void main(String[] args) {
        KlasaA obiektA = new KlasaA();
        obiektA.drukujA();
        KlasaB obiektB = new KlasaB();
        obiektB.drukujB();
        obiektB.drukujA();
    }


class KlasaA {

    public void drukujA() {
        System.out.println("Wydruk  A");
    }
}

/* Jeśli jedna klasa rozszerza drugą to dziedziczy po niej wszystkie właściwości i metody.
Klasa rozszerzająca może używać wszystkich zmiennych i metod publicznech z klasy bazowej.
Klasę bazową nazywamy nadklasą, a klasę pochodną podklasą.
Wszystkie klasy w Javie automatycznie dziedziczą po klasie Object.
 */
class KlasaB extends KlasaA {

    public void drukujB() {
        System.out.println("Wydruk  B");

    }
}
```

Klasy – przesłanianie metod

```
class Klasa2 extends Klasa1 {

    /* W podklasie można zastąpić istniejącą metodę z nadklasy. Mówimy, że metoda z nadklasy
zostaje przesłonięta.
    Aby mieć pewność, że zastępujemy właściwą metodę, można dodać specjalny opis ze znakiem
"@" - adnotację.
    Kompilator wtedy sprawdzi, czy nazwa, parametry i zwracany typ jest taki sam jak w metodzie z
nadklady.
     */
    @Override
    public void drukuj() {
        System.out.println("Wydruk  2");
    }
```

Klasy – opakowujące

Każdy typ podstawowy ma odpowiadającą mu klasę opakowującą.
(Różne nazwy tych klas: opakowujące, osłonowe, otoczkowe, nakładkowe.)

Byte - byte
Short - short
Integer - int
Long - long

Float - float
Double - double

Boolean - boolean
Character - char

Dzięki tym klasom proste dane mogą być używane jako obiekty,
np. listy tablicowe używają tylko obiektów.

```
Klasy te zawierają przydatne metody.
 */
public static void main(String[] args) {
   /* Każda klasa opakowująca dla typów liczbowych posiada metodę toString(),
   która zamienia liczbę na łańcuch znaków.
   Integer.toString( liczba)
   Float.toString( liczba)
   Double.toString( liczba)
    */

   int i = 12;
   String tekstInt = Integer.toString(i);
   double d = 17.21;
   String tekstDouble = Double.toString(d);

   /*
   Każda klasa opakowująca może dokonać zamiany
   - łańcucha znaków na typ podstawowy - metody parseXXX()
   - łańcucha znaków na typ obiektowy - metody valueOf()

   Zamiana łańcucha znaków na typ prosty:
      Integer.parse( łańcuch) - zamiana na int
      Float.parse( łańcuch)  - zamiana na float
   Przy zamianie trzeba uważać, aby łańcuch zawierał poprawną liczbę.
    */
   String rok = "2000";
   int liczbaInt = Integer.parseInt(rok);
   float liczbaFloat = Float.parseFloat(rok);

   /*
   Zamiana łańcucha znaków na typ obiektowy
      Integer.valueOf( łańcuch) - zamiana na Integer
```

```
      Float.valueOf( łańcuch) - zamiana na Float
    Przy zamianie trzeba uważać, aby łańcuch zawierał poprawną liczbę.
     */
    Integer liczbaIntegerObiekt = Integer.valueOf(rok);
    Float liczbaFloatObiekt = Float.valueOf(rok);

    /*
    Przy pomocy metody valueOf() (metoda przeciążona) można też dokonać
    zamiany typu podstawowego na typ obiektowy.
      Integer.valueOf( typ podstawowy) - zamiana na Integer
      Float.valueOf( typ podstawowy) - zamiana na Float
     */
    liczbaIntegerObiekt = Integer.valueOf(100);
    liczbaFloatObiekt = Float.valueOf(liczbaFloat);

    /*
    Zamiana typu podstawowego na obiektowy i odwrotnie odbywa się automatycznie,
    więc wywołanie metody valueOf(typ podstawowy) nie jest potrzebne.
     */
    Integer ilosc = 10;  // automatyczna zamiana typu int na Integer
    int ilosc2 = ilosc + 2; // automatyczna zamiana typu Integer na int
    Float cena = 20.5f;
    float cena2 = cena * 1.1f;
}
```

Listy tablicowe

```
// Tablice
	int[] tablicaProsta = new int[110];
	Integer[] tablicaObiektowa = new Integer[110];

	/* Listy tablicowe to struktury, które służą do przechowywania obiektów.
	Podobne są do tablic, ale mogą zwiększać i zmniejszać swoje rozmiary,
	można dodawać i usuwać elementy w dowolnym miejscu listy.
	 */
	// Deklaracja zmiennej listy tablicowej.
	// W nawiasach ostrych należy podać typ obiektów, które będą przechowywane.
	ArrayList<Integer> numery1 = new ArrayList<Integer>();
	ArrayList<Double> numery2 = new ArrayList<Double>(50);
	ArrayList<String> lista = new ArrayList<>();

	// Dodawanie elementów do listy
	lista.add("AAA");
	lista.add("BBB");
	lista.add("CCC");
	String tekst = "DDD";
	lista.add(tekst);
	System.out.println("Nowa lista " + lista.toString());

	// Wstawienie elementu w środku listy
	lista.add(1, "W1");
	System.out.println("Wstawienie elementu " + lista.toString());

	// Zastąpienie istniejącego elementu
	lista.set(2, "Z2");
	System.out.println("Zastąpienie " + lista.toString());

	// Usunięcie istniejącego elementu (należy podać numer elementu lub ten element)
	lista.remove(1);
	System.out.println("Usunięcie " + lista.toString());

	// Odczytanie rozmiaru listy
	lista.size();
	System.out.println("Ilość elementów na liście: " + lista.size());

	// Pobranie elementu
	lista.get(0);
	System.out.println("Element numer 0: " + lista.get(0));

	// Sprawdzenie na której pozycji znajduje się element
	lista.indexOf("ooo");
	System.out.println("Numer elementu \"ooo\": " + lista.indexOf("ooo"));
	lista.indexOf(tekst);
	System.out.println("Numer elementu tekst: " + lista.indexOf(tekst));

	lista.clear(); // Usunięcie wszystkich elementów
```

```
    lista.isEmpty(); // Sprawdenie czy lista jest pusta
    System.out.println("Czy lista jest pusta: " + lista.isEmpty());
}
```

Wyjątki

```
/*
   Jeśli program w czasie działania napotka problem, to zostanie wygenerowany
   wyjątek - obiekt i informacja o rodzaju błędu.
   Jeśli w programie nie przewidziano żadnego działania związanego
   z tym wyjątkiem, to program się zatrzyma.

   Przed zatrzymaniem się programu można zabezpieczyć się na dwa sposoby:
   1) przechwycić wyjątek, czyli
   umieścić kod, który może spowodować błąd, w bloku instrukcji "try-catch"
   2) zgłosić wyjątek, czyli
   przekazać informację o wyjątku do wywołującej metody przy użyciu słowa kluczowego
"throws".
   Metoda wywołująca bedzie musiała obsłużyć wyjątek.
    */
   public static void main(String[] args) {

      /* Będziemy odczytywać plik przy pomocy klasy Scanner
      File plik = new File("Mój plik.txt");
      Scanner odczytPliku = new Scanner(plik);
       */
      odczytPliku1("PlikABC.txt");
//       odczytPliku2("PlikABC.txt"); // wyjątek musi być przechwycony lub przekazany dalej
      try {
         odczytPliku2("PlikABC.txt");
      } catch (FileNotFoundException wyjatek) {
         System.out.println("Nie znaleziono pliku " + wyjatek);
      }
   }

   private static void odczytPliku1(String nazwaPliku) {
      try {
         File plik = new File(nazwaPliku);
         Scanner odczytPliku = new Scanner(plik);
      } catch (FileNotFoundException wyjatek) {
         System.out.println("odczyt1 Nie znaleziono pliku " + wyjatek);
      }
   }

   private static void odczytPliku2(String nazwaPliku) throws FileNotFoundException {
      File plik = new File("Mój plik.txt");
      Scanner odczytPliku = new Scanner(plik);
   }
   /*
   W tej sytuacji lepszym rozwiązaniem jest przekazanie wyjątku, aby metoda "main"
   wiedziała, że odczyt z pliku nie powiódł się.
    */
}
```

Wyjątki - łaczenie

```
// Pojedyncze instrukcje "catch" dla każdego wyjątku
try {
    File plik = new File(nazwaPliku);
    Scanner odczytPliku = new Scanner(plik);
    odczytPliku.nextLine();
} catch (FileNotFoundException wyjatek1) {
    System.out.println("Nie znaleziono pliku " + wyjatek1);
} catch (NoSuchElementException wyjatek2) {
    System.out.println("Brak linii do odczytu " + wyjatek2);
}

// Jedna instrukcja "catch" - połaczenie wyjątków
try {
    File plik = new File(nazwaPliku);
    Scanner odczytPliku = new Scanner(plik);
    odczytPliku.nextLine();
} catch (FileNotFoundException | NoSuchElementException wyjatek) {
    System.out.println("Bład odczytu pliku " + wyjatek);
}
```

Wyjątki - finally

```
try {
    File plik = new File(nazwaPliku);
    Scanner odczytPliku = new Scanner(plik);
} catch (FileNotFoundException wyjatek) {
    System.out.println("Nie znaleziono pliku " + wyjatek);
} finally {
    System.out.println("Wydruk w obu sytuacjach - gdy jest błąd lub go nie ma");
}
```