# Rozdział I

## Komentarze:

* // - kometarz jedno-liniowy
* /\* …. \*/ - komentarz wielo-liniowy
* /\*\* …. \*/ - komentarz Javadoc

W pliku może znajdować się wiele klas, ale tylko jedna może być oznaczona słowem public – ta, która ma tą samą nazwę co nazwa pliku.

## Metoda main (wszystkie wersje poprawne):

* **public** **static** **void** main(String... args)
* **public** **static** **void** main(String[] args)
* **public** **static** **void** main(String args[])
* **public** **static** **void** main(String[] xxx)

## Random

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

## Paczki i Import

Klasa Random znajduje się w pakiecie java.util.

**import** java.util.\*; - daje dostęp do wszystkich klas znajdujących się w util, ale nie powoduje importu podpakietów. Dlatego:

**import** java.\*; - nie powoduje importu klasy Random

**import** java.\*.\*; - nie dopuszczalny zapis

Istnieje specjalny pakiet w Javie, którego nie trzeba importować: **java.lang**. Jest on importowany automatycznie. Dlatego korzystając z klasy **System** nie potrzebujemy nic importować.

Jeżeli klasy znajdują się w dokładnie tym samym pakiecie ( w tym samym miejscu w hierarhi pakietów) to nie muszą się nawzajem importować.

Poniżej również poprawne zapisy:

**import** java.lang.\*; **public** **class** DeclaringVariables {}



## 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 {

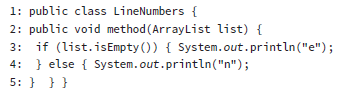
Date date;

java.sql.Date sqlDate;

}

Tip do egzaminu:

Jeśli nie widać pierwszej lini kawałka kodu, lub, nie ma w ogóle ponumerowanych linii, wtedy śmiało zakładaj, że odpowiednie importy są zrobione.



W tym przypadku odpowiedź będzie brzmieć: Kod się nie skompiluje bo brakuje importu ArrayList.

Natomiast jeśli są dwie klasy na egzaminie, wyglądają jak utworzone przez autorów tesu (nie są w bibliotece javy) i nie ma zadeklarowanych pakietów , w których się znadują, to nie przejmuj się importami. **Obie te klasy znajdują się w pakiecie domyślnym!!**

## Tworzenie obiektów

Klasa może mieć metodę o tej samej nazwie co jej własna nazwa( i nie jest to konstruktor)

**class** Test {

**public** **void** Test() {

System.***out***.println("aaa");

}

}

Powyższy zapis jest poprawny.

### Kolejność inicjalizacji

Pola i bloki inicjalizacyjne są uruchamiane w kolejności w, której występują w pliku

Konstruktor zostaje uruchomiony po zainicjowaniu wszystkich pól i wykonaniu bloków inicujących.

**public** **class** Test {

**private** String name = "Fluffy";

{

System.***out***.println(name);

}

}

Powyższy zapis jest poprawny. Najpierw zostanie zainicjowana zmienna name, a później zostanie wyświetlona jej zainicjowana wartość. Odwrotna kolejność spowodowałaby by błąd kompilacjim ponieważ nie można odnosić się do zmiennych, które nie zostały zainicjowane.

## Referencje do obiektów i typy prymitywne

### 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

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!

**double** d=1, **double** d2=2; // nie skompiluje się

* instance variable – niestatyczne pole klasy
* class variable – pole statyczne
* local variable – deklarowane wewnątrz metody.

Pamiętaj by zwracać baczną uwagę na zasięg zmiennych lokalnych, w blokach zagnieżdżonych.

Jeśli pole klasy typu String nie zostanie jawnie zainicjowane to zostanie do niego przypisane **null (nie „ ”).**

### Kolejność elementów w klasie

Kolejność zawsze musi być następująca:

* Package
* Import
* Definicja klasy

Komentarze mogą być w dowolnym miejscu, także przed pakietem.

## Niszczenie obiektów

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

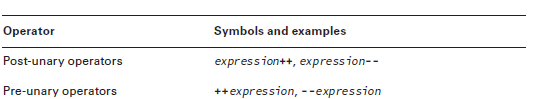
Obiekty składowane są na stercie. Zostają usnięte przez **GC** gdy nie są dłużej osiągalne. Obiekty nie są osiągalne w dwóch przypadkach:

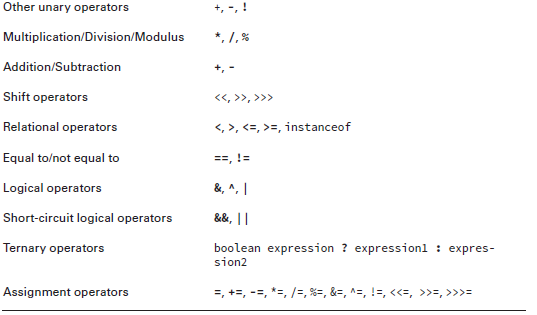
* Gdy nie istnieje, żadna referencja wskazująca na obiekt,
* Gdy wszystkie referencje do obiektu są poza zasięgiem.

### 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.

# Operatory i Instrukcje

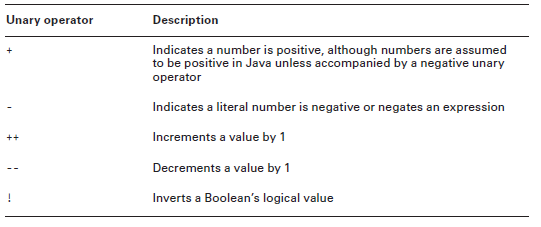




## Opeartory dwuargumentowe arytmetyczne

* 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.

## Operatory jedno-argumentowe



## 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**.

### Operatory równości

Operatorów równości można użyć w 3 przypadkach

* Porównanie dwóch typów prymitywnych. Jeśli wartości są różnych typów, to obowiązują takie same zasady jak opisane wyżej. Tzn. obie wartości zostaną sprowadzone do tego (wyższego) typu danych.
* Porównanie dwóch wartości boolean
* Porównanie dwóch obiektów, wliczając null i String

## Wyrażenia Java

### Switch

Typy danych obsługiwane przez wyrażenie switch:

* int and Integer
* byte and Byte
* short and Short
* char and Character
* int and Integer
* 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.**

### 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

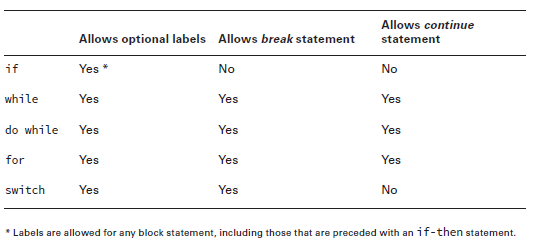
### Label

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.



# Podstawowe API Java

## 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

Metoda toString zwraca String, który nie jest przechowywany w **String pool.**

String łączony z jakimkolwiek innym typem, zwraca String.

### 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+2, 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*

## 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()

## Sprawdzanie równości obiektów

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

W metodzie equals klasy Object 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.

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.**

### Varargs

Można używać zmiennej varargs jak normalnej tablicy np.

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

args.length

### Tablice wiolowymiarowe

Tablice są obiektami i oczywiście, elementy tablicy też mogą być obiektami. Więc oczywiście elementami tablicy mogą być inne tablice. W ten sposób powstają tablice wielowymiarowe.

**int** [][] rectangle = **new** **int**[3][2];

Tablica wielowymiarowa rectangle to tak naprawdę tablica 3-elementowa, której każdy z elementów jest 2-elementową tablicą.

Tablice wielowymiarowe nie muszą mieć regularny kształtów

**int**[][] differentSize2 = **new** **int**[4][];

Nie jest konieczne podawanie 2 wymiaru. Jeśli ich nie podamy to elementy w tablicy będą nullami i jedynie deklarujemy, że wartości które możemy tam wstawić są typu int [] (tablica int).

## ArrayList

Jeśli widzisz pierwszą linie kodu to musi być import klasy ArrayList aby kod się skompilował. No chyba, że widzisz tylko wycinek metody.

**import** java.util.\*;

**import** java.util.ArrayList;

Do ArrayListy bez określonego typu, można dodać wszystkie typy obiektów oraz wartości prymitywne.

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 listę** (nie boolean!) uszczuploną o usunięty elelement.

### set()

zastępuje dotychczasowy obiekt nowym, pod wskazanym indeksem. Zwraca **zastępowany** element. Jeśli pod danym indeksem nie ma żadnego elementu rzuci wyjątkiem.

### contains()

Metoda contains() używa metody equals() do sprawdzenia czy obiekt znajduje się na liście.

### equals()

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

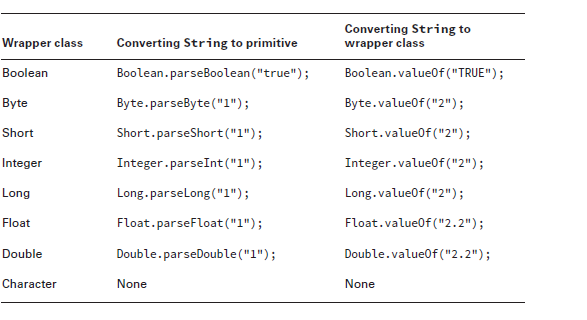
If you want to remove the items from ArrayList, while using Iterator or ListIterator, then use Iterator.remove() or ListIterator.remove() method and NOT List.remove(...) method. Using List.remove(...) method while iterating the list (using the Iterator/ListIterator or for-each) may throw java.util.ConcurrentModificationException.

## Wrapery

Integer j = Integer.*parseInt*("20");

Na egzaminie nie trzeba pamiętać jak metoda dokładnie wygląda i pamiętać np. że metoda parseInt() jest używana zamiast przykładowo parseInteger(). Po prostu trzeba pamiętać, że taka metoda isntieje i wiedzieć co robi.

Metody, które należy potrafić rozpoznawać, służące do tworzenia wartości prymitywnych lub ich wraperów ze Stringów:



### Autoboxing

### Konwersja pomiędzy listą a tablicą

String[] stringArray = list9.toArray(**new** String[0]);

Jak rozumiem: Dzięki podaniu tablicy w argumencie możemy przypisać nowo utworzoną tablicę z konkretnym typem innym niż Object. Dodatkowo wielkość tablicy w argumencie pozwala utworzyć większą tablicę niż lista elementów liście. Jeśli liczba będzie mniejsza lub równa ilości elementów w liście – powstanie tablica o takim samym rozmiarze jak lista. Jeśli będzie większa – powstanie tablica o takim rozmiarze.

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

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

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

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

## Data i czas

W java 8 Oracle całkowicie przeorganizował pracę z datą i czasem. Wciąż można pracować na nich w stary sposób, ale **nie ma go na egzaminie**. Pamiętaj o importach gdy pojawią się klasy związane z datą lub czasem. Większość z nich znajduje się w pakiecie **java.time**. Egzamin **nie uwzględnia stref czasowych.**

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.

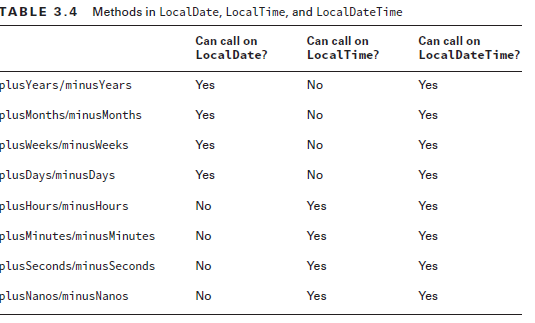
Nie wiem czy dobrze zrozumiałem: w USA miesiąc jest zapisywany przed dniem i tak też będą sformatowane daty na egzaminie(?)

W nowej wersji miesiące liczy się od **jedynki,** nie od zera.

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

### Operowanie na dacie i czasie

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

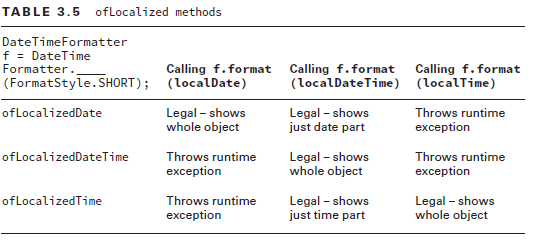


### Periody

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.**



Istnieją dwa predefiniowane formaty, które mogę pojawić się na egzaminie: **SHORT i MEDIUM.**

Oznaczenia podczas formatowania daty/czasu:

* 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

# Metody i enkapsulacja

## Projektowanie metod

### Modyfikatory dostępu

* Public – metoda może być wywołana z dowolnej klasy
* Private – metoda oże być wywołana wewnątrz tej samej klasy
* Protected – metoda może być wywołana z z tej samej klasy, podklas i tego samego pakietu
* Default – metoda może być wywołana z pakietu.

**Dostępu default nie deklarujemy jawnie.** Istnieje w Javie słowo kluczowe default, ale jest używane tylko w wyrażeniu switch oraz przy definiowaniu metod domyślnych w interfejsach. Nie używa się go jako modyfikatora dostępu. Domyślny poziom dostępu występuje gdy poziom dostępu nie jet jawnie zadeklarowany za pomocą modyfikatora dostępu(private, protected, default).

Default to inaczej **Package private.**

Przyjrzyj się modyfikatorowi **protected. On nie jest taki oczywisty.**

### Opcjonalne specyfikatory

\istnieje wiele opcjonalnych specyfikatorów, ale większość z inch nie jest wymagane na egzaminie OCA. Pecyfikatory opcjonalne mogą znajdować się przed modyfikatorem dostępu, ale **nie mogą być po typie zwracanym.**

Na egzaminie:

* Static
* Abstract
* Final

Nie ma na egzaminie:

* Synchronized
* Native
* Strictfp

### Typ zwracany

Deklaracja metody **zawsze** musi mieć typ zwracany. Jeśli metoda nic nie zwraca, należy użyć typu **void.** Pamiętaj, aby zwracać uwagę czy metoda posiada wyrażenie **return,** ,jeśli typ zwracany jest inny niż **void. Metody z typem zwracanym void nie muszą, ale mogą bieć wyrażenie return.**

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.

#### Wywoływanie metod statycznych i zmiennych

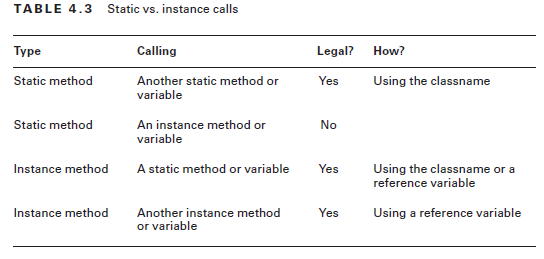
Zmienne i metody statyczne można wywołać na zmiennej do, której przypisany jest null!. Nie powoduje to **NullPointerException**.

Dostęp do zmiennych i metod statycznych odbywa się poprzez nazwę klasy lub referencję klasy. Będąc wewnątrz klasy nie trzeba używać jej nazwy w celu odwołania się do zmiennej statycznej.

### Static vs Instance

Metoda statyczna nie ma dostępu do niestatycznych zmiennych i metod klasy.

Zarówno metody statyczne, jak i niestatyczne mogą wywoływać metody statyczne i mieć dostęp do zmiennych statycznych. Podobnie pola klasy mogą korzystać ze zmiennych statycznych, ale nie odwrotnie.



### Statyczne bloki inicjujące

Finalne zmienne statyczne mogą być zainicjowane **tylko** przy deklaracji lub w statycznych blokach inicjujących. Co więcej zmienne te muszą zostać zainicjowane 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.

## Przekazywanie danych do metod

Java is a “pass-by-value” language.

## Przeciążanie metod

Przeciążanie metody ma miejsce kiedy istnieją różne metody o tej same nazwie, ale innych parametrach. Nie możliwe jest utworzenie w jednej klasie dwóch metod o takiej samej nazwie i parametrach, lub metod o takiej samej nazwie i parametrach, ale różniących się tylko typem.

Przeciążane metody mogą się różnić wszystkim innym poza nazwą: zwracanym typem, modyfikatorem dostępu. Specyfikatorem (np. static) i listą wyjątków. Po prostu każda musi mieć inny zestaw przyjmowanych parametrów.

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 tylko varargs!

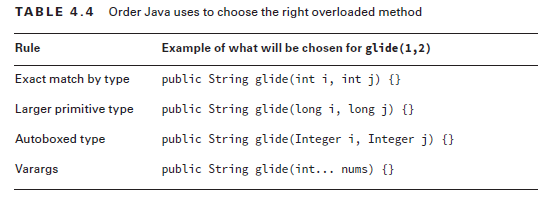
Mogę istnieć metody o tej samej nazwie – jedna z argumentem typu **int,** a druga **Integer.**

**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.)

### Konstruktor domyślny

Każda klasa posiada konstruktor, niezlaeżnie czy go napiszesz czy nie. Jeśli utworzysz żadnego konstruktora, Java stworzy konstruktor bez parametrowy. W pliku z rozszerzeniem **.java** nie znajdziesz konsrtuktora domyślnego, Java dodaje go podczas kompilacji, więc znajdzie się w pliku **.class.**

### Przeciązanie konstruktorów

Aby wywołać konstruktor z innego konstruktora należy użyć słówka **this.**

**public** Constructors2(String name) {

**this**(name, 2);

}

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

### Pola final

Pole final musi zainicjalizowane w jednym z trzech miejsc:

* Przy deklaracji
* Blok inicjalizacyjny
* Konstruktor

Przed zakończeniem pracy konstruktora, wszystkie zmienne finalne muszą zostać zainicjowane. Nie zostaną one zainicjowane domyślne, tak jak zwykłe zmienne klasowe.

### Kolejność inicjalizacji

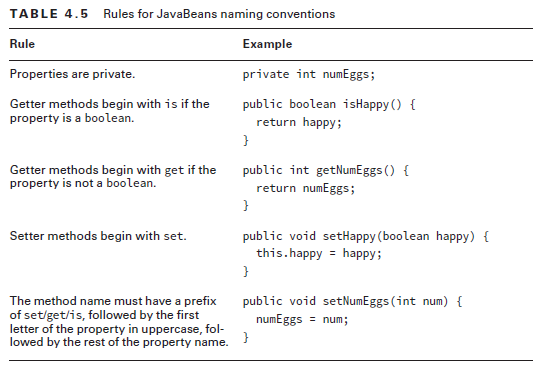
* Jeśli jest nadklasa, zostanie zainicjowana pierwsza
* Statyczne zmienne i statyczne bloki inizjalizujące w kolejności występowania
* Zmienne klasowe i bloki inicjalizacyjne w kolejności występowania
* Konstruktor

Wydaje mi się, że statyczne elementy zostaną zainicjowane jeszcze przed stworzeniem jakiegokolwiek obiektu.

## 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

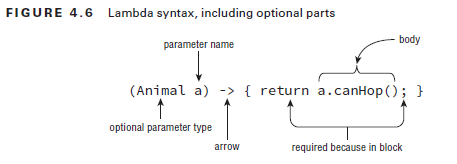
## Lambda

Java jest językiem obiektowym. W wersji 8 dodano wyrażenia lambda, które pozwalają pisać kod innym sposobem – **programowanie funkcyjne.** Możesz myśleć o wyrażeniach lambda jak o anonimowych metodach. Mają parametry i ciało, jak normalna metoda, ale nie mają nazwy.

Innymi słowy, wyrażenia lmbda są jak metody, które możesz przekazać jakby były zmienną.

Na egzaminie są tylko najprostsze wyrażenia lambda.

### 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.

# Projektowanie klas

## 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**

### Tworzenie obiektów

Każda klasa dziedziczy po **java.lang.Object**, Jeśli klasa nie rozszerza żadnej innej klasy, kompilator automatycznie dodaje *extends java.lang.Object.* Można też dodać taki zapis samemu do kodu.

### 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

**public** **class** Donkey {

}

**public** **class** Donkey {

**public** Donkey() {

}

}

**public** **class** Donkey {

**public** Donkey() {

**super**();

}

}

### 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.

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 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.

### Metody finalne

Metody finalne nie mogą być nadpisywane.

Nie można zakrywać statycznych metod finalnych

## Dziedziczenie zmiennych

W Javie nie można dziedziczyć zmiennych, ale można je zakrywać.

### Zakrywanie zmiennych

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.

## Tworzenie klas abstrakcyjnych

Klasa abstrakcyjna to klasa oznaczona słowem **abstract.** Nie można tworzyć obiektów takiej klasy.

Metody abstrakcyjne również oznaczane są słowem **abstract.** Mogą znajdować się tylko w klasach abstrakcyjnych

### 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. Zmienne mogą być prywatne, statyczne i finalne.

Metody abstrakcyjne nie mogą być static i final. 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.

### Tworzenie klas konkretnych

Podklasa musi implementować wszystkie metody abstrakcyjne nadklasy. (chyba, że sama też jest abstrakcyjna).

Klasa abstrakcyjna może rozszerzać inną klasę abstrakcyjną i nie musi implementować jej metod abstrakcyjnych. Musi to zrobić (najpóźniej) pierwsza klasa konkretna.

Klasa abstrakcyjna może posiadać konstruktor.

## Interfejsy

Klasa i interfejs mogą implementować dowolną liczbe interfejsów.

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.

### Dziedziczenie interfejsów

Jeśli interfejs rozszerza inny interfejs używamy słowa **extends** (nie implements). Interfejs dziedziczy wszystkie metody abstrakcyjne po interfejsie, który rozszerza. Podobnie klasa abstrakcyjna dziedziczy wszystkie metody abstrakcyjne z interfejsu, który imlementuje i nie musi tych metod implementować.

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

### Zmienne interfejsu

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

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

### 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 zostały dodane w Java 8 z dwóch powodów: dają nowe możliwości (lambda, interfejsy funkcyjne) oraz zapewniają wsteczną kompatybilność.

**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

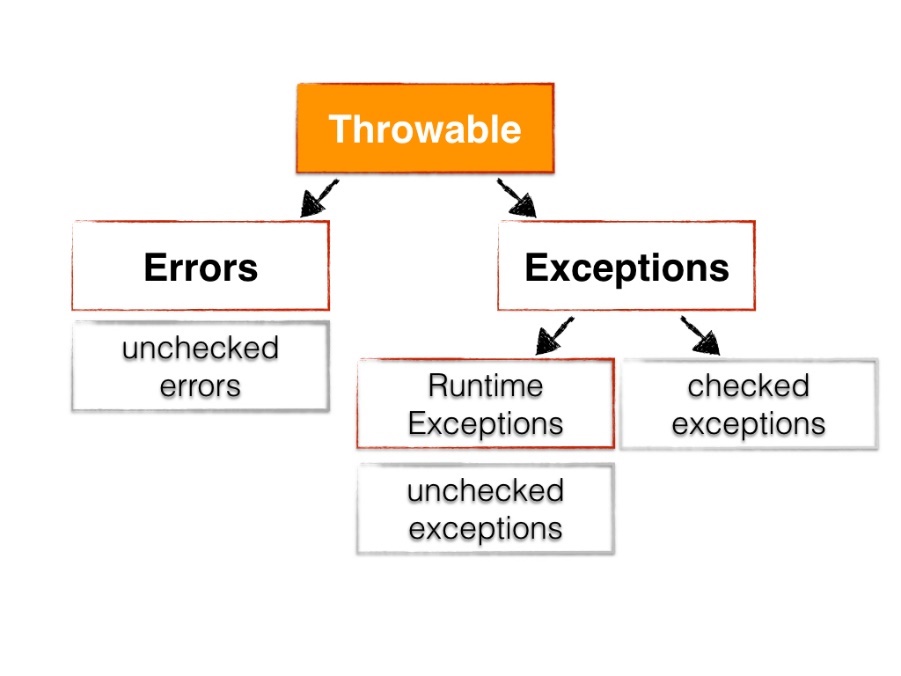
Jest tylko jedna różnica między metodą statyczną w klasie i interfejsie. **Statyczne metody zdefiniowane w interfejsie nie są dziedziczone do klas, które implementują interfejs.**

297

# Wyjątki

## Zrozumienie wyjątków

Istnieją dwa podejścia w używaniu wyjątków. Metoda może sama obsługiwać przypadki wystąpienia wyjątku lub przenieść odpowiedzialność na wywołującego metodę.



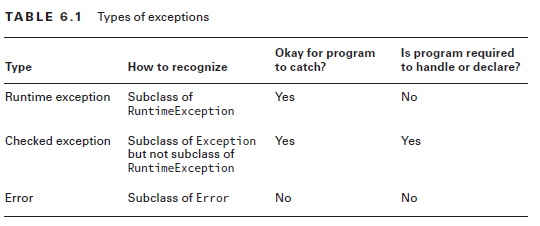
* Unchecked exception: Errors i Runime Exceptions
* Checked exception: Wszystkie pozostałe (w tym throwable i Exceptions)

Kiedy widzisz nazwę runtime exception odnosi się to do wyjątków unchecked. Jest to nieco mylące bo przecież wszystkie wyjątki pojawiąją się w runtime (czasie wykonania programu).

Wsystkie checked exceptions należy obsłużyć albo zadeklarować w definicji metody.

Wszystkie Klasy rozszerzające Throwable w tym Error mogą być zadeklarowane w części **throws** metody oraz mogą być przechwytywane.

Wyjątki mogą być typem zwrotnym metody.



## Wyrażenie *try*

Wyrażenie try oraz catch, nawet z jedną linią kodu, wymaga nawiasów klamrowych (w przeciwieństwie do np. ***if()***).

Po wyrażeniu ***try*** wymagany jest wystąpienie bloku ***catch***  i/lub ***finally.***

***Finally***  zostanie wykonane niezależnie od tego czy wyjątek zostanie rzucony czy nie.

Jeśli wyjątek wystąpi,  ***finally***  zostanie wykonane po bloku  ***catch.***

Istnieje też **try-wth-resources**, w przypadku, którego nie jest wymagane użycie ani **catch ani finally**. Nie obowiązuje on jednak na egzaminie OCA.

Jeśli w **try albo catch** zostanie wywołana metoda **System.exit()**, blok **finally**  nie zostanie wykonany.

Gdy zostanie rzucony wyjątek Java przegląda bloki catch w kolejności, w której są zapisane. Pierwszy blok catch, który pasuje do typu rzuconego wyjątku, zostanie wykonany. Pozostałe bloki nie zostaną wykonane. Jeśli blok **catch jest niemożliwy do wykonania,** kompilator poinformuje o błędzie. Taka sytuacja ma miejsce gdy blok catch deklaruje nadklasę, a inny blok catch, pod nim, podklasę.

Zarówno w catch jak i finally mogą znajdować się zagnieżdżone bloki try.

Jeśli finally rzuca wyjątek przykrywa on wyjątki rzucone wcześniej w try i catch.

Żeby przechwycić Unchecked exception w **catch ,** metoda wywoływana w **try**  musi deklarować w **throws**  ten wyjątek. W innym przypadku będzie błąd kompilacji **Unreachable code.** Jeśli metoda deklaruje unchecked exception nie ma obowiązku go przechwytywać

## Popularne typy wyjątków

### Runtime Exceptions

* ArithmeticException – rzucany gdy wykonamy niedozwoloną operacje matematyczną np. dzielimy przez zero. Co ciekawe gdy podzielimy wartość zmienno-przecinkową przez zero nie jest rzucany wyjątek, a **Infinity. Rzucany przez JVM**
* ArrayIndexOutOfBoundsException – Użycie nieistniejącego indeksu tablicy. **Rzucany przez JVM**
* ClassCastException – Rzutowanie obiektu do typu podklasy, której nie jest instancją.
* IllegalArgumentException – Rzucany przez programistę by wskazać, że do metody został przekazany nieodpowiedni argument. **Rzucany przez JVM**
* NullPointerException – referencja przechowuje null, w sytuacji gdy wymagany jest konkretny obiekt. **Rzucany przez JVM**
* NumberFormatException – Rzucany przez programistę kiedy podejmowana jest próba konwersji Stringa na typ numeryczny, ale String ma nieodpowiedni format. Jest podklasą IllegalArgumentException.

### Checked Exceptions

Mogą być rzucane przez JVM albo przez programistę.

* FileNotFoundException – Podczas próby odniesienia się do pliku, który nie istnieje. **Rzucany przez programistę**
* IOException – Gdy występuje problem z odczytem lub zapisem do pliku. **Rzucany przez programistę.** Jest rzucany przez wiele meto w pakiecie java.io, ale zawsze rzucane są w sposób zaprogramowany.

Na egzamin potrzebuje tylko wiedzieć, że te dwa są checked exceptions. Pierwszy jest podklasą drugiego.

### Errors

**Wszystkie rzucane przez JVM.**

* ExceptionInInitializerError – rzucany gdy podzas inicjalizacji statycznych elementów klasy rzucany zostanie wyjątek, który nie zostanie obsłużony. Czyli de facto wiele innych wyjątków wypisanych w tym rozdziale może go spowodować. Wystarczy, żeby wystąpił naprzykład podczas wykonania bloku statycznego. W konsoli wraz z tym wyjątkiem pokazywana jest również informacja o pierwotnym wyjątku.
* StackOverflowError – Gdy metoda wykonuje siebie samą bez końca. Nieskończona pętla go nie powoduje.
* NoClassDefFoundError – Gdy klasa, która jest użyta w kodzie jest dostępna podczas kompilacji, ale nie podczas czasu wykonania aplikacji.

## Wywoływanie metod, które rzucają wyjątki

Metoda może deklarować że rzuca wyjątek, mimo, że wcale go nie rzuca.

Jeśli metoda deklaruje checked exception to aby ją wywołać należy albo go obsłużyć (**try - catch)**, albo zadeklarować.

Jeśli metoda deklaruje unchecked exception nie trzeba z tym nic robić przy wywołaniu tej metody.

Metoda main może deklarować wyjątki.

### Podklasy

Metoda podklasy może deklarować takie same lub mniej wyjątków co metoda nadklasy lub interfejsu. Nie może deklarować wyjątków szerszych niż nadklasa czy metoda interfejsu.

Wyjątek stanowią checked expressions, ponieważ ich deklaracja nic nie zmienia.

### Wyświetlanie wyjątków

Istnieją trzy sposoby wyświetlenia wyjątku:

System.***out***.println(e);

System.***out***.println(e.getMessage());

e.printStackTrace();

# Inne

Nazwy (zmiennych, klas, method itp. ) muszą zaczynać się od litery, podkreślenia (\_) lub znaku dolara $. Każdy kolejny znak może dodatkowo być cyfrą. Wielkie litery są rozróżniane od małych. Nie wolno używać jako nazw zarezerwowanych słów Java.

**public** **void** $a8$() {} – poprawna nazwa metody

Słowa kluczowe Javy (najbardziej problematyczne):

* const
* case
* default
* goto
* native
* strictfp
* transient
* volatile

Jeśli coś jest wymagane (reuired) , to jest z automatu możliwe (allowed)

email: [joanna.gawron@opsenio.com](mailto:joanna.gawron@opsenio.com)  
hasło: opseniowo987

VPN jest jakby tunelem poprowadzonym przez internet, pomiędzy routerem do którego nie jestem bezpośrednio podpięty, a moim komputerem.

Czyli VPN to tak jakby połączenie z innym routerem za pośrednictwem internetu.