

JAVA CHEAT SHEET

KierunekJava.pl

WPROWADZENIE

Dzięki Javie można przygotować wiele różnych typów aplikacji, takich jak mobilne, desktopowe, czy webowe. Java wywodzi się głównie z dwóch popularnych języków programowania C++ oraz Smalltalk, czerpiąc z nich to, co najlepsze.

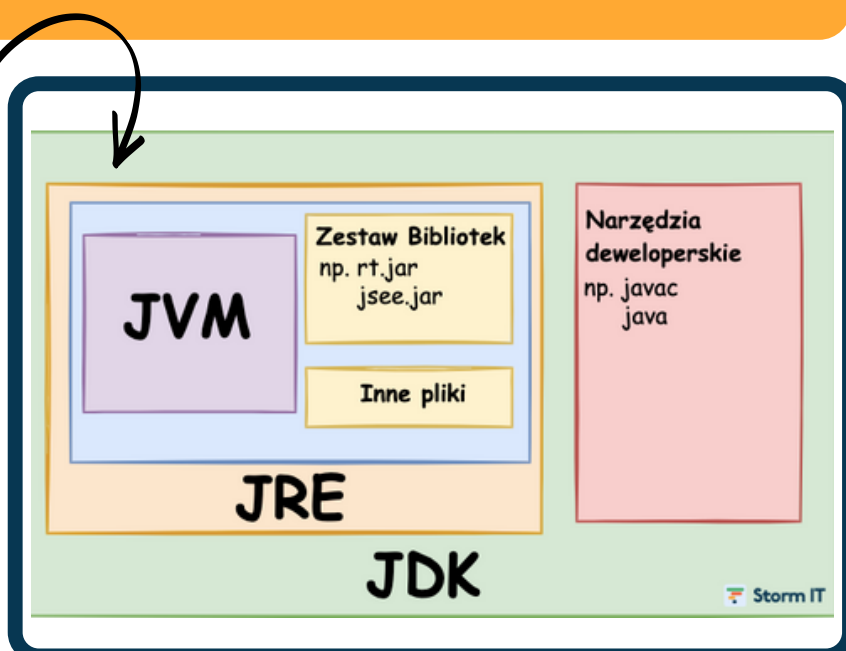
Język ten został zaprojektowany i zaimplementowany w laboratoriach Sun Microsystems (w 2010 r. Sun został przejęty przez Oracle) w Mountain View (Kalifornia) pod kierownictwem Jamesa Goslinga. James Gosling jest również autorem programu emacs pod UNIX oraz systemu okien NeWS.

EKOSYSTEM JAVA

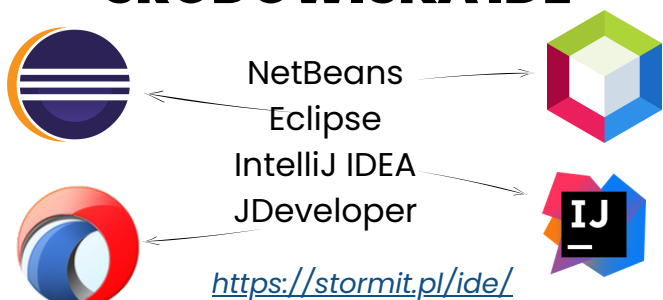
Java Runtime Environment (JRE) – Maszyna wirtualna (JVM) + biblioteki
Java Development Kit (JDK) = JRE + narzędzia deweloperskie:

- *javac* – kompilator java
- *java* – maszyna wirtualna Javy
- *jar* – obsługa archiwów
- *jdb* – debugger klas Java
- *javadoc* – generator dokumentacji klas w formacie HTML

<https://stormit.pl/instalacja-jdk/>



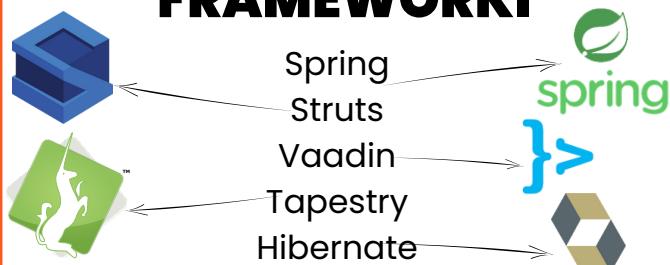
ŚRODOWISKA IDE



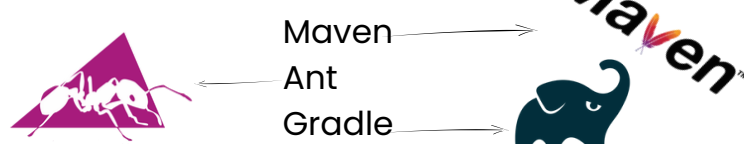
SERWERY APLIKACJI



FRAMEWORKI



NARZĘDZIA DO BUDOWANIA PROJEKTÓW



ARCHITEKTURA APLIKACJI JAVA

Kod programu w języku Java pogrupowany jest w **klasy**, które z kolei pogrupowane są w **pakiety**. Klasy mogą składać się z różnych elementów takich jak np. **zmienne**, **stałe** czy **metody** (nazywane również funkcjami w innych językach programowania).

Taka modułowość w Javie pozwala nam rozbić duże programy na mniejsze bloki konstrukcyjne, które są łatwiejsze do zrozumienia i ponownego użycia.

METODA MAIN

Metoda *main* to najważniejsza metoda w Javie, od której rozpoczyna się wykonanie całego programu. To punkt „wejściowy”, gdzie zaczyna się nasza aplikacja.

Metoda ta musi posiadać odpowiednią sygnaturę. W przeciwnym wypadku zostanie potraktowana jak każda inna metoda

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



- **public** – modyfikator dostępu, który oznacza, że metoda ta jest dostępna dla wszystkich klas, a także co najważniejsze dla środowiska uruchomieniowego Javy (ang. Java runtime),
- **static** – dla metod statycznych nie musimy tworzyć obiektu danej klasy, aby ją wywołać,
- **void** – typ zwracany przez metodę, w tym wypadku metoda nie zwraca nic (nie występuje słowo kluczowe return),
- **main** – nazwa metody. Wymagana jest dokładnie taka nazwa,
- **(String[] args) / (String... args) / (String args[])** – metoda main przyjmuje tylko jeden parametr, którym jest tablica Stringów lub varargs o typie String. Nazwa parametru może być inna, jednak przyjęło się i używa się najczęściej nazwy „args”.

KLASA

Klasa to zdefiniowany przez użytkownika swego rodzaju **schemat (prototyp)**, na podstawie którego tworzone są **obiekty**. Przykładowo klasą może być „człowiek”, a obiektami, czyli instancjami tej klasy może być „Ania”, „Tomek”, „Magda” itp.

Klasa reprezentuje zestaw właściwości i metod, które są wspólne dla wszystkich obiektów jednego typu.

Klasę w najprostszej swej postaci tworzy się poprzez użycie słowa kluczowego **class** oraz nazwy tej klasy.

Klasa posiada również ciało zdefiniowane w nawiasach klamrowych {}, gdzie umieszczamy jej zmienne oraz metody.

```
1 class Person {  
2     String name;  
3     String getName() {  
4         return name;  
5     }  
6 }
```

OBIEKT

Obiekt jest **konkretną instancją danej klasy**. Rozpatrując klasę jako szablon, można przyjąć, że każdy obiekt danej klasy jest elementem utworzonym na podstawie danego szablonu.

Wszystkie właściwości i metody klasy, są wspólne dla obiektów jednego typu. Przykładowo wszystkie osoby posiadają swoje dane, takie jak imię, nazwisko, wiek itp. oraz zachowanie np. wszystkie osoby mogą chodzić, spać, jeść itp.

<https://stormit.pl/typy-obiektowe/>

CECHY OBIEKTU

Obiekt posiada takie cechy jak:

- **Stan:** reprezentuje dane (wartości) obiektu np. wiek psa, rasa psa itp.
- **Zachowanie:** reprezentuje zachowanie (funkcjonalność) obiektu, takie jak np. szczekanie, warczenie.
- **Adres w pamięci:** do którego każdy obiekt jest przypisany tzw. referencja obiektu.

```
1 | Person person = new Person();  
2 | System.out.println(person.getName());
```

KOMENTARZE

Komentarze to swego rodzaju zapiski programisty w kodzie aplikacji.

Ich celem jest przekazanie jakiejś informacji dla siebie samego lub dla innych osób czytających ten kod.

Komentarze **są widoczne tylko w kodzie źródłowym** i przy kompilacji są pomijane, dlatego nie mają wpływu na samo działanie wynikowej aplikacji. Jednak jest to integralna część każdego programu i nie powinna być pomijana. Dzięki nim można lepiej zrozumieć intencje programisty, co jak w praktyce się okazuje, nie zawsze jest takie oczywiste

<https://stormit.pl/komentarze-java/>

TYPY KOMENTARZY

W Javie mamy dwa typy komentarzy:

- Do końca linii
- Blokowe – czyli takie, które mogą zawierać kilka linii

```
1 | //Komentarz do końca linii I.  
2 | System.out.println("Hello"); //Komentarz do końca linii II.  
3 | //Komentarz do końca linii III. System.out.println("Hi");
```

```
1 | /*  
2 | Komentarz blokowy  
3 | */  
4 |  
5 | /*  
6 | *  
7 | * Komentarz blokowy  
8 | * System.out.println("Hello");  
9 | *  
10 | */
```



KOMENTARZE – DOBRE PRAKTYKI

Komentarze służą poprawie czytelności kodu. Dlatego nie powinno się dodawać ich na siłę do każdego fragmentu kodu, a jedynie w miejscach, gdzie czujemy, że przydałoby się jakieś wyjaśnienie.

KOMENTARZE – DOBRE PRAKTYKI C.D.

Idąc krok dalej, można powiedzieć, że gdzie tylko to możliwe **komentarze powinny być zastępowane czytelnym kodem**, który nie wymaga dodatkowego objaśnienia.

Powstaje w ten sposób tak zwany **samodokumentujący się kod**.

Przykładowo, dużo lepiej jest nadać metodzie opisową nazwę niż wyjaśniać w komentarzu, co programista miał na myśli.

PRZYKŁAD

```
1 // <przykład złego komentarza>
2 // metoda oblicza obwód kwadratu
3 int x = calculate(2);
4
5 //<przykład samodokumentującego się kodu>
6 int squarePerimeter = calculateSquarePerimeter(2);
```



KOMENTARZE – NA CO UWAŻAĆ?

Przy stosowaniu komentarzy blokowych należy pamiętać, że nie można ich zagnieżdżać, tzn. **jeżeli spróbujemy wstawić jeden komentarz w drugim, to kompilator potraktuje koniec drugiego komentarza, jako koniec całego bloku**, a pozostały fragment spowoduje błąd.

Komentarze do końca linii, jeżeli zostaną wstawione w niewłaściwym miejscu, również mogą powodować kłopoty np. jeżeli zostaną wstawione między instrukcją `if` a rozpoczynającą klamrą to po zmianie formatowania kodu **może dojść do zakomentowania tej klamry i błędów kompilacji**.

```
/**
 *
 * /* próba wstawienia komentarza blokowego w komentarzu blokowym */
 *
 */
```



```
if (x > 0) // komentarz przed klamrą - przed formatowaniem
{
}
if (x > 0) // komentarz przed klamrą - po formatowaniu {
}
```

PSEUDOKOD

Dość często spotykaną praktyką jest najpierw pisanie komentarzy, a dopiero na ich podstawie tworzenie właściwego kodu. Powstaje w ten sposób tak zwany: pseudokod – czyli **fragmenty kodu pomieszczone z komentarzami**.

Rezygnuje się w nim ze ścisłych reguł składniowych narzuconych przez kompilator na korzyść prostoty zapisu i czytelności. Pomijane są również szczegóły implementacyjne.

Dzięki temu bardzo szybko można spisać algorytm i dopiero w kolejnych krokach po kawałku zamieniać go na właściwą implementację.

```
if [podano bok kwadratu]
    [oblicz obwód]
else [wyświetl komunikat o błędzie]
```

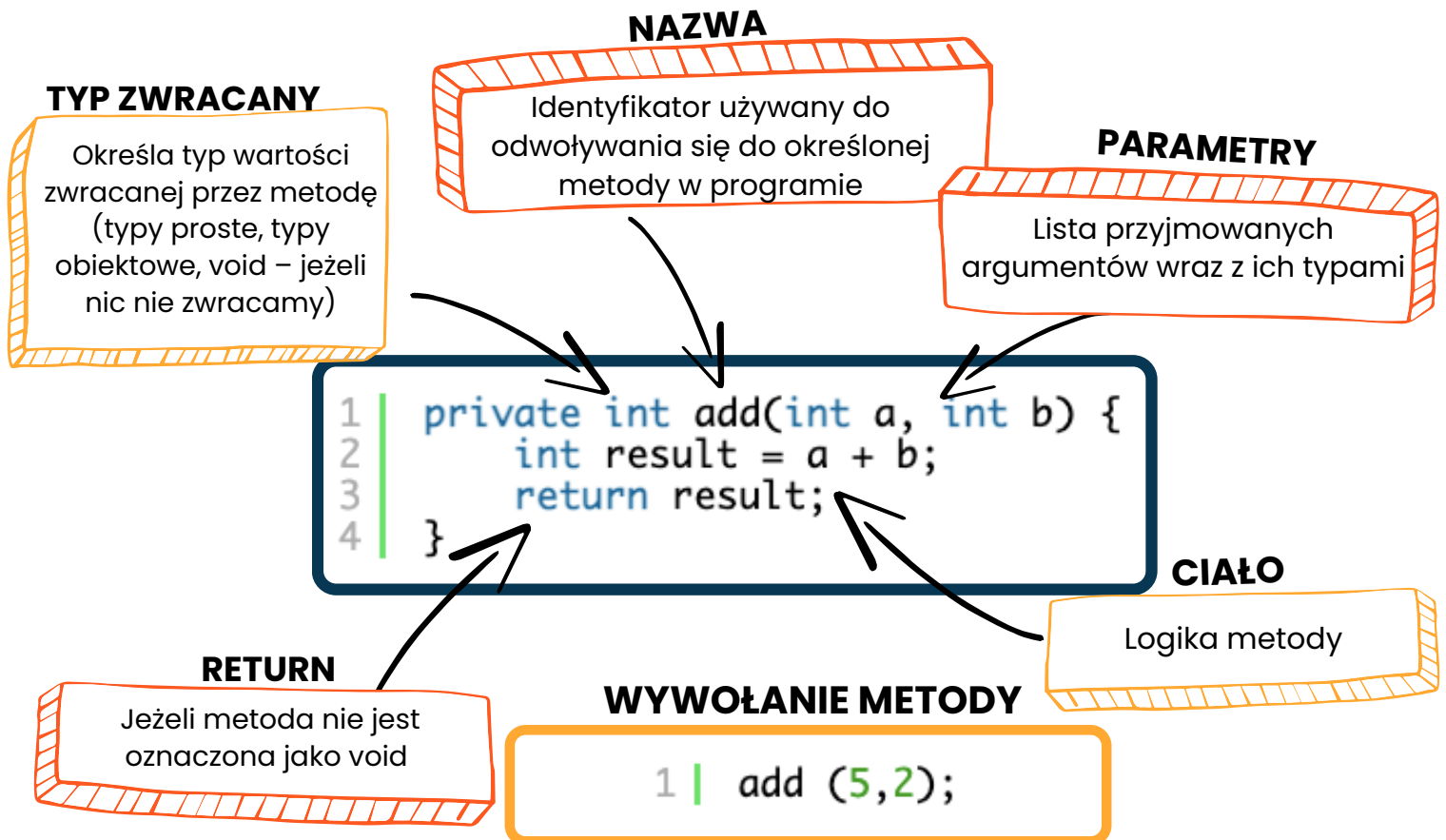


METODA

Metoda to blok kodu, który działa tylko wtedy, gdy dana metoda zostanie wywołana i służy do wykonywania określonych czynności (fragmentu logiki). Kod w ramach metody wykonywany jest linijka po linijce od góry do dołu – Jeżeli jednak Java napotka wywołanie innej metody, to będzie musiała ją też wywołać.

Możemy wyróżnić dwa typy metod:

- Metody zdefiniowane przez użytkownika – czyli nas :)
- Metody biblioteki standardowej lub innych zależności (metody przygotowane przez innych deweloperów)



KONSTRUKTOR

Specjalny typ metody wywoływanej podczas tworzenia nowego obiektu i wykorzystywany do inicjowania np. atrybutów.

CECHY KONSTRUKTORA

Konstruktor cechuje:

- Operator *new* wywołuje konstruktor
- Brak zwracanego typu
- Nazwa konstruktora musi zgadzać się z nazwą klasy

Kompilator w przypadku braku jawnego konstruktora dla wygody programistów spróbuje dodać konstruktor domyślny (bezargumentowy)

KONSTRUKTOR C.D.

```
1 public class User {
2     String name;
3     int age;
4
5     public User(String name) {
6         this.name = name;
7     }
8
9     public User(String name, int age) {
10        this.name = name;
11        this.age = age;
12    }
13 }
```

```
1 public class Animal {
2     public Animal() {
3     }
4 }
```

PRZECIĄŻENIE KONSTRUKTORA

Możliwość stworzenia w klasie, większej ilości konstruktorów niż jeden, jeżeli posiadają inny zestaw argumentów.

ŁAŃCUCH WYWOŁAŃ KONSTRUKTORÓW

Możliwość wywołania innego konstruktora z poziomu naszego konstruktora – Musi to jednak zadziać się jako pierwsza instrukcja

```
1 public User(String name) {
2     this.name = name;
3 }
4
5 public User(int age) {
6     this("Tomek");
7 }
```

PRZETWARZANIE DANYCH

Działanie praktycznie każdego z programów, ogólnie mówiąc, sprowadza się do przetwarzania danych. Dlatego jedną z podstawowych umiejętności, jaką powinien posiadać programista, jest znajomość typów danych oraz możliwości ich obróbki.

TYPY DANYCH

W każdym programie dane możemy przedstawić za pomocą literałów, zmiennych oraz stałych.

- **literał** – to napis w programie bezpośrednio przedstawiający wartość danej, np. liczba: 123 albo ciąg znaków: *Hello World*.
- **zmienna** – jest to natomiast symbol oznaczający wielkość, która może przyjmować różne wartości. Natomiast zbiór możliwych wartości jest to zakres tej zmiennej. Mówiąc bardziej technicznie: to, że zmienna jest symbolem oznacza, że jest swego rodzaju wskaźnikiem kierującym na obszar w pamięci komputera, w którym są zapisane różne dane, np. liczba.
- **stała** – jest również symbolem oznaczającym pewną wielkość, jednak, w przeciwieństwie do zmiennej, nie może zmieniać swojej wartości podczas działania programu.

TYPY DANYCH C.D.

ZMIENNE

Java jest językiem silnie typowanym, czyli każda zmienna musi być określonego typu. Poszczególne typy danych zostaną omówione za chwilę, tymczasem w przykładzie posłużymy się typem dla liczb całkowitych: `int`.

<https://stormit.pl/zmienne/>

DEKLARACJA ZMIENNYCH

W celu utworzenia zmiennej musimy ją najpierw zadeklarować, czyli podać jej typ, nadać jej nazwę oraz, opcjonalnie, zainicjować ją, czyli przypisać jej wartość.


```
[typ zmiennej] nazwaZmiennej = wartość;  
[typ zmiennej] nazwaZmiennej;  
nazwaZmiennej = wartość;  
nazwaZmiennej = nowaWartość;
```

Zmienną można jednocześnie zadeklarować i zainicjować w jednej linii, ale można to również rozbić na dwa kroki.

TYP ZMIENNYCH

Raz nadany **typ dla zmiennej nie może zostać już zmieniony**, czyli jeżeli zadeklarujemy zmienną jako typ `int` (liczba całkowita), to nigdy nie będzie można w niej już przechować, np. tablicy, czy ciągu znaków. Wartość zmiennej, w przeciwieństwie do jej typu, jak sama nazwa wskazuje, może zmieniać się w czasie.

Jeżeli zmienna (lokalna) zostanie zadeklarowana, ale nie będzie jeszcze zainicjowana, czyli nie będzie podana jej wartość, i spróbujemy z niej skorzystać, np. próbując wyświetlić jej wartość, to otrzymamy błąd kompilacji.



```
1 // deklaracja  
2 int nazwaZmiennej1;  
3  
4 // inicjalizacja  
5 nazwaZmiennej1 = 2;  
6  
7 // deklaracja z inicjalizacją  
8 int nazwaZmiennej2 = 1;
```



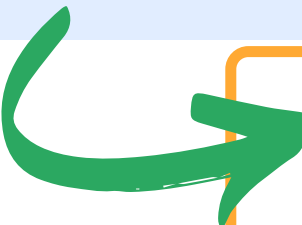
STAŁE

W Javie stałe deklaruje się bardzo podobnie do zmiennych, z tą różnicą, że przed deklarowanym typem trzeba dodać jeszcze słowo kluczowe: **final**.


Zadeklarowanej w ten sposób stałej można przypisać wartość tylko raz. Późniejsze próby jej modyfikacji zakończą się błędem kompilacji. W Javie stałe są wykorzystywane dość powszechnie, np. w klasach opakowujących (Integer, Long itp.).

Dobrym zwyczajem jest przechowywanie różnego rodzaju niezmiennych wartości właśnie jako stałych, dzięki czemu unikniemy ich przypadkowej modyfikacji oraz, jeżeli zaszłaby potrzeba zmiany tej wartości w całym systemie, możemy to zrobić, modyfikując kod tylko w jednym miejscu.

<https://stormit.pl/stale/>



```
1 final double CONSTANT_PI = 3.14;  
2  
3 final int OTHER_CONSTANT;  
4 OTHER_CONSTANT = 1;
```



STAŁE C.D.

KONWENCJA NAZEWNICTWA

Nie ma takiego wymogu składniowego, jednak według konwencji nazewnictwa w Javie w przypadku stałych powinno stosować się SCREAMING SNAKE CASE tzn. nazwa powinna być wielkimi literami i cyframi, a kolejne słowa powinny być oddzielone znakiem podkreślenia.

```
long maxValue = Long.MAX_VALUE;  
long minValue = Long.MIN_VALUE;
```

PRZYKŁAD



```
public final class Long extends Number implements Comparable<Long> {  
    /**  
     * A constant holding the minimum value a {@code long} can  
     * have, -263.  
     */  
    @Native public static final long MIN_VALUE = 0x8000000000000000L;  
  
    /**  
     * A constant holding the maximum value a {@code long} can  
     * have, 263-1.  
     */  
    @Native public static final long MAX_VALUE = 0x7fffffffffffffffL;  
}
```



TYPY W JAVIE

W Javie typy danych możemy podzielić na dwie główne kategorie:
typy proste oraz typy obiektowe.

TYPY PROSTE

Typy proste, inaczej mówiąc **prymitywne**, przechowują tylko „surowe” dane, takie jak:

- **liczby całkowite**, np. 1, 10, 200;
- **liczby zmiennoprzecinkowe**, np. 1.0, 10.5, 200.1;
- **znaki**, np. 'a', 'z', 'l';
- **typ logiczny** – czyli prawdę (true) lub fałsz (false);

<https://stormit.pl/typy-proste/>

LICZBY CAŁKOWITE

Liczby całkowite, czyli **stałoprzecinkowe**, dodatkowo podzielone są na 4 rodzaje ze względu na rozmiar liczby, jaki mogą przechowywać:

- **byte** – 1 bajt – zakres od -128 do 127;
- **short** – 2 bajty – zakres od -32 768 do 32 767;
- **int** – 4 bajty
– zakres od -2 147 483 648 do 2 147 483 647;
- **long** – 8 bajtów – zakres od -2⁶³ do 2⁶³-1;

LICZBY CAŁKOWITE – WYBÓR TYPU

W dzisiejszych czasach programista zazwyczaj ma do dyspozycji bardzo dużo taniej pamięci, dlatego głównie wykorzystywany jest uniwersalny typ: *int* nawet do przechowania stosunkowo niewielkich liczb.

LICZBY CAŁKOWITE – NA CO TRZEBA UWAŻAĆ?

W Javie, w przeciwieństwie np. do C/C++, nie ma osobnego typu dla liczb bez znaku (Unsigned), dlatego trzeba szczególnie uważać, żeby nie przekroczyć zakresu danych zmiennej, bo możemy otrzymać np. liczbę ujemną.

TYPY PROSTE C.D.



LICZBY ZMIENNOPRZECINKOWE

Liczby zmiennoprzecinkowe, czyli takie, w których możemy przechować np. ułamki, zostały dodatkowo podzielone na dwa typy:

- **float** – 4 bajty,
- **double** – 8 bajtów.

W liczbach zmiennoprzecinkowych część całkowita od ułamkowej oddzielona jest kropką.

```
1 float result = 1 / 3f;  
2 System.out.println(result); //0.33333334
```



LICZBY ZMIENNOPRZECINKOWE – ZAOKRĄGLENIA

Taki rodzaj przechowywania zmiennych niesie ze sobą pewne ograniczenia, a właściwie **zaokrąglenia** np. przechowując wynik dzielenia 1 przez 3, dostajemy ułamek nieskończony 0.(3), który w przypadku typu *float* zostanie zaokrąglony do 0.33333334.

Jeżeli nie możemy pozwolić sobie na takie zaokrąglenia, należy skorzystać np. z klasy pomocniczej **BigDecimal** do przechowywania zmiennych, która między innymi pozwala, na precyzyjne określenie, jak ma zachowywać się wynik obliczeń w przypadku zaokrągleń.

TYP ZNAKOWY CHAR

Java udostępnia również typ znakowy: *char*, który wykorzystywany jest do przechowywania kodu znaku z systemu kodowania *Unicode*, pozwalającego na przedstawienie znaków niemal ze wszystkich języków.

W pamięci komputera nie jest przechowywany tak naprawdę sam znak, np. litera 'a', a jego kod, w tym wypadku będzie to: liczba 97.

Dlatego jest to specyficzny typ liczbowy, przechowujący nieujemne całkowite liczby z zakresu: od 0 do 65535.

TYP LOGICZNY BOOLEAN

Typ logiczny *boolean* przeznaczony jest do przechowywania tylko dwóch wartości:

- **true** – prawda
- **false** – fałsz

PRZYKŁAD

```
1 // = 97  
2 System.out.println((int) 'a');  
3  
4 // = a; dlatego, że: (16^0 * 1) + (16^1 * 6) = 1+96 = 97  
5 System.out.println('\u0061');  
6  
7 // = a  
8 System.out.println((char) 97);
```



ALGORYTMY I STRUKTURY DANYCH

CO TO JEST ALGORYTM?

Algorytm możemy rozumieć jako jednoznaczny przepis postępowania, gdzie zamieniamy jakieś informacje wejściowe na oczekiwany wynik.

STRUKTURY DANYCH

Strukturę danych można postrzegać jako swego rodzaju pojemnik na dane, który gromadzi informacje i układa je w odpowiedni sposób.

REKORD – STATYCZNA STRUKTURA DANYCH

Rekord jest jedną z prostszych struktur danych, która świetnie nadaje się do prezentowania informacji o jednym obiekcie. Ponieważ w trakcie działania algorytmu nie zmienia on swego rozmiaru ani struktury, mówimy, że jest to statyczna struktura danych. **Rekord bardzo dobrze sprawdza się, gdy chcemy przechować dane na temat tylko jednego obiektu.**

TABLICE (ARRAY)

Tablice bardzo **dobrze sprawdzają się w sytuacji, gdy mamy pewien zbiór danych tego samego typu, który nie będzie modyfikowany podczas działania algorytmu.** Rozmiar tego typu struktur musimy jednak określić już na samym początku, podczas podawania wstępnych parametrów.

Inną charakterystyczną cechą tablicy, poza jej stałym rozmiarem, jest umiejscowienie kolejnych jej elementów obok siebie w pamięci, dzięki czemu **mamy możliwość prostego i szybkiego dostępu do nich przy pomocy indeksów.**

Struktury danych – Tablica

```
1 City[] cities = new City[100];  
2 cities[0] = new City("Gdańsk", "Pomorskie", 500000);
```

<https://stormit.pl/tablice/>

PRZYKŁAD

Struktury danych – rekord

```
1 {  
2     name:      "Gdańsk",  
3     province:  "Pomorskie",  
4     population: 500000  
5 }
```

W Javie możemy w tym celu wykorzystać klasę:

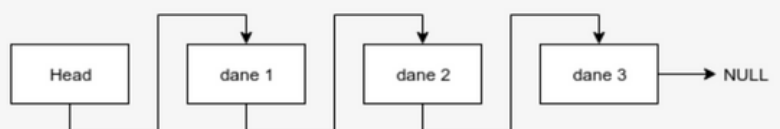
Struktury danych – Klasa

```
1 public class City {  
2  
3     private String name;  
4     private String province;  
5     private long population;  
6  
7     public City(String name, String province, long population)  
8     {  
9         this.name = name;  
10        this.province = province;  
11        this.population = population;  
12    }  
13  
14    City gdansk = new City("Gdańsk", "Pomorskie", 500000);
```

Indeks	0	1	2	3	4
Wartość	v1	v2	v3	v4	v5

LISTY (LIST)

Lista, podobnie jak tablica, przeznaczona jest do przechowywania większej ilości danych. Z tym wyjątkiem, że lista umożliwia łatwe dodawanie nowych elementów. Listę można postrzegać jako łańcuszek połączonych ze sobą danych.



Struktury danych lista

LISTY (LIST) C.D.

Istnieją dwie popularne realizacje struktury listy: **tablicowa** oraz **wskaźnikowa**.

LISTA TABLICOWA – ARRAYLIST

Do implementacji tej listy została wykorzystana tablica. Jest to tak naprawdę zwykła tablica wyposażona w dodatkowe funkcje, które ukrywają złożoną logikę. Największą zaletą listy tablicowej stanowi jej prosta nawigacja z wykorzystaniem indeksu.

Lista tablicowa

```
1 List list = new ArrayList();
2 list.add("A");
3 list.add("B");
4 String b = (String) list.get(1);
```

LISTA WSKAŹNIKOWA – LINKEDLIST

Implementacja wskaźnikowa wymaga dodania do każdego elementu listy nowej wartości: wskaźnika, czyli referencji, dzięki czemu zrealizowana jest lokalna nawigacja w liście. W przeciwieństwie do tablicy logiczna kolejność elementów w liście wskaźnikowej może być inna niż jej fizyczne ułożenie w pamięci.

Lista wskaźnikowa

```
1 List list = new java.util.LinkedList();
2 list.add("A");
3 list.add("B");
4 String b = (String) list.get(1);
```

STOS (STACK) – LIFO

Stos nazywany jest również kolejką LIFO, czyli „ostatni na wejściu, pierwszy na wyjściu” (z ang. *Last In, First Out*).

Bardzo dobrze sprawdza się, gdy chcemy przeprowadzać operacje najpierw na najnowszych danych, a dopiero później – na starszych.

Podstawowe operacje na stosie to:

- **push (obiekt)** – dodanie obiektu na wierzch stosu;
- **pop ()** – ściągnięcie jednego elementu ze stosu i zwrócenie jego wartości.

<https://stormit.pl/stos-lifo/>

Struktury danych – Stos

```
1 Deque<String> stack = new java.util.LinkedList();
2 stack.push("A");
3 stack.push("B");
4
5 stack.pop();
6 String a = stack.pop();
```

UWAGA

W zależności od ilości wskaźników oraz sposobu ich wykorzystania w liście możemy wyróżnić kilka szczególnych przypadków jej implementacji:

- **Lista jednokierunkowa**
- **Lista dwukierunkowa**
- **Lista cykliczna**



KOLEJKA (QUEUE) – FIFO

Przeciwieństwem stosu LIFO jest kolejka typu FIFO (ang. *First In, First Out*), czyli „pierwszy na wejściu, pierwszy na wyjściu”. W tym przypadku zaczynamy przetwarzanie danych od elementów, które pojawiły się jako pierwsze. Jest to odpowiednik zwykłej kolejki w sklepie.

Szczególną odmianą tej struktury danych jest kolejka priorytetowa. W tej implementacji konkretne rekordy dodatkowo posiadają przypisany priorytet, który wpływa na kolejność późniejszego pobierania elementów z listy.

<https://stormit.pl/kolejka-fifo/>

Struktury danych – Kolejka

```
1 java.util.Queue<String> queue = new java.util.LinkedList();
2 queue.add("A");
3 queue.add("B");
4
5 String a = queue.poll();
```

ZBIÓR (SET)

Zbiór to kolekcja elementów, która nie może zawierać duplikatów. W Javie najczęściej wykorzystywane są dwie implementacje: *HashSet* oraz *TreeSet*.

Główna różnica między tymi dwiema implementacjami polega na tym, że ***TreeSet* wyznacza kolejność elementów (domyślnie jest ona rosnąca), a *HashSet* – nie.**

W szczególnych przypadkach może się jednak zdarzyć, że *HashSet* również posortuje elementy rosnąco, jednak implementacja tego nie gwarantuje.

Struktury danych – Zbiór

```
1 Set<String> hashSet = new HashSet<>();
2 hashSet.add("c1");
3 hashSet.add("a1");
4 hashSet.add("b1");
5 hashSet.add("b1");
6
7 Set<String> treeSet = new TreeSet<>();
8 treeSet.add("c1");
9 treeSet.add("a1");
10 treeSet.add("b1");
11 treeSet.add("b1");
12
13 System.out.println(hashSet); // [a1, c1, b1]
14 System.out.println(treeSet); // [a1, b1, c1]
```

DYNAMICZNE STRUKTURY DANYCH

Dynamiczne struktury danych w przeciwieństwie do statystycznych (takich jak np. rekord) pozwalają na zmianę ich struktury już podczas działania algorytmu.

Przykłady struktur dynamicznych to np. stos, kolejka, lista jedno i dwukierunkowa, drzewo itp.



Storm IT

MAPA (MAP)

Mapa to specyficzna kolekcja, która przechowuje pary danych składające się z klucza oraz przypisanej wartości.

Klucz w obrębie mapy musi być unikatowy, natomiast wartości mogą się powtarzać.

Struktury danych – Mapa

```
1 Map<String, Integer> map = new HashMap<>();
2 map.put("Gdańsk", 10);
3 map.put("Warszawa", 5);
4 map.put("Warszawa", 7);
5
6 System.out.println(map); // {Gdańsk=10, Warszawa=7}
```

OPERATORY RELACYJNE

Wynik zawsze boolean (true lub false)

- **Równy**
`boolean result = v1 == v2;`
- **Nie równy**
`boolean result = v1 != v2;`
- **Większy niż**
`boolean result = v1 > v2;`
- **Mniejszy niż**
`boolean result = v1 < v2;`
- **Większy niż lub równy**
`boolean result = v1 >= v2;`
- **Mniejszy niż lub równy**
`boolean result = v1 <= v2;`

<https://stormit.pl/operator-relacyjne/>

OPERATORY

Dostępne różnego rodzaju operatory pozwalają na manipulację wartościami zmiennych.

OPERATORY RELACYJNE

- **I**
`boolean result = v1 && v2;`
- **Lub**
`boolean result = v1 || v2;`
- **Nie**
`boolean result = !v1;`

OPERATORY

ARYTMETYCZNE

Wynik wyrażenia zależy od wykorzystywanych typów

- **Dodawanie**
`int result = v1 + v2;`
- **Odejmowanie**
`int result = v1 - v2;`
- **Mnożenie**
`int result = v1 * v2;`
- **Dzielenie**
`int result = v1 / v2;`
- **Modulo**
`int result = v1 % v2;`

<https://stormit.pl/operator-matematyczne/>

ZŁOŻONE OPERATORY PRZYPISANIA

Służą one przede wszystkim skróceniu zapisu. Wszystkie operacje można zapisać bez ich wykorzystania, posługując się jedynie zwykłym operatorem przypisania oraz operatorami arytmetycznymi np. `x += y` to `x = x + y`.

OPERATORY PRZYPISANIA

Operator przypisania to „równa się” =

Operator ten przypisuje zmiennej po lewej stronie wartość wyrażenia po prawej stronie.

Operator jednocześnie:

- powoduje przypisanie,
- zwraca wartość równą przypisanej wartości.

1. Jeżeli zmienna jest **typu prostego**, to jej wartość jest zwyczajnie kopiowana.
2. Jeżeli zmienna jest **typu obiektowego**, czyli jest referencją, to kopiowana jest sama referencja, a nie obiekt, na który wskazuje.
3. Zmienne typu obiektowego dodatkowo potrafią przechowywać **wartość pustą**, czyli `null`.

```
int x = 6;  
x = 5;  
int result = x;  
String str = "Hello, World!";
```

<https://stormit.pl/operator-przypisania/>

PĘTLE

Pętle są jednym z podstawowych narzędzi wykorzystywanych przez programistę.

Dzięki nim można wywołać określoną funkcjonalność podaną ilość razy, zamiast za każdym razem wywoływać ją ręcznie np. zamiast wypisywać kolejno dni tygodnia w 7 instrukcjach, możemy wykorzystać do tego pętlę. W Javie mamy do dyspozycji kilka rodzajów pętli: `while`, `do while`, `for` i `foreach`. Funkcjonalność wszystkich jest wymienna, a decyzja, którą należy użyć w danym momencie, zależy głównie od kontekstu, w jakim ma być wykorzystana oraz od preferencji programisty.

<https://stormit.pl/petle/>

PĘTLE C.D.

WHILE

Pętla *while* najczęściej jest wykorzystywana, gdy nie możemy określić konkretnej ilości powtórzeń do wykonania. Znamy jednak warunki określające, jak długo dana sytuacja będzie zachodziła. Przykładowo: „dopóki licznik nie przekroczy zakresu tablicy, wypisz jej kolejny element”. Pętla jest wykonywana tak długo, póki warunek jest spełniony. Jeżeli warunek od początku nie będzie spełniony, taka pętla nie zostanie wywołana ani razu.

PRZYKŁAD

```
1 String[] days = {"Poniedziałek", "Wtorek",  
2 "Środa", "Czwartek", "Piątek",  
3 "Sobota", "Niedziela"};  
4  
5 int index = 0;  
6 while (index < days.length) {  
7     System.out.println(days[index]);  
8     index++;  
9 }
```

NIESKOŃCZONA PĘTLA

Warunek kończący pętlę jest bardzo istotny – błędy w postaci warunku, który nigdy nie zakończy pętli, mogą spowodować poważne konsekwencje tj. zawieszenie programu (program nie będzie mógł przejść dalej).

```
1 // 1  
2 while (true) {  
3 }  
4  
5 // 2  
6 for (;;) {  
7 }  
8  
9 // 3  
10 int i = 0;  
11 while (i < 10) {  
12 }
```

DO WHILE

Pętla *do while* jest to modyfikacja pętli *while*.

Główna różnica polega na tym, że ciało pętli zostanie wykonane przynajmniej raz, nawet jeżeli warunek zawsze jest niespełniony. Najpierw wykonywane są instrukcje zdefiniowane wewnątrz pętli, a dopiero potem sprawdzany jest jej warunek.

FOR

Pętla *for* zazwyczaj wykorzystywana jest, gdy z góry znamy ilość wykonania kolejnych iteracji pętli.

`for([inicjalizacja]; [warunek]; [modyfikacja]) {
 [ciało pętli - instrukcje do wykonania] }`

Wszystkie cztery elementy pętli są opcjonalne, jednak dla wygody i czytelności kodu warto z nich korzystać.

1. **inicjalizacja** – służy do zainicjowania zmiennych początkowymi wartościami, np. utworzenie i zainicjowanie licznika;
2. **warunek** – w warunku należy sprawdzić, czy ciało pętli ma być wykonane;
3. **modyfikacja** – ten element pętli jest wykonywany po ciele funkcji, można tu np. zmodyfikować licznik pętli;
4. **ciało pętli** – czyli wszystkie instrukcje, które chcemy, żeby były wykonane określoną ilość razy;

```
1 int index = 0;  
2 do {  
3     System.out.println(days[index]);  
4     index++;  
5 } while (index < days.length);
```

PRZYKŁAD

```
1 // 1  
2 for (int index = 0; index < days.length; index++) {  
3     System.out.println(days[index]);  
4 }  
5  
6 // 2  
7 int index = 0;  
8 for (; ; ) {  
9     if (index < days.length) {  
10        System.out.println(days[index]);  
11        index++;  
12    } else {  
13        break;  
14    }  
15 }
```



FOR EACH

Pętla *for each* jest modyfikacją pętli *for*. Za jej pomocą w bardzo prosty i czytelny sposób można przeglądać wszystkie elementy różnych zbiorów, np. tablic.

Pętlę można przeczytać jako: „dla każdego elementu tablicy zrób coś...”.

```
1 for (String day : days) {  
2     System.out.println(day);  
3 }
```

```
1 for (int i = 1; i <= 10; i++) {  
2     if (i % 2 == 0) {  
3         continue;  
4     }  
5     if (i >= 5) {  
6         break;  
7     }  
8     System.out.println(i);  
9 }
```

CONTINUE & BREAK

Korzystając z pętli, czasami zachodzi potrzeba ominięcia wykonywania danej iteracji lub przerwanie całej pętli. W takim przypadku z pomocą przychodzą nam dwie instrukcje:

- **continue** – kończy wykonywanie aktualnej iteracji pętli. Jeżeli warunek w pętli będzie dalej spełniony, aplikacja przejdzie do wykonywania dalszych iteracji.
- **break** – po wykonaniu instrukcji `break` kończona jest wykonywanie aktualnej iteracji oraz całej pętli. Niezależnie od wartości warunku pętli kolejna iteracja nie będzie już wykonana – warunek pętli nie będzie w tym wypadku już nawet sprawdzany.

INSTRUKCJE WARUNKOWE

IF ELSE

Instrukcje warunkowe IF ELSE to konstrukcja języka, dzięki której można rozwidlić ścieżkę wykonywania programu.

Przy jej pomocy możemy określić warunki, jakie mają zajść, żeby dany fragment kodu został wykonany.

W wolnym tłumaczeniu można przeczytać ją jako: jeżeli zajdzie pewien warunek, to zrób 'to', w przeciwnym wypadku zrób 'tamto'.

IF

Instrukcja *if* to podstawowa i najprostsza instrukcja warunkowa. Jej działanie polega na wykonaniu kodu z jej ciała, jeżeli warunek został spełniony lub ich pominięciu w przeciwnym wypadku.

<https://stormit.pl/if-else/>

```
1 int number = 10;  
2  
3 if (number > 0) {  
4     System.out.println("+");  
5 }  
6 else if (number < 0) {  
7     System.out.println("-");  
8 }  
9 else if (number == 0) {  
10    System.out.println("0");  
11 }
```

ELSE

Rozwinięciem podstawowej instrukcji *if* jest instrukcja *if else*.

Dzięki niej można określić szereg opcjonalnych warunków oraz określić, co się wtedy ma wydarzyć.

```
1 if (10 > 0) {  
2     System.out.println("+");  
3 }
```

```
1 if (number == 0) {  
2     System.out.println("0");  
3 } else {  
4     System.out.println("");  
5 }
```

ELSE IF

Kolejne warunki w instrukcji *if else* są sprawdzane po kolei aż do momentu, gdy któryś z nich nie zwróci prawdy (`true`).

Wykonywany jest wtedy przypisany mu fragment kodu.

Pozostałe warunki, nawet jeżeli byłyby prawdziwe, nie będą nawet sprawdzane.

SWITCH CASE

To instrukcja wielokrotnego wyboru, dzięki której można warunkowo wykonać pewne fragmenty kodu. Jest to swego rodzaju rozszerzenie instrukcji IF ELSE. Jednak od pierwowzoru różni się przede wszystkim typem przyjmowanych argumentów wejściowych, możliwością wykonania kilku bloków kodu i samą czytelnością zapisu, ale o tym wszystkim za chwilę.

<https://stormit.pl/switch-case/>

SWITCH CASE C.D.

Switch nie jest typową instrukcją warunkową, a raczej swego rodzaju przełącznikiem.

Przy jego pomocy można „przełączyć” wykonywanie kodu aplikacji na różne tory w zależności od zaistniałej sytuacji.

WARIANTY CASE

W instrukcji *Switch*, w przeciwieństwie do *If Else*, nie określamy warunków, które muszą być spełnione, aby dany kod został wykonany. W przypadku tej instrukcji określamy wyrażenie wejściowe oraz warianty kodu, które mają być wykonane w zależności od jego wartości. Poszczególne warianty oznaczamy słowem kluczowym *case*, po którym następuje wartość obsługiwanego argumentu wejściowego i fragment przypisanego kodu do wykonania.

```
1  int number = 1;
2  switch (number) {
3
4
5  switch (number) {
6      case 1:
7          System.out.println("jeden");
8      case 2:
9          System.out.println("dwa");
10 }
```

BREAK

Poszczególne warianty *case* nie muszą być całkowicie niezależnymi fragmentami kodu i wcale nie muszą się wykluczać. Powinny być raczej rozumiane jako początkowe punkty wejścia do instrukcji *switch*.

Jeżeli aplikacja wejdzie do jakiegoś warunku *case*, to będzie wykonywała kaskadowo wszystkie kolejne bloki kodu. Wykonywanie kolejnych bloków kończy się dopiero instrukcją *break* lub końcem całej instrukcji *switch*.

DEFAULT

Instrukcja *Switch* daje również możliwość określenia opcjonalnego wariantu domyślnego, który zostanie wywołany, jeżeli argument wejściowy nie zostanie dopasowany do żadnego warunku *case*. Wariant domyślny określamy słowem kluczowym *default*. Zazwyczaj taki warunek umieszcza się na samym końcu instrukcji *switch*, jednak nie jest to konieczne.

SWITCH VS IF ELSE

Nie ma jasno zdefiniowanych reguł o tym decydujących. Te dwie instrukcje są praktycznie wymienne i to czytelność kodu powinna zadecydować, z której z nich w danym momencie skorzystać. Zazwyczaj dla prostych porównań lepiej sprawdza się instrukcja *if*, a dla bardziej rozbudowanych *switch*. *Switch* bardzo dobrze sprawdza się też w przypadku zmiennych typu *enum*.

```
1  switch (number) {
2      case 1:
3          System.out.println("jeden");
4
5      case 2:
6          System.out.println("dwa");
7          break;
8
9      default: {
10         System.out.println("default");
11     }
12 }
```

MODYFIKATORY DOSTĘPU

Służą do ustawiania poziomu dostępu do klas, zmiennych, metod i konstruktorów.

- **Prywatny** (ang. *private*) – element dostępny tylko z poziomu tej samej klasy.
- **Publiczny** (ang. *public*) – wszystkie inne klasy we wszystkich pakietach będą mogły korzystać z takiego elementu – najmniej restrykcyjny modyfikator dostępu.
- **Domyślny** (ang. *default*) – *package private* – element jest dostępny tylko dla klas z tego samego pakietu.
- **Chroniony** (ang. *protected*) – możemy uzyskać dostęp do tego elementu z tego samego pakietu (jak w przypadku modyfikatora domyślnego) oraz dodatkowo ze wszystkich podklas, nawet jeśli znajdują się one w innych pakietach.

KONWERSJA TYPÓW

ZAWĘŻENIE

Zawężanie (ang. *narrowing*) odbywa się za pomocą rzutowania typu

– wymuszona jawnie konwersja. Korzystaj z niej w momencie kiedy:

- chcesz przypisać wartość większego typu danych do mniejszego typu
- istnieje szansa na utratę precyzji wyniku podczas wykonania wyrażenia, której nie chcemy
- chcemy zamienić wartość liczbową na *char*, lub odwrotnie

KONWERSJA TYPÓW

Zmiana jednego typu prymitywnego w inny, poprzez:

- poszerzanie
- zawężanie

<https://stormit.pl/konwersja-rzutowanie-typow/>

POSZERZANIE

Poszerzanie (ang. *widening*) odbywa się za pomocą konwersji typu – automatycznego przekształcenia jednego typu w drugi może nastąpić – tylko z typu węższego, do szerszego, czyli np. z *int* do *long*

```
1 int value1 = 1;
2 int value2 = 2;
3 int value3 = value2 / value1; // = 2
4 double value4 = value2 / value1; // = 2.0
```

```
1 int value1 = 10;
2 int value2 = 3;
3
4 double value3 = value2 / value1;
5 double value4 = value2 / (double)value1;
6 int value5 = (int) 1.7;
7 char value6 = (char)65;
```

TESTOWANIE

Testowanie oprogramowania pozwala ocenić jego jakość i zmniejszyć ryzyko wystąpienia awarii.

W podstawowej wersji polega na uruchomieniu naszej aplikacji lub jej fragmentu i sprawdzeniu uzyskanych rezultatów.

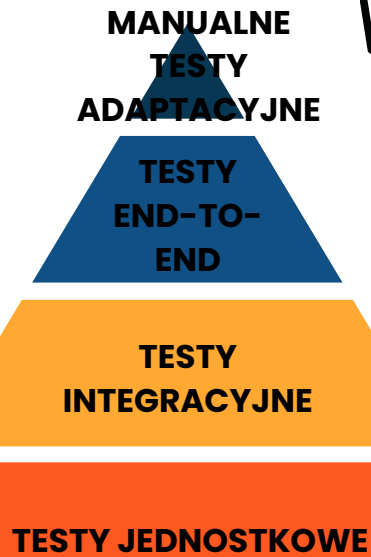
<https://stormit.pl/testowanie-oprogramowania/>

TESTOWANIE – KORZYŚCI

Co dają nam testy?

- Pomagają poprawić stabilność i jakość naszego rozwiązania
- Pozwalają również bezpiecznie wprowadzać zmiany w naszym kodzie
- Wymuszają myślenie o tym, jak kod będzie wykorzystywany
- Większa przewidywalność
- Pozwalają w łatwy sposób uczyć się nowych technologii i bibliotek
- Umożliwiają zweryfikować założenia oraz uniknięcia pomyłek swoich i innych

TESTOWANIE C.D.



ZDROWA PIRAMIDA TESTÓW (POZIOMY TESTÓW)

- **Manualne testy akceptacyjne** – ich celem jest nabranie zaufania do testowanego systemu, jego części lub tylko pewnych atrybutów нефункциональных – np. wydajności aplikacji.
- **Testy end-to-end/systemowe** – symulują zachowanie konkretnego użytkownika korzystającego z aplikacji.
- **Testy integracyjne** – weryfikują, czy kolejne komponenty aplikacji odpowiednio ze sobą współpracują – Testujemy kilka modułów systemu jednocześnie.
- **Testy jednostkowe** – przeprowadzane są na bardzo niskim poziomie, testowane są poszczególne klasy lub nawet same metody (testują jeden wybrany fragment logiki).

<https://stormit.pl/testy-jednostkowe-junit/>

PROGRAMOWANIE OBIEKTOWE

PROGRAMOWANIE OBIEKTOWE (ANG. OBJECT ORIENTED PROGRAMMING)

Popularny paradygmat programowania, w którym modelujemy istniejącą rzeczywistość za pomocą obiektów, zamiast stosować tylko funkcje i logikę. Projektowanie obiektowe polega na umiejętności przedstawienia sytuacji z życia codziennego pod postacią obiektów i relacji między nimi.

Obiekty to elementy łączące stan – czyli przechowywane dane, nazywane często polami oraz zachowanie – czyli procedury/metody.

Ogromną zaletą programowania obiektowego jest jego zgodność ze światem rzeczywistym.

<https://stormit.pl/programowanie-obiektowe/>



PROGRAMOWANIE OBIEKTOWE – PARADYGMATY (ZASADY)

Główne paradygmaty programowania obiektowego:

- *Abstrakcja*
- *Enkapsulacja*
- *Dziedziczenie*
- *Polimorfizm*

PROGRAMOWANIE OBIEKTOWE C.D.

DZIEDZICZENIE

Java pozwala na dziedziczenie metod oraz atrybutów z jednej klasy do drugiej np. Samochód dziedziczy po Pojazd.

- **Podklasa / klasa pochodna (dziecko)** – klasa, która dziedziczy od innej
- **Nadklasa / klasa bazowa / superklasa (rodzic)** – klasa po której dziedziczymy

Można dziedziczyć tylko po jednej klasie – Java nie wspiera wielodziedziczenia (Wyjątek stanowią metody domyślne w interfejsach).

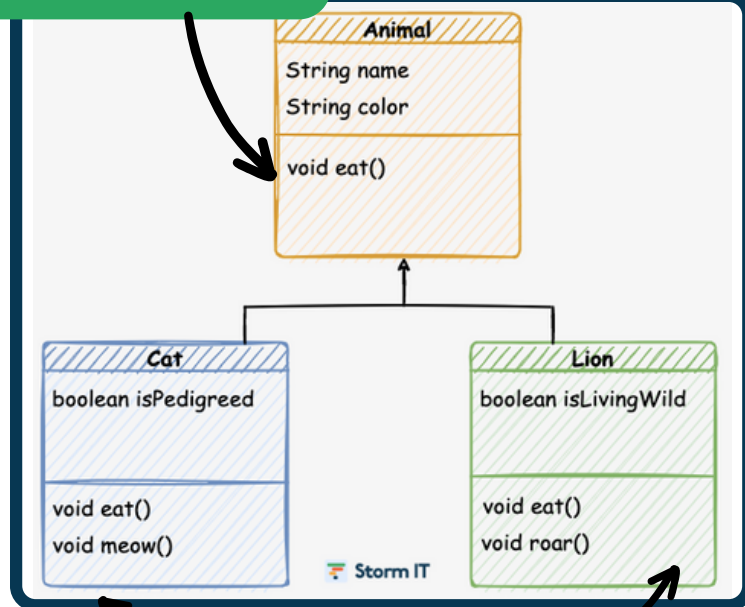
ZALETY

- Przejmujemy dane i zachowanie klasy bazowej
- Możliwość powtórnego wykorzystania kodu
- Lepsze odzwierciedlenie rzeczywistości

DZIEDZICZENIE METOD

- Musi być zgodny kontrakt: Ta sama nazwa oraz zwracany typ (akceptowany jest również zgodny podtyp)
- Opcjonalnie można użyć adnotacji `@Override` (Pomaga to w czytelności kodu, jednak nie jest niezbędne)

NADKLASA



PODKLASA

ABSTRAKCJA

Abstrakcja jest procesem, który dostarcza różne funkcjonalności użytkownikowi poprzez ukrywanie przed nimi szczegółów jej implementacji.

```
1 abstract class Animal {  
2     abstract void makeSound();  
3 }  
4  
5 class Cat extends Animal {  
6  
7     @Override  
8     public void makeSound() {  
9         System.out.println("Miauuuu");  
10    }  
11 }
```

KLASA ABSTRAKCYJNA

Klasa, której nie można utworzyć instancji – Można po niej dziedziczyć i dzięki temu dostarczyć metod zdefiniowanych i/lub abstrakcyjnych. Deklaruję się ją za pomocą słowa kluczowego **abstract** a można ją dziedziczyć poprzez użycie **extends**.

- Może, ale nie musi zawierać metody abstrakcyjnej (metody bez ciała) – muszą one zostać zdefiniowane w klasach potomnych.
- Nie może być finalna (tak samo, jak metoda abstrakcyjna)
- Możliwe jest dodanie w klasie abstrakcyjnej konstruktora bez lub parametryzowanego – jeżeli go nie utworzysz, klasa zawsze będzie miała konstruktor domyślny bezparametrowy.

INTERFEJS

„Całkowicie abstrakcyjna klasa”, która zawiera deklarację metod, ale nie zawiera ich implementacji. Do deklaracji interfejsu używa się słowa kluczowego *interface*, a do implementacji *implements*. Klasy, które implementują dany interfejs, muszą zaimplementować wszystkie metody zawarte w nim. Klasa może implementować wiele interfejsów jednocześnie. Interfejs gwarantuje, że jakaś metoda istnieje, a nie jak się konkretnie zachowuje. Na podstawie interfejsów nie można tworzyć nowych obiektów.

```
1 interface Programmer {
2     String code();
3
4     String sleep();
5 }
6
7 class JavaProgrammer implements Programmer {
8     @Override
9     public String code() {
10         return "Java code";
11     }
12
13     @Override
14     public String sleep() {
15         return "Sleeping about Java...";
16     }
17 }
```

POLIMORFIZM

Zdolność do przyjmowania wielu form. Mechanizm ten umożliwia wykonywanie różnych operacji na różnych obiektach przy użyciu tego samego kodu. Jest bardzo mocno związany z dziedziczeniem, klasami abstrakcyjnymi oraz interfejsami. Polimorfizm pomaga w ponownym wykorzystaniu kodu.

POLIMORFIZM CZASU KOMPILACJI

Polimorfizm statyczny uzyskiwany poprzez przeciążenie metod (ang. *Method Overloading*). Kiedy mamy w klasie np. 2 metody o tej samej nazwie, ale z różnymi argumentami. Już podczas kompilacji na podstawie przekazywanych argumentów można wywnioskować, która z metod powinna zostać wywołana.

POLIMORFIZM W CZASIE WYKONYWANIA

Polimorfizm dynamiczny uzyskiwany poprzez nadpisywanie metod (ang. *Method Overriding*). Kiedy np. w 2 klasach, w których mamy analogiczną metodę o takiej samej nazwie i o takim samym zestawie parametrów. To kontekst, czyli obiekt decyduje o tym, która z metod zostanie wywołana.

```
1 String getInfo() {
2     return "I am person";
3 }
4 String getInfo(String name) {
5     return "I am person. My name is: " + name;
6 }
```



ENKAPSULACJA

Mechanizm polegający na ukrywaniu szczegółów implementacyjnych klasy lub obiektu za pomocą abstrakcji. Pozwala to na zapewnienie integralności danych przechowywanych przez obiekt poprzez wiązanie danych, wspólnie z metodami operującymi na tych danych, w jedną całość. Enkapsulacja umożliwia również łatwą zmianę implementacji obiektu bez konieczności modyfikacji kodu, który z nim współpracuje. Przykładem jest klasa, ponieważ składa się z danych i metod, które zostały połączone w jedną całość.

HERMETYZACJA

Hermetyzacja polega na ukryciu informacji (szczegółów implementacji), które nie powinny być widoczne poza klasą lub modulem. Celem jest ograniczanie bezpośredniego dostępu do poszczególnych komponentów naszej implementacji, czyli między innymi na ukrywaniu metod i atrybutów dla klas zewnętrznych. Hermetyzacja może być używana do ukrywania zarówno członków danych, jak i metod związanych z instancją klasy lub obiektu.

Aby osiągnąć hermetyzację, należy:

- Zadeklarować zmienne lub metody klasy jako prywatne
- Zapewnić publiczne metody get i set, aby uzyskać dostęp i aktualizować wartość prywatnej zmiennej

WRAPPERY

Klasy opakujące typy proste w typy obiektowe – pozwalają by nasz typ prosty, zachowywał się przynajmniej częściowo jak typ obiektowy, np. *Integer* dla typu *int*.

AUTOBOXING

Zapakowanie typów prostych

UNBOXING

Rozpakowanie do typów prostych – odwrotna konwersja do wcześniejszego przypadku, następuje tu „rozpakowanie” typu obiektowego do typu prymitywnego.

AUTOBOXING I UNBOXING

Mechanizmy, które pozwalają:

- Płynnie przechodzić między typami prostymi a obiektowymi np. przypisując `Integer v = 10;`
- Wykorzystywać typy proste, tam, gdzie wymagane są obiekty np. `new ArrayList().add(1);`

<https://stormit.pl/klasy-oslonowe/>

KLASA OBJECT

KLASA OBJECT

Główny element w hierarchii klas i domyślnie klasa nadrzędna wszystkich klas w Javie – Każda klasa w Javie dziedziczy po klasie *Object*. Jeśli klasa nie rozszerza żadnej innej klasy, jest bezpośrednią klasą potomną klasy *Object* – Jeśli rozszerza inną klasę, jest pośrednio klasą potomną klasy *Object*.

Klasa dziedzicząca (dziecko) przejmuje zachowanie od rodzica, do którego może dodać swoje własne specyficzne cechy. Metody klasy *Object* są dostępne dla wszystkich klas Java – Niektóre z metod z tej klasy można nadpisać.

METODY

• toString()

Zapewnia tekstową reprezentację obiektu. Domyślnie zwraca ciąg znaków składający się z nazwy klasy, której obiekt jest instancją, znaku „@” oraz reprezentacji szesnastkowej *hashCode*'u.

• hashCode()

Zwraca różne liczby całkowite, dla różnych obiektów. Powszechnie wykorzystywane do optymalizacji w kolekcjach.

• equals(Object obj)

Służy, do porównania dwóch obiektów np. porównaniu 2 klientów po numerze klienta lub peselu. Możliwe jest jej nadpisanie. Jeżeli tego nie zrobimy to domyślna implementacja z klasy *Object* sprawdzi identyczność obiektów (czy są w tym samym miejscu w pamięci).

HASH

- Generowany jest na podstawie stanu obiektu – wartości pól obiektu, które wyróżniają go od innych obiektów tej samej klasy
- Jest jednoznaczny – każde wygenerowanie powinno zwracać ten sam wynik
- Ze względów wydajnościowych możliwie niewielka liczba obiektów powinno zwracać identyczny hash.

#

EQUALS()

```
1 @Override
2 public boolean equals(Object o) {
3     if (this == o) return true;
4     if (o == null || getClass() != o.getClass()) return false;
5     Person person = (Person) o;
6     return name.equals(person.name);
7 }
```

KLASA OBJECT C.D.

KONTRAKT MIĘDZY METODAMI HASHCODE() I EQUALS()

- Każdy obiekt jest różny od *null*, czyli wywołanie *x.equals(null)* dla obiektu *x* różnego od *null*, zawsze musi zwrócić *false*.
 - Relacja wyznaczona metodą *equals* musi być:
 - spójna, czyli każdorazowe wywołanie *x.equals(y)* (przy założeniu, że między wywołaniami obiekty nie były modyfikowane) zawsze musi zwracać tę samą wartość – zawsze *true* albo zawsze *false*.
 - przechodnia, czyli dla dowolnych zmiennych *z*, *y* i *z*, jeżeli *x.equals(y) = true* oraz *y.equals(z) = true*, to *x.equals(z)* musi również = *true*.
 - symetryczna, czyli dla każdej pary zmiennych *x* i *y*, wyrażenie *x.equals(y)* ma wartość *true* wtedy i tylko wtedy gdy *y.equals(x) = true*.
 - zwrotna, czyli dla każdej zmiennej *x* różnej od *null* wyrażenie *x.equals(x)* musi zwracać wartość = *true*.
 - Jeżeli obiekty są różne (wg metody *equals*), to ich *hashCode* może być równy.
 - Jeżeli dwa obiekty są sobie równe (wg metody *equals*), to ich *hashCode* również musi być równy.
- Brak zachowania kontraktu niestety nie powoduje błędów kompilacji. Może jednak przyczynić się do powstania trudnych do wyśledzenia błędów podczas działania aplikacji.

<https://stormit.pl/hashcode-equals/>

TYPY GENERYCZNE

TYPY GENERYCZNE

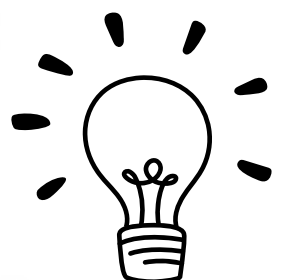
Mechanizm, który umożliwia tworzenie klas, metod lub funkcji, które mogą pracować z różnymi typami danych. Pozwalają na napisanie uniwersalnych struktur danych: klas, interfejsów, a także metod, które mogą być używane z różnymi typami danych bez konieczności tworzenia wielu wersji dla każdego z możliwych typów.

```
1 List v1 = new ArrayList();
2 List<Integer> v2 = new ArrayList();
3 List<String> v3 = new ArrayList();
4 List<Boolean> v4 = new ArrayList();
5
6 v1.add("Ala");
7 v2.add(123);
8 v3.add("Kasia");
9 v4.add(true);
```

SAMODZIELNA DEKLARACJA

Typ generyczny definiujemy poprzez podanie parametrów w nawiasach kątowych. Parametry te używa się w ciele klasy, interfejsu bądź metody w miejscu, gdzie używalibyśmy określonych „normalnych” typów.

```
1 class Box<T> {
2     private T t;
3
4     public void add(T t) {
5         this.t = t;
6     }
7
8     public T get() {
9         return t;
10    }
11 }
12
13 Box box = new Box<Integer>();
14 Box box1 = new Box<String>();
15 box.add(34);
16 box1.add("Kasia");
```



STRINGBUFFER & STRINGBUILDER

Klasy służące do reprezentowania ciągów znaków. Różnią się od klasy `String` tym, że są mutowalne – można dokonywać zmian na obiektach tej klasy bez konieczności tworzenia nowych obiektów. Klasy te posiadają wiele metod, które umożliwiają manipulowanie ciągiem znaków, takich jak dodawanie lub usuwanie znaków, zamienianie zawartości na inną, itp. `StringBuffer` i `StringBuilder` mogą być wydajniejsze niż `String`, jeśli chodzi o operacje modyfikujące zawartość ciągu znaków, ponieważ nie tworzy nowych obiektów podczas takich operacji.

<https://stormit.pl/stringbuilder/>

```
1 | StringBuilder sb = new StringBuilder();
2 | sb.append("Ala ");
3 | sb.append("ma ");
4 | sb.append("kota.");
5 | String str = sb.toString();
6 | System.out.println(str);
```

	STRINGBUFFER	STRINGBUILDER
MUTOWALNOŚĆ	✓	✓
MECHANIZMY SYNCHRONIZACJI	✓	✗
THREAD-SAFE	✓	✗

WYJĄTKI (ANG. EXCEPTIONS)

Sposób na zarządzanie błędami lub sytuacjami wyjątkowymi (nieoczekiwanymi), które mogą wystąpić w trakcie działania programu.

Są one zazwyczaj rzucane przez metody i klasy, gdy występuje sytuacja, która jest poza zakresem normalnego działania danej struktury. Podczas zgłoszenia wyjątku przez program normalny przepływ aplikacji jest przerywany. Wyjątki mogą zostać przechwycone i obsłużone przez odpowiedni kod, jeżeli nie zostaną, normalny przepływ aplikacji zostaje zakłócony, i zakończy on nieprawidłowo swoje działanie. Wyjątki są obiektami klasy `java.lang.Exception` lub jej podklas.

RODZAJE WYJĄTKÓW

- **Czasu kompilacji**, oznaczonym (ang. *checked exceptions*)

Wyjątki dziedziczące po `Exception`, ale nie dziedziczące po `RuntimeException` np. `IOException`.

Są sprawdzane przez kompilator podczas kompilacji – muszą zostać obsłużone za pomocą bloku `try-catch`, albo za pomocą słowa kluczowego `throws`.

- **Czasu wykonania**, nieoznaczone

(ang. *unchecked exceptions*)

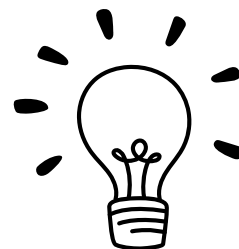
Występują podczas wykonywania programu np.

`java.lang.NullPointerException`

Możliwe jest zapewnienie obsługi wyjątków niekontrolowanych, jednak nie jest to konieczne z punktu widzenia kompilatora.

PRZYKŁAD

```
1 | try {
2 |     File file = readFile();
3 |     saveFile(file);
4 | } catch (Exception) {
5 |     // handle exception
6 | }
```



CLEAN CODE

Kod, który jest łatwy do zrozumienia i utrzymania. Jest on zazwyczaj czytelny, zwięzły i dobrze zorganizowany. Powstały różne zasady i wytyczne, które pomagają tworzyć czysty kod np. stosowanie dobrych praktyk programistycznych, takich jak unikanie duplikacji kodu, stosowanie odpowiednich komentarzy i zachowanie zasad związanych z formatowaniem kodu. Clean code pozwala na szybkie zrozumienie kodu i łatwe jego modyfikowanie lub rozszerzanie w przyszłości. W rezultacie może przyczynić się do zwiększenia wydajności i jakości tworzonego oprogramowania.

DEBUGGING

Proces systematycznej redukcji liczby błędów w oprogramowaniu. Istnieje wiele różnych narzędzi i technik, które można wykorzystać do debugowania kodu np. IntelliJ, które umożliwiają debugowanie kodu w czasie rzeczywistym. Dzięki takim narzędziom można śledzić wykonanie kodu linijka po linijce, ustawiać punkty przerwania, a także przeglądać zawartość zmiennych i stosu wywołań.

Reprodukcja błędu → Wyizolowanie źródła błędu → Identyfikacja przyczyny awarii → Usunięcie defektu → Weryfikacja powodzenia naprawy

<https://stormit.pl/debugowanie/>

GIT

Popularny systemem kontroli wersji, który umożliwia zarządzanie zmianami w kodzie źródłowym. Pozwala on na tworzenie wielu wersji kodu (tzw. gałęzi), a także łatwe przechodzenie między nimi i łączenie zmian w jedną całość. Dzięki temu możliwe jest tworzenie wspólnych projektów przez wielu programistów, a także łatwe rozwiązywanie problemów i konfliktów w kodzie. Git jest szczególnie przydatny w sytuacjach, gdy kod jest często zmieniany i aktualizowany, a także gdy wiele osób pracuje nad tym samym projektem. Jest on też często wykorzystywany w połączeniu z serwisami hostingowymi, takimi jak GitHub, gdzie można przechowywać i udostępniać kody źródłowe wspólnie z innymi osobami → <https://github.com/>

<https://stormit.pl/git/>

CODE REVIEW

Proces, w trakcie którego kod źródłowy jest sprawdzany przez inne osoby zespołu, zazwyczaj przed jego wdrożeniem do produkcji. Celem code review jest zapewnienie jak najwyższej jakości kodu, poprzez wykrycie i usunięcie ewentualnych błędów, a także poprawę jego struktury i czytelności. Code review może być przeprowadzane na wiele różnych sposobów, ale najczęściej wykorzystuje się do tego specjalne narzędzia np. GitHub, które umożliwiają przeglądanie kodu i dodawanie komentarzy przez wielu użytkowników. Code review jest szczególnie przydatne w dużych zespołach programistów, gdzie wspólna praca nad kodem może prowadzić do powstawania różnych błędów i niejasności.

<https://stormit.pl/code-review/>



ŚWIAT STORMIT

STORMIT BLOG

Poznaj tematy zarówno z zakresu Javy jak i programowania – wiele ciekawych artykułów czeka na Ciebie :)

<https://stormit.pl/>

EBOOK 8 RZECZY

Dowiedz się, jakie 8 niezbędnych umiejętności, pomoże Ci dostać pierwszą pracę jako Junior Java Developer.

<https://kierunekjava.pl/zacznij/>

CZUJESZ, ŻE CHCESZ JESZCZE WIĘCEJ?

Pragniesz jeszcze bardziej zagłębić się w świat Javy i programowania? Zajrzyj do moich programów i podążaj dalej w kierunku pierwszej pracy jako programista oraz stań się jeszcze lepszym programistą!

KIERUNEK JAVA

Przejdź transformację od „ZERA” do komercyjnego programowania w Java. Tu poznasz wszystkie tajniki Javy, które pozwolą Ci stworzyć pierwszą aplikację Javową.

<https://kierunekjava.pl/>



Storm IT