# Rozdział I

**int** r = **new** Random().nextInt(10); - zwraca liczę od 0 do 9

Istnieje specjalny pakiet w Javie, którego nie trzeba importować: **java.lang**. Jest on importowany automatycznie. Dlatego korzystając z klasy **System** nie potrzebujemy nic importować. Znajdują się tam m.in.: Double, Math, Object, Number, StringBuilder, System, String

## Konflikt nazw

**import** java.util.Date;

**import** java.sql.Date;

Powyższy zapis spowoduje błąd.

Aby użyć dwóch klas o takiej samej nazwie, w przypadku przynajmniej jednej z nich należy używać całej ścieżki obiektu np.:

**import** java.util.Date;

**class** Conflicts {

Date date;

java.sql.Date sqlDate;

### Typy prymitywne

Kompilator wykrywa czy podana wartość nie wychodzi poza dopuszalny zakres danego typu.

**byte** = 129; - nie skompiluje się.

Aby przypisać liczbę większą niż zakres int, do typu long zależy poinformować kompilator, że typ jest long (domyślnie potraktuje go jako int)

**long** max = 3123456789; // nie skompiluje się

W tym celu należy dopiasć do liczby L lub l

// float j = 20.0; //nie skompiluje się

**float** k = 20;

**float** l = (**float**)20.0;

Nie można przypisać wartości **null** do typów prostych.

### Deklaracja zmiennych

Można deklarować dowolną ilość zmiennych w jednej deklaracji o ile wszystkie są jednego typu.

Można je również wszystkie zainicjować, w jednej linii, albo tylko niektóre z nich. Przy czym typ danych należy podać tylko raz w jednej deklaracji!

Nie można używać nie zainicjowanych zmiennych. Nawet jako argument metody.

Nie można rzutować tablicy int na tablicę Integer.

## Niszczenie obiektów

Metoda System.gc() – może, ale nie musi uruchomić działanie Garbage Collectora.

### Metoda finalize()

Metoda z klasy Object. Jest wywoływana gdy garbage collector próbuje usunąć obiekt. Jeśli **GC** nie powiedzie się usunięcie obiektu i następnie spróbuje ponownie, to metoda nie zostanie wywołana drugi raz.

## Opeartory dwuargumentowe arytmetyczne

* Jeśli dwie wartości mają różne typy, Java automatycznie zmieni typ jednej z tych wartości na wyższy wśród tych dwóch typów.
* Jeśli jedna z wartości jest liczbą całkowitą, a druga zmienno-przecinkową, Java automatycznie zmieni liczbę całkowitą na liczbę zmienno-przecinkową.
* Miejsze typy danych: byte, short i char, są najpierw zmieniane na int za każdym razem gdy są na nich wykonywane dwuargumentowe operacje arytmetyczne (czyli np. wykonanie operacji ++ na short nie powoduje zmiany na int), nawet jeśli żaden z argumentów nie jest int
* Po wykonaniu wszystkich powyżej opisanych zmian typów, wartość końcowa operacji ma taki sam typ danych jak nowo -dostosowane typy danych.

short g = e+f; //Nie skompiluje się

**short** gg = 20000+200; //skompiluje

## Pozostałe operatory dwu-argumentowe

Wymagane na egzaminie operatory przypisania: „=”, „+=”. „-=”

Skrócone operatory przypisania (np. += ) są użyteczne nie tylko jako skrót, ale także mogą uchronić nas przed rzutowaniem typów. Na przykład:

**long** j = 10;

**int** k = 5;

k = k \* j; - Nie skompiluje się

**long** j = 10;

**int** k = 5;

k \*= k \* j; - Skompiluje się poprawnie

W powyższym przykładzie, użyty operator sprawi, że najpierw k zostanie zrzutowane na **long**, wykona się mnożenie dwóch wartości typu **long**, i następnie rezultat zostanie zrzutowany do typu **int**.

## Wyrażenia Java

### Switch

Typy danych obsługiwane przez wyrażenie switch:

* int and Integer
* byte and Byte
* short and Short
* char and Character
* String
* enum values

Wartość w każdym wyrażeniu **case** musi być niezmienna w trakcie kompilacji oraz tego samego typu co wartość w **switch.** Niezmienna w trakcie kompilacji oznacza, że można użyć tylko **literałów** (np. 5), stałych enum albo zmiennych **finalnych**.

Gdy **case** spełnia warunek zostają wykonane instrukcję wewnątrz niego, jeśli nie jest zakończony instrukcją **break,** zostaną wykonane wszystkie kolejne **case’y** niezależnie czy spełniają warunek, aż do momentu zapotkania pierwszego **break. Switch może nie mieć żadnego case’a;**

### While

Nie są wymagane nawiasy klamrowe w przypadku jednej linii zarówno w while, jak i do-while

### For

W petli for można zdefiniować dowolną ilość jej składowych w dowolnych kombinacjach. Można naet nie zdefiniować żadnego.

**for**(;;); - poprawny zapis

**for**(;); - błąd kompilacji

**for**(); - błąd kompilacji

Wstawienie jakiekolwiek kodu PO pętli for( ; ; ) – powoduje błąd kompilacji. UNREACHABLE CODE

Pamiętaj! - zmienne zadeklarowane w inicjalizayjnym bloku pętli for maja zasięg tylko w ramach pętli.

Pętla for-each może być wykorzystywana do iteracji na tablicach i Kolekcjach implementujących interfejs iterable

### Label

Wyrażenia Java można oznaczać za pomocą label’a. Nazwa labelki tworzona jest wg takich samych zasad co nazwy zmiennych.

### Break i continue

Insstrukcja **break** powoduje wyjście z najbardziej zagnieżdżonej pętli. Aby wyjść z pętli zewnętrznych, należy skorzystać **z break z labelką.** Analogicznie działa continue, tyle, że zamiast wyjścia z pętli powoduje, przejście do następnej jej iteracji.

W instrukcji **if generalnie nie można** umieszczać break. Jeśli oznaczymy wyrażenie **if**  za pomocą labela, wtedy możemy użyć break z labelem w środku **ifa.** Dzięki temu można opuścić zagnieżdżonego ifa.

# Rozdział3

## String

String has a fixed size (tłum. String ma staly rozmiar)

### Konkatenacja

Zasady:

* Jeśli oba argumenty operacji są numeryczne, „+” oznacza dodawanie numeryczne
* Jeśli którykolwiek z argumentów operacji jest **Stringiem** , „+” oznacza konkatenacje
* Wyrażenie jest wykonywane od lewej do prawej

s += „2” znaczy dokładnie to samo co s += 2

String łączony z jakimkolwiek innym typem, zwraca String.

String a = "bb";

a+=**new** ArrayList<>();

Stringi można łączyć poprzez dodawanie (+), ale nie można odejmować. Nie można zainicjować Stringa za pomocą wartości char np. ‘a’. Ale można konkatenować chary.

Stringi tworzone za pomocą konkatenacji umieszczane są w String pool, o ile oba argumenty są stałymi w czasie kompilacji tak jak **literały czy zmienne final.**

### Metody klasy String

Ponieważ String jest klasą niezmienną, żadna z metod wywoływana na obiekcie String, nie może zmienić tego obiektu. Dlatego wszystkie zwracają nowy obiekt String.

* Length() ( w tablicach jest zmienna length)
* *char charAt(int index) – zwraca znak występujący na danej pozycji.*
* *indexOf – przyjmuje znak, może być String. Zwraca indeks pierwszego napotkanego znaku.*
* *substring() – Zwraca część napisu z podanego zakresu. Index startowy jest inkluzywny, index końcowy ekskluzywny. Podanie indeksu końcowego wyższego niż długość Stringa+1, spowoduje wyjątek. Podanie liczby startowej wyższej niż końcowej – spowoduje wyjątek.*
* *toLowerCase()*
* *toUpperCase()*
* *equals() – jest wrażliwe na wielkość liter*
* *equalsIgnoreCase()*
* *startsWtih()*
* *endsWith()*
* *contains()*
* *replace()*
* *trim() – usuwa białe znaki z początku i końca napisu*
* *concat()*

## StringBuilder

Metoda append() – dadaje fragment tekstu o obiektu na, którym została wywołana i dodatkowo zwraca referencję do tego obiektu.

Zarówno **StringBuilder** jak i **StringBuffer** **nie** są **immutable**.

Metody append, insert, replace, reverse, delete – zwracają zmieniony obiekt StrinBuilder.

Metoda **substring** zwraca **Stringa.**

### Metody

* charAt()
* indexOf()
* length()
* substring() – zwraca String, nie StringBuilder

4 Powyższe metody działają dokładnie tak samo jak metody z Klasy String

* append()
* inseret() – dodaje nowe znaki w określonym przez index miejscu
* delete() – usuwa zakres
* deleteCharAt() – usuwa jeden znak
* reverse()
* capacity()

metoda delete() **nie zwraca błędu jeśli przekroczymy długość. Np.:**

**StringBuilder sb = new StringBuilder(„JAVA”);**

**Sb.delete(0,100); - nie zwraca błędu.**

**Błąd** zostanie rzucony gdy:

* Start jest ujemny
* Start jest większy niż koniec
* Start jest większy niż sb.lenght()

Metoda toString zwraca String, który nie jest przechowywany w **String pool, ponieważ jest tworzony z użyciem konstruktora.**

## Sprawdzanie równości obiektów

Wywołanie metody equals() w celu porównania dwóch obiektów StringBuilder zwróci false, ponieważ StringBuilder nie rozszerza metoda equals(), dlatego porównywane są referencje.

## Tablice

Aby zainicjować tablicę należy podać albo jej wielkość, albo listę elementów. Nie można podać obu.

**int**[] array = **new** **int**[3];

powyższy zapis tworzy tablicę o pojemności 3 elementów. Ponieważ nie podaliśmy jakie to są elementy, każdy zostaje zainicjowany domyślną wartością, w tym wypadku zerem.

Poprawne deklaracje:

**int**[] array = **new** **int**[3];

**int**[] array2 = { 3, 0, 2 };

**int**[] array5 = **new** **int**[] { 1, 4, 10 };

**int**[]array8 = { 1 }; // brak spacji w deklaracji

**int** array11[] = { 1 }, a1 = 2;

**Złe deklaracje:**

int [] array4 = new int[3] {1,4,10};

int array14[] = new int {1, 4, 6};

**int** [] z ;

z = {1, 2};

Nie można zmienić wielkości tablicy. Ale można tablicę przypisać do innej zmiennej, która wcześniej przechowywała mniejszą tablicę.

Tablicę można tworzyć dowolnych typów, włącznie z samodzielnie utworzonymi klasami.

Aby wyświetlić zawartość tablicy należy użyć metody toString() z klasy **Arrays.** Metoda **toString()**  wywołana bezpośrednio na tablicy zwróci typ obiektu i jego hashcode.

**Dwie tablice z tą samą zawartością nie będą równe! Tablice nie mają nadpisanej metody equals, więc porównywane będą zawsze referencje.**

### Wyszukiwanie

W klasie **Arrays**  znajduję się metoda **binarySearch()**, która służy do wyszukiwania indexu elementu w tablicy. Aby wyszukiwanie tą metodą przyniosło przewidywany skutek, tablica musi być **posortowana**. W innym przypadku wynik wyszukiwania będzie **nieprzewidywalny.**

**Jeśli element nie zjaduje się w tablicy zostanie zwrócona liczba =** (-(*insertion point*) - 1),

Gdzie insertion point : index pierwszego elementu większego niż wyszukiwany.

### Varargs

Można używać zmiennej varargs jak normalnej tablicy np.

**public** **static** **void** main(String... args) // varargs

args.length

## ArrayList

Klasa ArrayList nie posiada metody **capacity(). (Ma ją StringBuilder)**

### remove()

* metoda remove(Object o) – usuwa tylko pierwszy napotkany egzemplarz obiektu. Zwraca **true** albo **false**, w zależności czy znajdzie obiekt na liście.
* Metoda remove(int index) – usuwa element znajdujący się pod wybranym indexem. Jeśli index znajduję się poza zakresem, **rzuci wyjątkiem**. **Zwraca usunięty obiekt** (nie boolean!) .

Można usuwać null tj. zapisać list.remove(null)

### equals()

ArrayList posiada metodę equals(), która sprawdza czy 2 listy posiadają dokładnie te same elementy, w tej samej kolejności.

Get() – rzuci wyjątek (nie null!) jeśli przekroczymy indeks.

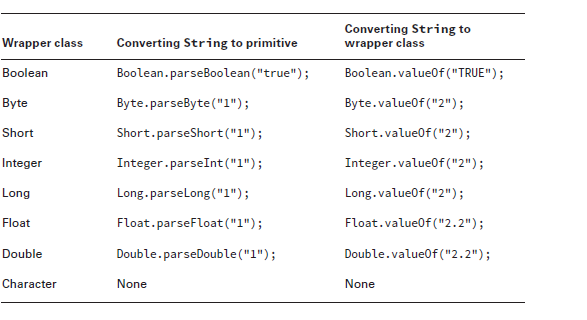
Clone() – wykonuje klonowanie płytkie, tzn. lista1!=lista2, ale lista1.get(0)==lista2.get(0)

If you want to remove the items from ArrayList, while using Iterator or ListIterator, then use Iterator.remove() or ListIterator.remove() method and NOT List.remove(...) method. Using List.remove(...) method while iterating the list (using the Iterator/ListIterator or for-each) may throw java.util.ConcurrentModificationException.

## Wrapery

Integer.valueOf() – zwraca obiekt **Integer**

Integer.parseInt() – zwraca typ prymitywny **int.**



// Short b = Short.valueOf(1);

// Byte c = Byte.valueOf(2); - nie skompilują się

|  |  |
| --- | --- |
|  | **Dostępne wersje metody valueOf w Klasie Short:**  [**valueOf**](https://docs.oracle.com/javase/7/docs/api/java/lang/Short.html#valueOf(short))(short s) |
|  | [**valueOf**](https://docs.oracle.com/javase/7/docs/api/java/lang/Short.html#valueOf(java.lang.String))**(**[**String**](https://docs.oracle.com/javase/7/docs/api/java/lang/String.html)**s)** |
|  | [**valueOf**](https://docs.oracle.com/javase/7/docs/api/java/lang/Short.html#valueOf(java.lang.String,%20int))**(**[**String**](https://docs.oracle.com/javase/7/docs/api/java/lang/String.html)**s, int radix)** |

Można przypisać Obiekt Wrapera do typu prymitywnego.

Jeśli zmienna wrapera będzie zawierać null wystąpi NullPointerException:

Double d = **null**;

**double** f = d; - //NullPointerException

Inkrementacja obiektu Integer powoduje utworzenie nowego obiektu!

Porównanie (==) dwóch obiektów Integer, czy nawet Boolean przechowujących tą samą wartość zwróci **false, jeśli będą utworzone za pomocą konstruktora. Przy autoboxingu jest inaczej.**

**Nie można przypisać obiektu Integer do zmiennej typu Double. Tak samo Short do Integer. Nawet short do Integer się nie da. Ale można int do Double.**

Zamiast tego można zrobić:

Short ss = 2;

Integer ss3 = ss.intValue();

Two instances of following wrapper objects, created through auto-boxing will always be same, if their primitive values are same: (Również gdy korzystamy z metody valueOf())

* Boolean,
* Byte,
* Character from \u0000 to \u007f (7f equals to 127),
* Short and Integer from -128 to 127.

For 1st statement, list.add(27); => Auto-boxing creates an integer object for 27.

For 2nd statement, list.add(27); => Java compiler finds that there is already an Integer object in the memory with value 27, so it uses the same object.

### Konwersja pomiędzy listą a tablicą

Tablica na listę:

List<String> list10 = Arrays.*asList*(array);

Zmiany na liście powodują zmiany w tablicy, z której została utworzona, dlatego operacje add() i remove() powodują wyjątek.

Lista na tablicę:

|  |  |
| --- | --- |
| [Object](https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html)[] | [toArray](https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html#toArray--)() |
| <T> T[] | [toArray](https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html#toArray-T:A-)(T[] a) |

Dzięki podaniu tablicy w argumencie, możemy wybrać typ zwracanej tablicy oraz zwiększyć jej rozmiar.

## Data i czas

Pamiętaj o importach gdy pojawią się klasy związane z datą lub czasem. Większość z nich znajduje się w pakiecie **java.time**.

W zależności od ilości informacji, które potrzebujemy można wyróżnić trzy klasy:

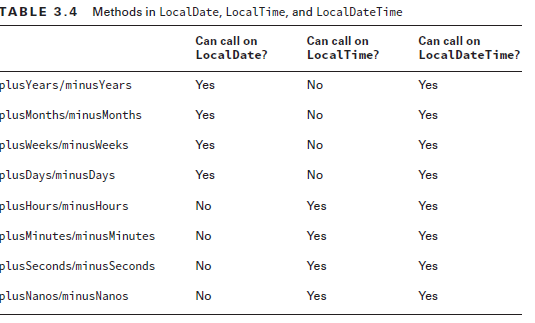
* LocalDate – Zawiera tylko datę – bez czasu i strefy czasowej.
* LocalTime – Zawiera tylko czas – bez daty i strefy czasowej.
* LocalDateTime – Zawiera datę i czas, ale bez strefy czasowej.

Obiekty klas tworzy się za pomocą metody statycznej ***of(),*** Klasy te nie posiadają publicznych konstruktorów.

Metoda **of()** w klasie LocalDateTime przyjmuje najmniej 5 argumentów (minuty są wymagane).

### Operowanie na dacie i czasie

Klasy reprezentujące czas i datę są **niezmienialne**. Więc wynik operacji dokonanych na obiektach, należy przypisać do zmiennej.



Powyższe metody przyjmują **long** jako argument. Można przekazywać wartości ujemne, stosując zasadę, że dwa minusy dają plus.

Minimalna obsługiwana data to {-999999999-01-01} i maksimum to {+999999999-12-31}.

Wynika z tego, że lata przed nasz erą poprzedzone są minusem.

### Periody

Klasa Period również jest immutable.

Metoda toEpochDay() – zwraca liczbę dni, która upłynęła od 01.01.1970r.

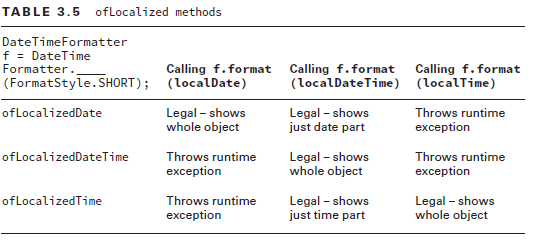
### Formatowanie

**DateTimeFormatter –** klasa umożliwiająca formatowanie zarówno daty i czasu. Znajduję się w pakiecie **java.time.format.**

Obiekty klasy tworzymy za pomocą metod statycznych :

* [ofLocalizedDate](https://docs.oracle.com/javase/8/docs/api/java/time/format/DateTimeFormatter.html#ofLocalizedDate-java.time.format.FormatStyle-)([FormatStyle](https://docs.oracle.com/javase/8/docs/api/java/time/format/FormatStyle.html" \o "enum in java.time.format) dateStyle)
* [ofLocalizedDateTime](https://docs.oracle.com/javase/8/docs/api/java/time/format/DateTimeFormatter.html#ofLocalizedDateTime-java.time.format.FormatStyle-)([FormatStyle](https://docs.oracle.com/javase/8/docs/api/java/time/format/FormatStyle.html" \o "enum in java.time.format) dateTimeStyle)
* [ofLocalizedTime](https://docs.oracle.com/javase/8/docs/api/java/time/format/DateTimeFormatter.html#ofLocalizedTime-java.time.format.FormatStyle-)([FormatStyle](https://docs.oracle.com/javase/8/docs/api/java/time/format/FormatStyle.html" \o "enum in java.time.format) timeStyle)
* [ofPattern](https://docs.oracle.com/javase/8/docs/api/java/time/format/DateTimeFormatter.html#ofPattern-java.lang.String-)([String](https://docs.oracle.com/javase/8/docs/api/java/lang/String.html) pattern)

Powyższe 3 pierwsze metody tworzą formater odpowiedni dla konkretnego typu reprezentującego Czas lub/i datę. Poniżej rozpiska:



Z powyższej tabelki wynika, że to Formater jest ważniejszy. Nie da się zastosować formatu typu „data i czas” do pokazania tylko daty lub tylko czasu. Innymi słowy muszę mieć wszystkie informacje w wymagane w danym formacie aby go zastosować. Mogę mieć więcej informacji niż wymaga format, ale nie mogę mieć mniej.

Zarówno DateTimeFormater, jak i klasy reprezentujące czas i datę, posiadają metodę ***format()***, która zwraca **String.** Nie ma chyba znaczenia , której użyjemy.

### Parsowanie

Metoda parse() przyjmuje jako argument string, który może być tylko w formacie „9999-99-99”, chyba, że podamy jako drugi argument odpowiedni formater.

* M – oznacza miesiąc. Im więcej literek M, tym bardziej rozległa informacja. Na przykład M zwraca 1, MM zwraca 01, MMM zwraca Jan, MMMM zwraca January.
* dd- dzień w miesiącu
* yyyy – y oznacza rok
* hh – h oznacza godzinę
* mm – m oznacza minuty
* DD – reprezentuje dzień roku

# Rozdział 4. Metody

### Modyfikatory dostępu

* Protected – metoda może być wywołana z z tej samej klasy, podklas i tego samego pakietu
* Default – metoda może być wywołana z pakietu.

Default to inaczej **Package private.**

Specyfikatory opcjonalne mogą znajdować się przed modyfikatorem dostępu, ale **nie mogą być po typie zwracanym.**

### Typ zwracany

Zwracana zmienna musi być typu użytego w deklaracji metody. Innymi słowy: Jeśli nie jesteś w stanie przypisać obiektu do zmiennej danego typu, no nie możesz go zwrócić w metodzie.

## Varargs

Varargs traktujemy jak tablicę.

Parametr vararg musi zawsze znajdować się na ostnim miejscu listy parametrów, co oznacza, że może być maksymalnie jeden vararg w parametrach metody.

Wywołując metodę z parametrem vararg masz wybór. Możesz podać jako argument:

* - tablicę
* - listę elementów, wtedy Java stworzy tablicę za Ciebie.
* - pominąć argument, wtedy Java stworzy tablicę o długości 0 elementów

## Static

Metodę  **main** można wywołać w kodzie jak każdą inną.

Zmienne i metody statyczne można wywołać na zmiennej do, której przypisany jest null!. Nie powoduje to **NullPointerException.** Nie można wywołać elementów statycznych na niezainicjowanej zmiennej.

Będąc wewnątrz klasy nie trzeba używać jej nazwy w celu odwołania się do zmiennej statycznej.

### Statyczne bloki inicjujące

Finalne zmienne statyczne mogą być zainicjowane **tylko** przy deklaracji lub w statycznych blokach inicjujących. Co więcej zmienne te muszą zostać zainicjowane tylko w jednym z tych miejsc, inaczej kompilator zgłosi błąd.

### Statyczny import

Statyczny import jest używany do importu statycznych elementów klasy. Tak samo jak przy zwykłym imporcie, można importować przy użyciu znaku wieloznacznego (\*) lub określając konkretny element.

Analogicznie jak przy normalnym imporcie, kompilator rzuci błąd jeśli zrobimy statyczny import dwóch metod o takiej samej nazwie, lub dwóch zmiennych statycznych o tej samej nazwie.

**import** **static** java.util.Arrays.*asList*; - dobrze

//static import java.util.Arrays.\*; - źle

## Przeciążanie metod

Java traktuje varargs jak tablicę więc nie można mieć metody o tej samej nazwie z tablicą i varargs tego samego typu, jako parametrem. Obie metody można wywołać z tablicą jako argumentem, ale wersje z varargs, można wywołać podając jako argument varargs!

*doA*({"a"}); - nie poprawne wywołanie metody przyjmującej varargs lub tablicę

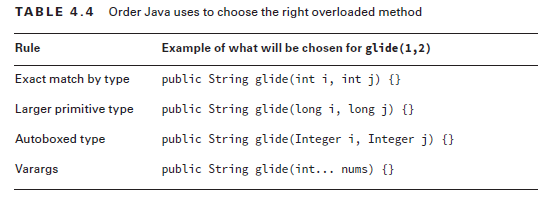
Mogę istnieć metody o tej samej nazwie – jedna z argumentem typu **int,** a druga **Integer.**

**Java stara się wywołać najbardziej sprecyzowaną metodę jaką może.Przykladowo jeśli jako argument podamy String to będzie szukać metody w parametrem String, a nie Object.**

**Kolejność: Najpierw szukane jest dokładne dopasowanie typów, później szersze prymitywy, później autoboxing i na końcu varargs.**

Starając się znaleźć odpowiednią metodę, Java jest w stanie wykonać tylko jedną konwersję.

int -> Long – nie skompiluje się.



## Tworzenie konstruktorów

Słowo kluczowe **this** jest w większości przypadków opcjonalne. Problem pojawia się gdy istnieją dwie zmienne o tej samej nazwie. Przykład: jedna zmienna jest parametrem konstruktora, druga zmienną klasową. Jeśli nie poinformujesz Javy inaczej, użyje tej, która ma mniejszy zasięg. Czyli bez słówka this zostanie zmieniona zmienna reprezentująca parametr metody. (To samo wydarzy się jeśli zadeklarujemy w ciele nową zmienną lokalną o tej samej nazwie co zmienna klasowa.)

### Przeciązanie konstruktorów

**This()** musi być pierwszą niezakomentowaną komendą w konstruktorze.

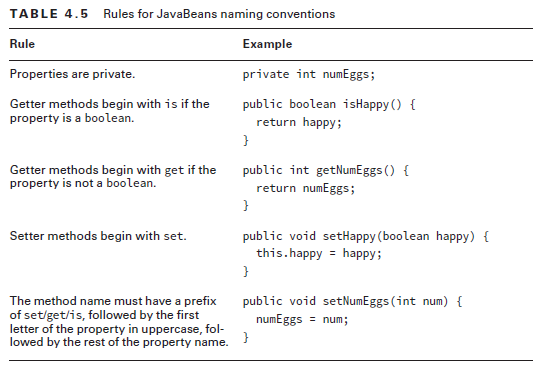
### Pola final

Przed zakończeniem pracy konstruktora, wszystkie zmienne finalne muszą zostać zainicjowane. Nie zostaną one zainicjowane domyślne, tak jak zwykłe zmienne klasowe. Zmienną finalną można zainicjować tylko w konstruktorze, który nie wywołuje żadnych innych konstruktorów.

## Enkapsulacja danych

Poprawna eknapsulacja na egzamin zawiera:

* Prywatne pola
* Publiczne gettery i settery



Wg konwencji JavaBeans:

Jeśli mam pole typu boolean – canSwim. Getter public boolean getCanSwim() { return canSwim;}

Nie jest zgodny z konwencją. (To co miało by być isCanSwim wg tej konwencji?!)

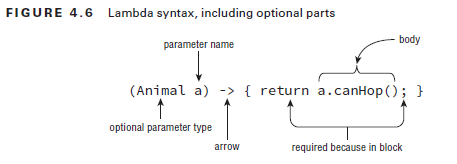
PS. A jednak w erracie jest napisane, że getCanSwim jest poprawne, ale żzeby się nie przejmować bo nie ma tego na egzaminie.

### Tworzenie klas niezmiennych

Zasady:

* Wszystkie pola prywatne
* Brak seterów
* Gettery zwracają kopie obiektów
* Pola finalne
* Klasa oznaczona jako finalna

### Lambda – składnia



* Nawiasy mogą być pominięte, jeśli jest tylko jeden parametr i nie podajemy jego typu.
* Klamry mogą być pominięte tylko jeśli jest jedna linia kodu.
* Jeśli są klamry – wymagany jest return i średnik.
* Return i średnik może być tylko jeśli są klamry

Na egzaminie wyrażenia lambda zawsze zwracają typ **boolean**.

Nie można przeprowadzać ponownej deklaracji zmiennej w ciele wyrażenia lambda

(a, b) -> { **int** a = 0; **return** 5;} // DOES NOT COMPILE

(a, b) -> { **int** c = 0; **return** 5;} // COMPILE

### Predicate

Lambda działa z interfejsami, które mają tylko jedną metodę abstrakcyjną. Nazywamy je **interfejsami funkcyjnymi.** Biblioteka javy dostarcza wiele zdefiniowanych interfejsów funkcyjnych. Jednym z nich jest **Predicate.**

W klasie ArrayList znajduję się metoda ***removeIf()*** *,* która przyjmuje **Predicate** jako argument. Dzięki niej możemy usunąć z listy obiekty, które spełniają określony warunek.

# Rozdział 5

## Wprowadzenie do dziedziczenia

### Modyfikatory dostępu

Do egzaminu OCA, powinieneś być zaznajomiony z modyfikatorami **public i default** w kontekście klas, ponieważ tylko te dwie mogą być użyte w klasach. **protected i private**  można zastosować tylko do klas wewnętrznych, ale ich nie ma na egzaminie.

**public –** oznacza, że klasa może być użyta w dowolnej innej klasie

**default (nie za pisujemy w kodzie) –** Ozanacza dostęp do klas z tego samego pakietu lub z podklas.

W pliku może znajdować się co najwyżej jedna klasa lub interfejs z oznaczeniem **public**

### Definiowanie konstruktorów

**Każdy konstruktor w Javie jako pierwsze wyrażenie posiada this(), albo super().**

Zarówno  **super()** , jak i **this()** muszą być pierwszym wyrażeniem w konstruktorze.

Jeśli nadklasa posiada konstruktory, ale nie ma wśród nich **konstruktora bez parametrowego**, wtedy pod klasa musi posiadać co najmniej jeden kostruktor i musi on jawnie wywołać **super().** (Dlatego, że konstruktor domyślny wywołuje super() bez parametrów, a takiego konstruktora nie ma w nadklasie).

Konstruktor podklasy może wywołać konstruktor nadklasy o innej liczbie parametrów niż on sam.

Kompilator Javy automatycznie wstawia super() do każdego konstruktora, jeśli jawnie go nie wpiszemy. Poniższe 3 klasy są równorzędne

### Wywoływanie konstruktorów

W Javie konstruktory nadklasy są zawsze wykonywane przed konstruktorami podklasy.

## Wywoływanie dziedziczonych elementów klasy

Klasa Java może używać wszystkich elementów klasy nadrzędnej o widoczności **public i protected, i ewentualnie default,** jeśli znajdują się w tym samym pakiecie.

Jeśli są w innym pakiecie to w przypadku widoczności **protected widoczność** jest ograniczona. A mianowicie, nie można uzyskać dostępu do elementów Obiektu nadklasy utworzonego w obrębie podklasy. (ClassDD)

Za pomocą słówka **this** można uzyskać dostęp elementów nadklasy o ile widoczność pozwala.

Zarówno **this**  i **super**  mogą być używana do wywoływania metod i zmiennych zdefiniowanych w nadklasie, ale tylko **this** może być użyte do elementów zdefiniowanych w obecnej klasie.

## Dziedziczone metody

### Nadpisywanie metod

Zasady:

* Aby nadpisać metodę w podklasie musi mieć taką samą sygnaturę(nazwa i lista parametrów) jak metoda w podklasie
* Metoda w podklasie musi mieć taki sam lub większy poziom widoczności co metoda w nadklasie.
* Metoda w podklasie nie może rzucać checked exception (te która trzeba osbłużyć), które są nowe lub szersze niż wyjątki rzucane w nadklasie.
* Jeśli metoda zwraca wartość musi być tego samego typu, lub podklasą wartości z nadklasy.

Jeśli metoda w podklasie ma taką samą nazwe, ale inne parametry niż metoda z nadklasy, to nie jest nadpisywanie tylko przeciążanie metody.

### Redeklaracja metod prywatnych

Nie można w Javie nadpisywać metod prywatnych. Nie oznacza to jednak, że podklasa nie może zdefiniować własnej wersji metody.

### Ukrywanie metod statycznych

Ukryte metody (hidden methods) pojawiają się gdy podklasa definiuje statyczną metodę o takiej samej nazwie i sygnaturze jak statyczna metoda nadklasy. Zakrywanie metody jest podobne, ale nie tym samym co nadpisywanie metody. 4 zasady dotyczące nadpisywania metod mają zastosowanie również tutaj, dodatkowo dochodzi piąta zasada.:

* Aby nadpisać metodę w podklasie musi mieć taką samą sygnaturę(nazwa i lista parametrów) jak metoda w podklasie
* Metoda w podklasie musi mieć taki sam lub większy poziom widoczności co metoda w nadklasie.
* Metoda w podklasie nie może rzucać checked exception (te która trzeba osbłużyć), które są nowe lub szersze niż wyjątki rzucane w nadklasie.
* Jeśli metoda zwraca wartość musi być tego samego typu, lub podklasą wartości z nadklasy.
* Metoda zdefiniowana w podklasie musi być oznaczona jako **static**  jeśli jest oznaczona jako **static** w nadklasie.

### Nadpisywanie vs Zakrywanie

W czasie wykonania programu, zostanie wykonana wersja nadpisywanej metody z podklasy w przypadku instance method, nie zależnie czy metoda jest wywoływana wewnątrz klasy parenta czy childa. W związku z tym, wersja z nadklasy, nie jest nigdy wywoływana.

W przypadku zakrytych metod: Zostanie wykonana wersja zakrytej metody z nadklasy, jeśli wywołanie tej metody odbędzie się w nadklasie.

Jeśli metoda statyczna wywoływana jest na referencji obiektu, zostanie uruchomiona wersja z Klasy o typie referencji.

Za pomocą referencji typu podklasy, można wywołać metodę statyczną z nadklasy. (Gdy podklasa nie zakrywa tej metody.)

### Metody finalne

Metody finalne nie mogą być nadpisywane.

Nie można zakrywać statycznych metod finalnych

### Zakrywanie zmiennych (ClassŁ)

Zakrywanie zmiennych – definiowanie zmiennej o takiej samej nazwie jak zmienna w nadklasie. Powoduje to, że podklasa posiada dwie zmienne o tej samej nazwie.

Jest to podobny mechanizm do zakrywania metod statycznych. Jeśli odnosisz się do zmiennej z wnętrza nadklasy, zmienna zdefiniowana w nadklasie będzie użyta, natomiast jeśli odnosisz się do zmiennej z wnętrza podklasy – zmienna z podklasy będzie użyta. Jeśli odnosisz się za pomocą referencji – zdecyduje typ referencji.

### Definiowanie klas abstrakcyjnych.

Klasa abstrakcyjna może posiadać metody i zmienne konkretne. Klasa abstrakcyjna może nie mieć ani jednej abstrakcyjnej metody. W klasie abstrakcyjnej mogą istnieć metody konkretne. Metody abstrakcyjne nie mogą być private, static oraz final.

Zmienne mogą być prywatne, statyczne oraz finalne.

Klasa abstrakcyjna nie może być oznaczona jako final.

Nadpisywanie metod abstrakcyjnych obowiązuje te same zasady co przy nadpisywaniu zwykłych metod. Np. nadpisując metodę abstrakcyjną, metoda w nadklasie musi ten sam lub mniejszy poziom widoczności.

Klasa abstrakcyjna może posiadać konstruktor.

## Interfejsy

Interfejs domyślnie jest **abstract i public**. Nie trzeba tego pisać (nic to nie zmienia jeśli się napiszę).

Interfejs może nie posiadać żadnej metody. Interfejs nie może być oznaczony jako final i static, private, protected.

Metody bez słowa default, są traktowane jako public i abstract (nie trzeba tego pisać, ale można), a skoro tak to nie mogą być jednocześnie private, protected i final.

Wszystkie metody ( w tym domyślne i statyczne) są publiczne.

Zmienne w interfejsach zawsze są **public, final, static!!!**

**Ponieważ** są **final**  muszą być zawsze **zainicjowane**  podczas deklaracji.

Klasa może implementować dwa interfejsy mające metodę o tej samej nazwie (i parametrach), pod warunkiem, że tym zwracany obu metod jest ten sam.

Klasa może zarówno rozszerzać inną klasę i implementować interfejsy. Najpierw należy podać klase

**class** classD **extends** Object **implements** InterfaceB

### Metody domyślne

Metody domyślne posiadają zaimplementowane ciało. Aby je zdefiniować należy użyć słówka **default. Domyślnie (bez podania specyfikatora) metody są abstract.**

Klasa implementująca może, ale nie musi nadpisywać metodę domyślną.

**Metody domyślne tak jak każde w interfejsie, muszą być publiczne!! Nie mogą być statyczne i finalne.**

Interfejs implementujący inny interfejs, może sprowadzić metody domyślne nad-interfejsu, z powrotem do abstract.

### Metody statyczne

Jest tylko jedna różnica między metodą statyczną w klasie i interfejsie. **Statyczne metody zdefiniowane w interfejsie nie są dziedziczone do klas, które implementują interfejs.**

Statyczne metody zdefiniowane w interfejsie można wywołać tylko używając nazwy interfejsu, w którym są zdefiniowane. Natomiast nie można ich wywołać poprzez referencje o typie danego interfejsu, ani używając nazwy interfejsu który rozszerza interfejs z metodą statyczną!!

Klasa może implementować dwa interfejsy posiadające metodę statyczną o tej samej nazwie.

## Polimorfizm

Kiedy obiekt zostanie przypisany do referencji określonego typu, tylko te metody i zmienne są dostępne zdefiniowane w tym typie. (bez rzutowania)

### Rzutowanie obiektów

Zasady:

* Rzutowanie obiektu z podklasy na superklasę nie wymaga jawnego rzutowania
* Rzutowania obiektu z nadklasy do podklasy wymaga jawnego rzutowania
* Kompilator nie pozwoli na rzutowanie do typu, który nie jest w żaden sposób powiązany z dotychczasowym
* Nawet jeśli kod się skompiluje, może zostać rzucony wyjątek jeśli obiekt jest rzutowany do typu, którego nie jest instancją.

Ważne klasy: StringTest, ArraysTest, ArrayListTest(wrappery i autoboxing), DateAndTime(formatowanie i parsowanie), ClassDD, ClassJ, ClassK, ClassL, ClassŁ, StaticMethod,