

**PRZYGOTOWANIE DO CERTYFIKACJI**

# **SCJP 6**



**MARIUSZ LIPÍŃSKI**

**PRZYGOTOWANIE DO CERTYFIKACJI**

# **SCJP 6**

**wydanie drugie**

© Copyright by Mariusz Lipiński, Warszawa 2009

kontakt: [mariusz.lipinski@getcjp.pl](mailto:mariusz.lipinski@getcjp.pl)

[www.getscjp.pl](http://www.getscjp.pl)

**ISBN 978-83-929398-1-8**

**Książkę dedykuję mojej ukochanej Magdzie,  
która od początku we mnie wierzyła i wspierała**



# SPIS TREŚCI

<b>WSTĘP</b> .....	<b>9</b>
<b>PODSTAWOWE INFORMACJE O SCJP</b> .....	<b>11</b>
<b>ZAKRES EGZAMINU SCJP</b> .....	<b>13</b>
<b>KLASY, INTERFEJSY, TYPY WYLICZENIOWE</b> .....	<b>23</b>
ORGANIZACJA PLIKU KODU ŹRÓDŁOWEGO .....	24
PAKIETY .....	25
INSTRUKCJA IMPORTU .....	25
MODYFIKATORY DLA DEKLARACJI KLAS .....	28
DEKLARACJA INTERFEJSU .....	30
IMPLEMENTACJA INTERFEJSU .....	31
POPRAWNE IDENTYFIKATORY I KONWENCJE NAZEWNICZE.....	31
KLASY WEWNĘTRZNE.....	33
KLASY LOKALNE METODY.....	37
KLASY ANONIMOWE.....	40
STATYCZNE KLASY ZAGNIEŹDŻONE .....	42
TYPY WYLICZENIOWE .....	45
<b>METODY, KONSTRUKTORY I ZMIENNE</b> .....	<b>49</b>
DEKLARACJA ZMIENNYCH I STAŁYCH .....	50
WARTOŚCI DOMYŚLNE ZMIENNYCH .....	52
LITERAŁY .....	54
OBIEKTY TYPU TABLICOWEGO .....	56
INICJALIZACJA TABLIC .....	58
ZMIENNE TYPÓW PROSTYCH I REFERENCYJNYCH .....	60
PARAMETRY METOD.....	62
OPERATORY PRZYPISANIA DLA TYPÓW PROSTYCH .....	64
OPERATORY ARYTMETYCZNE, LOGICZNE I RELACYJNE .....	67
ZAKRES WIDOCZNOŚCI ZMIENNYCH .....	68
MODYFIKATORY WIDOCZNOŚCI W DEKLARACJI METOD I ZMIENNYCH .....	69
DEKLARACJE METOD.....	72
PRZESŁANIANIE METOD.....	74
PRZECIĄŻANIE METOD.....	77
REDEFINIOWANIE METOD STATYCZNYCH.....	80
DEKLARACJA KONSTRUKTORA.....	81
KONSTRUKTORY I INICJALIZACJA.....	83
BLOKI INICJALIZACYJNE.....	86
IN-BOXING I OUT-BOXING.....	88
<b>ZAGADNIENIA PROGRAMOWANIA OBIEKTOWEGO</b> .....	<b>91</b>
HERMETYZACJA, ZALEŻNOŚĆ, SPÓJNOŚĆ .....	92
WIELODZIEDZICZENIE .....	94
POLIMORFIZM.....	94
RZUTOWANIE REFERENCJI.....	97

ZWIĄZKI TYPU IS-A ORAZ HAS-A .....	99
<b>PĘTLE, ITERATORY I INSTRUKCJE WARUNKOWE .....</b>	<b>101</b>
INSTRUKCJA WARUNKOWA IF .....	102
INSTRUKCJA WYBORU SWITCH .....	103
PĘTLE WHILE I DO-WHILE .....	106
PĘTLA FOR.....	107
INSTRUKCJE BREAK I CONTINUE .....	108
PĘTLA FOR-EACH.....	110
<b>WYJĄTKI.....</b>	<b>113</b>
RZUCANIE I OBSŁUGA WYJĄTKÓW.....	114
POPULARNE TYPY WYJĄTKÓW .....	118
<b>API.....</b>	<b>121</b>
KLASY OPAKOWUJĄCE TYPÓW PROSTYCH.....	122
KLASA STRING .....	125
KLASY STRINGBUFFER I STRINGBUILDER.....	129
KLASY PAKIETU JAVA IO .....	131
OPERACJE NA PLIKACH.....	135
DATY I CZAS.....	143
FORMATOWANIE I PARSOWANIE LICZB I WARTOŚCI WALUTOWYCH .....	148
WYSZUKIWANIE WZORCA W TEKŚCIE.....	150
TOKENIZACJA TEKSTU .....	153
<b>KOLEKCJE I TYPY GENERYCZNE .....</b>	<b>155</b>
METODY EQUALS() I HASHCODE().....	156
STRUKTURY DANYCH .....	158
MAPY .....	159
KOLEKCJE .....	166
SORTOWANIE LIST I TABLIC ORAZ WYSZUKIWANIE BINARNE .....	172
TYPY GENERYCZNE .....	176
TYPY GENERYCZNE A POLIMORFIZM .....	180
DEKLARACJE GENERYCZNE.....	184
<b>WĄTKI.....</b>	<b>187</b>
TWORZENIE I URUCHAMIANIE WĄTKÓW .....	188
SYNCHRONIZACJA WĄTKÓW .....	191
STANY WĄTKÓW .....	195
PRIORYTETY WĄTKÓW .....	198
<b>KOMPILATOR ORAZ MASZYNA WIRTUALNA .....</b>	<b>199</b>
ASERCJE.....	200
MECHANIZM ODZYSKIWANIA PAMIĘCI.....	202
KOMPILACJA PROGRAMU .....	204
URUCHAMIANIE PROGRAMU.....	206
MECHANIZM WYSZUKIWANIA KLAS .....	208
ARCHIWA JAR .....	210

## WSTĘP

Celem niniejszej książki jest przygotowanie czytelnika do egzaminu na certyfikat Sun Certified Programmer for the Java Platform, Standard Edition 6 (CX-310-065) skrótowo nazywanego SCJP (skr. Sun Certified Java Programmer).

Książka koncentruje się na przygotowaniu do egzaminu, jednak przygotowanie to sprowadza się do nauki języka – polega na usystematyzowaniu wiedzy i zgłębieniu tajników Javy. Przedstawiono więc zagadnienia związane z programowaniem w języku Java, tyle, że położono szczególny nacisk na omówienie tematów eksploatowanych na egzaminie oraz zwrócono uwagę czytelnika za szczegóły, które normalnie mogłyby umknąć jego uwadze a są istotne z punktu widzenia adepta SCJP.

Zakłada się, że czytelnik posiada umiejętność programowania w języku Java na poziomie podstawowym. Nie zaleca się, aby do przygotowania do certyfikacji przystępowały osoby nieposiadające minimum wiedzy i doświadczenia programistycznego.



## PODSTAWOWE INFORMACJE O SCJP

Certyfikat SCJP – z założenia – ma być potwierdzeniem wysokiego poziomu umiejętności i profesjonalizmu w zakresie programowania w języku Java. Aby uzyskać ten certyfikat kandydat musi zdać egzamin w formie testu wielokrotnego wyboru (choć niektóre pytania mogą wymagać np. poukładania fragmentów kodu, nie tylko wyboru opcji A, B, C, D...). Egzamin składa się z 60 pytań; limit czasowy to 180 minut. Aby zaliczyć trzeba uzyskać co najmniej 58,33% punktów, tak więc należy w pełni poprawnie odpowiedzieć na co najmniej 35 pytań.

By móc przystąpić do egzaminu nie trzeba posiadać żadnych innych certyfikatów, tj. nie ma żadnych warunków wstępnych dopuszczenia do egzaminu. Posiadanie certyfikatu SCJP jest natomiast warunkiem wstępnym do przystąpienia do egzaminów dla certyfikatów specjalistycznych, takich jak Sun Certified Web Component Developer (SCWCD) czy Sun Certified Developer For Java Web Services (SCDJWS).



# ZAKRES EGZAMINU SCJP

Zakres materiału jaki obowiązuje na egzaminie – wiedza którą trzeba przed nim osiąść – są dość precyzyjnie określone. Firma Sun Microsystems, wystawca certyfikatu SCJP, publikuje te wymagania na swych stronach internetowych. Poniżej przedstawiono je w formie oryginalnej – tj. po angielsku, dla zachowania pełnej precyzji – oraz dodatkowo, pod każdym z punktów podano polskie tłumaczenie.

## 1. Deklaracje

*1.1. Develop code that declares classes (including abstract and all forms of nested classes), interfaces, and enums, and includes the appropriate use of package and import statements (including static imports).*

Napisz kod, w którym deklarujesz klasy (wliczając klasy abstrakcyjne i wszelkie formy klas zagnieżdżonych), interfejsy i typy wyliczeniowe oraz zademonstruj poprawne użycie instrukcji `package` i `import` (w tym `import` statyczny).

*1.2. Develop code that declares an interface. Develop code that implements or extends one or more interfaces. Develop code that declares an abstract class. Develop code that extends an abstract class.*

Napisz kod, w którym deklarujesz interfejs oraz kod, w którym implementujesz lub dziedziczysz z jednego lub wielu interfejsów. Zaimplementuj klasę abstrakcyjną oraz klasę, która dziedziczy z klasy abstrakcyjnej.

*1.3. Develop code that declares, initializes, and uses primitives, arrays, enums, and objects as static, instance, and local variables. Also, use legal identifiers for variable names.*

Napisz kod, w którym deklarujesz, inicjalizujesz oraz używasz zmiennych statycznych, instancyjnych oraz lokalnych typów prostych, tablicowych, wyliczeniowych oraz obiektowych. Użyj poprawnych identyfikatorów dla nazw zmiennych.

- 1.4. *Given a code example, determine if a method is correctly overriding or overloading another method, and identify legal return values (including covariant returns), for the method.*

Mając podany fragment kodu, oceń czy metoda poprawnie przysłania lub przeciąża inną metodę oraz wskaż, jakie są poprawne wartości zwracane przez tę metodę (uwzględniając kowariantność).

- 1.5. *Given a set of classes and superclasses, develop constructors for one or more of the classes. Given a class declaration, determine if a default constructor will be created, and if so, determine the behavior of that constructor. Given a nested or non-nested class listing, write code to instantiate the class.*

Mając podany zestaw klas i nadklas zaimplementuj konstruktory dla jednej lub kilku z pośród nich. Mając podany kod klasy, oceń czy zostanie dla niej wygenerowany konstruktor domyślny i jeśli tak, określ jakie będzie działanie tego konstruktora. Mając podaną listę klas – niezagnieżdżonych i zagnieżdżonych – napisz kod który tworzy instancje tych klas.

## 2. Kontrola sterowania

- 2.1. *Develop code that implements an if or switch statement; and identify legal argument types for these statements.*

Napisz kod, w którym poprawnie używasz instrukcji `if` i `switch` oraz wskaż, jakie typy argumentów mogą być użyte dla tych instrukcji.

- 2.2. *Develop code that implements all forms of loops and iterators, including the use of for, the enhanced for loop (for-each), do, while, labels, break, and continue; and explain the values taken by loop counter variables during and after loop execution.*

Napisz kod, w którym implementujesz wszelkie rodzaje pętli i iteracji, włączając pętle `for` w formie podstawowej i uproszczonej (nowej), pętle `do` i `while` oraz instrukcje `continue` i `break` (także wariant z etykietami). Wy tłumacz, w jaki sposób zmienia się wartość licznika pętli w trakcie wykonania i jaka jest wartość tego licznika po wykonaniu pętli.

- 2.3. *Develop code that makes use of assertions, and distinguish appropriate from inappropriate uses of assertions.*

Napisz kod, w którym używasz asercji, oraz wskaż w jakich sytuacjach i w jaki sposób mechanizm ten powinien być używany a w jakich nie.

- 2.4. *Develop code that makes use of exceptions and exception handling clauses (try, catch, finally), and declares methods and overriding methods that throw exceptions.*

Napisz kod, w którym używasz wyjątków i zaimplementuj kod obsługi wyjątków (try, catch, finally) oraz zadeklaruj metody i metody przysłaniające, które rzucają wyjątki.

- 2.5. *Recognize the effect of an exception arising at a specified point in a code fragment. Note that the exception may be a runtime exception, a checked exception, or an error.*

Określ, jakie są efekty rzucenia wyjątku przez wskazany fragment danego programu. Wyjątek ten może być wyjątkiem kontrolowanym (tj. typu „checked”) jak i niekontrolowanym (typu „un-checked”).

- 2.6. *Recognize situations that will result in any of the following being thrown: `ArrayIndexOutOfBoundsException`, `ClassCastException`, `IllegalArgumentException`, `IllegalStateException`, `StackOverflowError`, `NullPointerException`, `AssertionError`, `ExceptionInInitializerError`, `NumberFormatException` or `NoClassDefFoundError`. Understand which of these are thrown by the virtual machine and recognize situations in which others should be thrown programmatically.*

Określ, jakie sytuacje mogą spowodować rzucenie wyjątków: `ArrayIndexOutOfBoundsException`, `AssertionError`, `IllegalStateException`, `IllegalArgumentException`, `NumberFormatException`, `NullPointerException`, `ExceptionInInitializerError`, `StackOverflowError`, `ClassCastException` oraz `NoClassDefFoundError`. Wskaż, które z nich są rzucane przez Wirtualną Maszynę Javy (ang. akr. JVM) a które – i w jakich okolicznościach – powinny być i są rzucane przez programistę.

### 3. API

- 3.1. *Develop code that uses the primitive wrapper classes (such as `Boolean`, `Character`, `Double`, `Integer`, etc.), and/or autoboxing & unboxing. Discuss the differences between the `String`, `StringBuilder`, and `StringBuffer` classes.*

Napisz kod, w którym używasz klas opakowujących typów prostych (np. Boolean, Character, Double, Integer) oraz wykorzystujesz funkcjonalność „auto-boxingu” i „un-boxingu”. Wskaż różnice między klasami String, StringBuilder oraz StringBuffer.

- 3.2. *Given a scenario involving navigating file systems, reading from files, writing to files, or interacting with the user, develop the correct solution using the following classes (sometimes in combination), from java.io: BufferedReader, BufferedWriter, File, FileReader, FileWriter, PrintWriter, and Console.*

Mając podany scenariusz pracy z systemem plików, czytania z plików, zapisu do plików oraz interakcji z użytkownikiem zaimplementuj rozwiązanie używając następujących klas z pakietu java.io: BufferedReader, BufferedWriter, File, FileReader, FileWriter, PrintWriter oraz Console.

- 3.3. *Use standard J2SE APIs in the java.text package to correctly format or parse dates, numbers, and currency values for a specific locale; and, given a scenario, determine the appropriate methods to use if you want to use the default locale or a specific locale. Describe the purpose and use of the java.util.Locale class.*

Używając klas standardu Java SE z pakietu java.text napisz kod, w którym formatujesz oraz parsujesz daty, liczby i wartości walutowe z uwzględnieniem lokalizacji. Mając podany scenariusz, wskaż metody których należy użyć aby uwzględnić konkretną albo domyślną lokalizację. Opisz przeznaczenie oraz sposób użycia klasy java.util.Locale.

- 3.4. *Write code that uses standard J2SE APIs in the java.util and java.util.regex packages to format or parse strings or streams. For strings, write code that uses the Pattern and Matcher classes and the String.split method. Recognize and use regular expression patterns for matching (limited to: . (dot), \* (star), + (plus), ?, \d, \s, \w, [], ()). The use of \*, +, and ? will be limited to greedy quantifiers, and the parenthesis operator will only be used as a grouping mechanism, not for capturing content during matching. For streams, write code using the Formatter and Scanner classes and the PrintWriter.format/printf methods. Recognize and use formatting parameters (limited to: %b, %c, %d, %f, %s) in format strings.*

Używając klas standardu Java SE z pakietu `java.util` oraz `java.util.regex` napisz kod, w którym formatujesz oraz parsujesz stringi lub strumienie. Napisz kod w którym używasz klas `Pattern` i `Matcher` oraz operacji `split(...)` z klasy `String`; użyj wyrażeń regularnych (ograniczone do elementów `.` (kropka), `*`, `+`, `?`, `\d`, `\s`, `\w`, `[]` oraz `()`). Elementy `*`, `+` oraz `?` musisz umieć zastosować tylko jako operatory zachłanne a nawiasy jako elementy grupujące – nie jako mechanizm pobierania dopasowanych fragmentów tekstu. Dla operacji na strumieniach napisz kod który używa klas `Formatter` i `Scanner` oraz operacji `printf(...)` i `format(...)` z klasy `PrintWriter`. Użyj odpowiednich parametrów formatujących tekst (ograniczone do parametrów `%b`, `%c`, `%d`, `%f` oraz `%s`).

#### 4. Wątki

4.1. *Write code to define, instantiate, and start new threads using both `java.lang.Thread` and `java.lang.Runnable`.*

Napisz kod, w którym definiujesz, tworzysz oraz uruchamiasz wątki z użyciem klas `java.lang.Thread` oraz `java.lang.Runnable`.

4.2. *Recognize the states in which a thread can exist, and identify ways in which a thread can transition from one state to another.*

Opisz stany w jakich wątek może się znajdować oraz opisz sytuacje, w wyniku których następują przejścia pomiędzy tymi stanami.

4.3. *Given a scenario, write code that makes appropriate use of object locking to protect static or instance variables from concurrent access problems.*

Mając podany scenariusz napisz kod, w którym w poprawny sposób używasz mechanizmu monitorów obiektów dla chronienia zmiennych statycznych i instancyjnych przed problemami przetwarzania współbieżnego.

#### 5. Obiektowość

5.1. *Develop code that implements tight encapsulation, loose coupling, and high cohesion in classes, and describe the benefits.*

Napisz kod zgodnie z pryncypiami ścisłej hermetyzacji, prostych zależności i dużej spójności oraz opisz zalety takiego kodu.

- 5.2. *Given a scenario, develop code that demonstrates the use of polymorphism. Further, determine when casting will be necessary and recognize compiler vs. runtime errors related to object reference casting.*

Mając podany scenariusz napisz kod w którym demonstrujesz zrozumienie polimorfizmu. Oceń, kiedy konieczne jest zastosowanie rzutowania oraz opisz błędy jakie mogą powstać w związku z rzutowaniem. Wskaż, które z tych błędów są błędami wykonania a które błędami kompilacji.

- 5.3. *Explain the effect of modifiers on inheritance with respect to constructors, instance or static variables, and instance or static methods.*

Opisz jaki wpływ na dziedziczenie ma zastosowanie poszczególnych modyfikatorów dostępu w stosunku do konstruktorów oraz zmiennych i metod statycznych i instancyjnych.

- 5.4. *Given a scenario, develop code that declares and/or invokes overridden or overloaded methods and code that declares and/or invokes superclass, or overloaded constructors.*

Mając podany scenariusz napisz kod, w którym deklarujesz i wywołujesz przysłonięte lub przeciążone metody oraz kod, w którym deklarujesz i wywołujesz konstruktor z nadklasy oraz konstruktor przeciążony.

- 5.5. *Develop code that implements "is-a" and/or "has-a" relationships.*

Napisz kod, w którym występują relacje IS-A oraz HAS-A. Wyjaśnij, na czym polegają te relacje.

## 6. Kolekcje i typy generyczne

- 6.1. *Given a design scenario, determine which collection classes and/or interfaces should be used to properly implement that design, including the use of the Comparable interface.*

Mając podany projekt rozwiązania oceń, które klasy i interfejsy kolekcji (włączając interfejs Comparable) powinny być użyte by prawidłowo zaimplementować ten projekt.

- 6.2. *Distinguish between correct and incorrect overrides of corresponding hashCode and equals methods, and explain the difference between == and the equals method.*

Wskaż, które z pośród zaprezentowanych implementacji metod hashCode() i equals(...) są poprawne oraz wyjaśnij na czym polega różnica między metodą equals(...) a operatorem ==.

- 6.3. *Write code that uses the generic versions of the Collections API, in particular, the Set, List, and Map interfaces and implementation classes. Recognize the limitations of the non-generic Collections API and how to refactor code to use the generic versions. Write code that uses the NavigableSet and NavigableMap interfaces.*

Napisz kod, w którym używasz generycznych wersji klas i interfejsów kolekcji, w szczególności interfejsów Set, List oraz Map i ich klas implementujących. Określ, jakie są ograniczenia wersji niegenerycznych oraz w jaki sposób zrefaktoryzować kod używający tych wersji tak, aby użyć wersji generycznych. Napisz kod, w którym użyjesz interfejsów NavigableSet oraz NavigableMap.

- 6.4. *Develop code that makes proper use of type parameters in class/interface declarations, instance variables, method arguments, and return types; and write generic methods or methods that make use of wildcard types and understand the similarities and differences between these two approaches.*

Napisz kod, w którym używasz typów parametryzowanych w deklaracjach klas, interfejsów, zmiennych instancyjnych oraz argumentów i typów zwracanych metod. Zaimplementuj metody generyczne oraz metody które używają typów niedookreślonych i wytłumacz na czym polegają podobieństwa i różnice między tymi podejściami.

- 6.5. *Use capabilities in the java.util package to write code to manipulate a list by sorting, performing a binary search, or converting the list to an array. Use capabilities in the java.util package to write code to manipulate an array by sorting, performing a binary search, or converting the array to a list. Use the java.util.Comparator and java.lang.Comparable interfaces to affect the sorting of lists and arrays. Furthermore, recognize the effect of the "natural ordering" of primitive wrapper classes and java.lang.String on sorting.*

Używając klas i interfejsów z pakietu `java.util` napisz kod, w którym sortujesz oraz stosujesz wyszukiwanie binarne dla list i tablic a także konwertujesz listy na tablice i – na odwrót – tablice na listy. Użyj interfejsów `java.util.Comparator` oraz `java.lang.Comparable` aby zmienić porządek sortowania elementów list i tablic. Określ, co oznacza i jaki jest „porządek naturalny” dla klas opakowujących typów prostych i klasy `java.lang.String`.

## 7. Podstawy

7.1. *Given a code example and a scenario, write code that uses the appropriate access modifiers, package declarations, and import statements to interact with (through access or inheritance) the code in the example.*

Mając podany fragment kodu oraz scenariusz napisz program, w którym używasz odpowiednich modyfikatorów dostępu, prawidłowo deklarujesz pakiet oraz używasz instrukcji importu, w którym używasz podanego fragmentu kodu poprzez dziedziczenie lub kompozycje.

7.2. *Given an example of a class and a command-line, determine the expected runtime behavior.*

Mając podany kod klasy lub skompilowaną klasę oraz instrukcję linii poleceń oceń, jaki efekt będzie miało wykonanie tej instrukcji.

7.3. *Determine the effect upon object references and primitive values when they are passed into methods that perform assignments or other modifying operations on the parameters.*

Opisz efekt operacji przypisania i innych operacji modyfikujących wykonanych na zmiennych referencyjnych i prostych przekazanych jako parametry wywołania metody.

7.4. *Given a code example, recognize the point at which an object becomes eligible for garbage collection, determine what is and is not guaranteed by the garbage collection system, and recognize the behaviors of the `Object.finalize()` method.*

Mając podany przykład kodu oceń, kiedy obiekt staje się dostępny dla mechanizmu odzyskiwania pamięci. Wskaż, co jest, a co nie jest

gwarantowane przez mechanizm odzyskiwania pamięci. Opisz działanie metody `finalize()`.

- 7.5. *Given the fully-qualified name of a class that is deployed inside and/or outside a JAR file, construct the appropriate directory structure for that class. Given a code example and a classpath, determine whether the classpath will allow the code to compile successfully.*

Mając podaną w pełni kwalifikowaną nazwę klasy, która ma być umieszczona wewnątrz pliku JAR albo poza plikiem, skonstruuj poprawną strukturę katalogów dla tej klasy. Mając podany przykład kodu oraz ścieżkę klas (ang. classpath) oceń, czy podana ścieżka klas jest poprawna i wystarczająca dla skompilowania kodu.

- 7.6. *Write code that correctly applies the appropriate operators including assignment operators (limited to: =, +=, -=), arithmetic operators (limited to: +, -, \*, /, %, ++, --), relational operators (limited to: <, <=, >, >=, ==, !=), the instanceof operator, logical operators (limited to: &, |, ^, !, &&, ||), and the conditional operator ( ? : ), to produce a desired result. Write code that determines the equality of two objects or two primitives.*

Napisz kod, w którym poprawnie używasz operatorów przypisania (tylko =, += i -=), operatorów arytmetycznych (tylko +, -, \*, /, %, ++ i --), operatorów relacyjnych (tylko <, <=, >, >=, == i !=), operatora `instanceof`, operatorów logicznych (tylko &, |, ^, !, && i ||) oraz operatora warunkowego. Napisz kod, który ocenia równość dwu obiektów lub wartości typów prostych.



## KLASY, INTERFEJSY, TYPY WYLICZENIOWE

1.1 Napisz kod, w którym deklarujesz klasy (wliczając klasy abstrakcyjne i wszelkie formy klas zagnieżdżonych), interfejsy i typy wyliczeniowe oraz zademonstruj poprawne użycie instrukcji `package` i `import` (w tym `import statyczny`).

1.2 Napisz kod, w którym deklarujesz interfejs oraz kod, w którym implementujesz lub dziedziczysz z jednego lub wielu interfejsów. Zaimplementuj klasę abstrakcyjną oraz klasę, która dziedziczy z klasy abstrakcyjnej.

7.1 Mając podany fragment kodu oraz scenariusz napisz program w którym używasz odpowiednich modyfikatorów dostępu, prawidłowo deklarujesz pakiet oraz używasz instrukcji importu, w którym używasz podanego fragmentu kodu poprzez dziedziczenie lub kompozycję.

## ORGANIZACJA PLIKU KODU ŹRÓDŁOWEGO

Program zaimplementowany w języku Java składa się z pewnej liczby klas, interfejsów i typów wyliczeniowych zdefiniowanych w postaci serii plików umieszczonych w odpowiedniej – odpowiadającej pakietom – strukturze katalogowej. Zarówno podział klas i interfejsów oraz typów wyliczeniowych pomiędzy pliki, jak i podział plików na katalogi a nawet nazewnictwo plików nie są dowolne; wprost przeciwnie, podlegają serii reguł i ograniczeń. Każdy z plików, który jest elementem aplikacji, tj. zawierający w sobie definicje klas, interfejsów czy typów wyliczeniowych ma także ściśle określoną strukturę wewnętrzną. Zasady organizacji plików w struktury katalogowe zostały omówione w następnym podrozdziale. Niniejszy podrozdział opisuje zaś wewnętrzną strukturę pojedynczego pliku programu. Ilekroć poniżej użyto słowa „klasa” należy rozumieć to jako „klasa, interfejs lub typ wyliczeniowy”; te same reguły dotyczą bowiem – w omawianych kwestiach – wszystkich typów definicji, tj. zarówno klas jak i interfejsów i typów wyliczeniowych.

W pojedynczym pliku źródłowym może być zdefiniowana tylko jedna klasa publiczna (tj. oznaczona modyfikatorem `public`), ale klas nie publicznych może być dowolnie wiele. Jeśli więc w pliku zdefiniowano jakąś klasę publiczną, to nie można w tym samym pliku zdefiniować kolejnej klasy publicznej, można natomiast w jednym pliku zdefiniować dowolną ilość klas nie publicznych (także wiele klas nie publicznych i klasę publiczną razem w jednym i tym samym pliku). Kolejność definicji w ramach jednego pliku nie ma znaczenia. Jeśli plik zawiera klasę publiczną to musi się on nazywać tak jak ta klasa i dodatkowo posiadać rozszerzenie (przyrostek) „.java”. Jeśli w pliku nie zdefiniowano żadnej klasy publicznej (tj. zdefiniowano tylko pewną ilość klas nie publicznych) to jego nazwa może być dowolna, z tym że nadal musi mieć rozszerzenie „.java”.

Jeśli klasa znajduje się w jakimś pakiecie (tj. w pakiecie innym niż domyślny) to deklaracja pakietu (słowo kluczowe `package`) musi być pierwszą instrukcją w pliku. Jeśli klasa znajduje się w pakiecie domyślnym to deklarację pakietu pomijamy.

Jeśli w pliku używa się instrukcji importu (słowo kluczowe `import`) to musi to być zrobione po deklaracji pakietu a przed definicją klas, interfejsów czy typów wyliczeniowych.

Importy i deklaracja pakietu dotyczą wszystkich definicji z pliku, tj. w pliku występować może tylko jedna deklaracja pakietu i dotyczy ona wszystkich klas,

interfejsów i typów wyliczeniowych zdefiniowanych w danym pliku. Także typy importowane „widoczne są” w ten sam sposób dla wszystkich definicji z pliku.

## PAKIETY

Definiując nowy typ, czy to klasę, interfejs czy typ wyliczeniowy określamy nie tylko nazwę tego typu, ale także przestrzeń nazw, czyli pakiet, do którego ten typ należy. Klasy dzielimy na pakiety by pogrupować je według ich znaczenia oraz by uniknąć konfliktu nazw. Identyfikatorem klasy jest de facto nie nazwa klasy, tylko nazwa klasy poprzedzona nazwą pakietu w którym została ona zdefiniowana. Weźmy dla przykładu nazwę „Date” – sama w sobie nazwa ta nie identyfikuje żadnej klasy; dopiero powiedzenie, że chodzi o klasę `java.util.Date`, albo `java.sql.Date` wyjaśnia sprawę. W jaki sposób określamy przestrzeń nazw, tj. pakiet? Przy pomocy instrukcji `package`, tak jak to pokazano na poniższym przykładzie:

```
package my.pkg;  
  
public class SomeClass {  
    // implementacja klasy  
}
```

Powyższy kod jest definicją klasy o nazwie `SomeClass` należącej do pakietu `my.pkg`. Klasa ta jest klasą publiczną a więc musi być zdefiniowana w pliku o nazwie „`SomeClass.java`”. Dodatkowo, plik ten musi być umieszczony w strukturze katalogowej odpowiadającej pakietowi. Ponieważ klasa należy do pakietu `my.pkg` plik ten musi znajdować się w katalogu `pkg`, który z kolei musi znajdować się w katalogu `my`. Jest to bezwzględny wymóg języka Java – plik zawierający definicje typów z pewnego pakietu musi znajdować się w strukturze katalogowej odpowiadającej temu pakietowi. Dotyczy to zarówno plików z kodem źródłowym jak i plików już skompilowanych.

## INSTRUKCJA IMPORTU

Aby w sposób jednoznaczny zidentyfikować klasę lub interfejs czy typ wyliczeniowy nie wystarczy sama nazwa – trzeba jeszcze określić pakiet w obrębie którego typ ten został zdefiniowany. Pisanie pełnej nazwy klasy (a więc poprzedzonej nazwą pakietu) każdorazowo gdy danej klasy używamy jest uciążliwe i prowadzi do zmniejszenia czytelności kodu. Instrukcja `import` pozwala rozwiązać ten problem; na początku pliku kodu źródłowego możemy wymienić klasy których będziemy w obrębie danego pliku używać, posługując się ich pełną nazwą, dzięki czemu

wiadomo będzie do której klasy odnosi się późniejsze użycie samej tylko nazwy klasy. Zerknijmy na poniższy program nieużywający jeszcze instrukcji importu:

```
public class TestClass {
    public static void main(String[] args) {
        java.util.Date date = new java.util.Date();

        java.lang.System.out.println(date.toString());
    }
}
```

A teraz zobaczmy, jak wygląda ten sam program, tyle że importujący klasy których używa:

```
import java.util.Date;

public class TestClass {
    public static void main(String[] args) {
        Date date = new Date();

        System.out.println(date.toString());
    }
}
```

Importując klasę `java.util.Date` mówimy, że ilekroć w danym pliku zostanie użyty typ `Date` będzie to typ `Date` z pakietu `java.util`. To rodzi następujące pytanie – a co jeśli w tym samym pliku chcemy używać jednocześnie typów `java.util.Date` i `java.sql.Date`, a więc dwu typów o tej samej nazwie, ale z innego pakietu? Rozwiązanie jest następujące – importujemy jeden z typów (ten, którego częściej używamy), np. `java.util.Date` – a więc późniejsze użycie typu `Date` będzie oznaczało `java.util.Date` – a gdy chcemy użyć klasy `java.sql.Date` piszemy każdorazowo pełną nazwę, poprzedzoną nazwą pakietu. Próba importowania obydwu typów zakończy się naturalnie błędem. Import obydwu klas `Date` jednocześnie nie ma przecież sensu, jako że i tak nie wiadomo by było do której z klas odnosi się późniejsze użycie samej tylko nazwy „Date”. Oczywiście możemy też nie importować żadnego z tych dwu typów i każdorazowo używać dla obydwu ich pełnej nazwy. Powyższy przykład rodzi też drugie pytanie – jak to możliwe, że kod ten jest poprawny skoro klasa `System` nie jest ani poprzedzona nazwą pakietu, ani też nie jest zaimportowana? Otóż jest zaimportowana! Wszystkie klasy z pakietu `java.lang` są automatycznie zaimportowane do każdego pliku kodu źródłowego, zupełnie jakbyśmy w każdym pliku napisali instrukcję importu:

```
import java.lang.*;
```

tyle, że nie musimy tego robić, bo robi to za nas kompilator. Nie musimy także importować klas znajdujących się w tym samym pakiecie co klasa którą implementujemy. Klasy umieszczone w tym samym pakiecie są wzajemnie widoczne bezpośrednio. Przy okazji zauważamy, że klasy można importować grupami, tzn. można zaimportować na raz wszystkie klasy z danego pakietu, posługując się znakiem \* (gwiazdka). Uwaga! Symbolem \* można zastąpić nazwy klas ale nie nazwy podpakietów. Instrukcja `import java.lang.*` dla przykładu nie powoduje zaimportowania klasy `java.lang.reflect.Proxy` czy interfejsu `java.lang.annotation.Annotation`.

Instrukcją pokrewną do instrukcji importu jest import statyczny, który służy do importowania statycznych zmiennych i metod. Ogólna przesłanka jest taka sama jak w przypadku importu klas – chęć uniknięcia konieczności wpisywania pełnych nazw; tyle że tak jak tam chodziło o uniknięcie konieczności podawania każdorazowo nazwy pakietu, tak tym razem chodzi o uniknięcie konieczności podawania nazwy klasy czy interfejsu. Zerknijmy na poniższy fragment kodu liczący pole koła o promieniu 2 i wyświetlający tę wartość zaokrągloną do liczby całkowitej:

```
public class TestClass {
    public static void main(String[] args) {
        System.out.println(Math.round(circleArea(2)));
    }

    public static double circleArea(double circleRadius) {
        return Math.PI * circleRadius * circleRadius;
    }
}
```

Użyta klasa `Math` pochodzi z pakietu `java.lang`, tak więc importowanie jej explicite nie było potrzebne. Żeby natomiast użyć stałej `PI` czy metody statycznej `round(...)` konieczne było poprzedzenie ich nazwą klasy w której zostały zdefiniowane (klasa `Math`). To samo dotyczy zmiennej `out`, która jest zmienną statyczną z klasy `System`. Ten sam kod, tyle że używający instrukcji statycznego importu wygląda następująco:

```
import static java.lang.System.out;

// import wszystkich metod i zmiennych statycznych z klasy Math
import static java.lang.Math.*;

public class TestClass {
    public static void main(String[] args) {
        out.println(round(circleArea(2)));
    }
}
```

```
public static double circleArea(double circleRadius) {  
    return PI * circleRadius * circleRadius;  
}
```

Podobnie jak w przypadku importu klas próba jednoczesnego importu dwu metod czy zmiennych statycznych o tych samych nazwach zakończy się błędem. Możliwa jest również sytuacja, w której importuje się dwa różne byty o tych samych nazwach nieświadomie, za sprawą symbolu \*, nie jest to jednak problemem do czasu kiedy nie zechcemy takiego niejednoznacznie zaimportowanego bytu użyć. Zerknijmy na poniższy przykład:

```
import static java.lang.Integer.*;  
import static java.lang.Long.*;  
  
public class TestClass {  
    public static void main(String[] args) {  
        // błąd! chodzi o Integer.MAX_VALUE czy Long.MAX_VALUE?  
        System.out.println(MAX_VALUE);  
    }  
}
```

Mimo, że źródłem problemu jest niejednoznaczny import to błędem jest dopiero próba użycia niejednoznacznie określonego symbolu – stałej `MAX_VALUE`. Tak długo jak długo symbol `MAX_VALUE` czy jakiś inny niejednoznacznie zaimportowany symbol nie zostałby użyty, tak długo program byłby poprawny.

## MODYFIKATORY DLA DEKLARACJI KLAS

Klasy są podstawowym bytem w języku Java. Można by rzec, że programowanie w Javie to nic innego jak implementowanie coraz to nowych, kolejnych, współpracujących ze sobą klas. Znaczna część niniejszej książki poświęcona jest objaśnieniu tego właśnie – w jaki sposób klasy implementować. Zacznijmy od modyfikatorów dla klas zwykłych, tj. niezagnieżdżonych.

Java określa cztery modyfikatory, które mogą być użyte w deklaracji klasy zwykłej (nie wewnętrznej). Dzielą się one na dwie kategorie. Pierwsza to modyfikatory widoczności, czyli w kontekście klas tylko `public`, alternatywnie brak modyfikatora widoczności. Druga to modyfikatory innego typu, czyli: `strictfp`, `final` i `abstract`.

Zacznijmy od modyfikatorów widoczności. Deklarację klasy możemy ozdobić modyfikatorem `public`. Możemy też nie specyfikować zakresu widoczności,

efektywnym będzie wówczas zakres domyślny. Modyfikator `public` oznacza, że klasa będzie widoczna dla każdej innej klasy. Brak modyfikatora, czyli zakres domyślny spowoduje, że klasa będzie widoczna tylko i wyłącznie dla klas zdefiniowanych w tym samym pakiecie.

Modyfikator `strictfp` oznacza, że wszystkie operacje zmiennoprzecinkowe w danej klasie będą zgodne ze standardem IEEE 754, tj. Wirtualna Maszyna Javy wykona instrukcje zmiennoprzecinkowe w sposób zgodny z tym standardem. Szczęśliwie egzamin na SCJP nie wymaga, abyśmy wiedzieli co to dokładnie oznacza. Trzeba natomiast wiedzieć, że modyfikator ten może być zastosowany tylko przy deklaracji klasy lub metody, ale nigdy przy deklaracji zmiennej; dotyczy on w końcu operacji (operacji wyliczania wartości zmiennoprzecinkowych).

Została jeszcze para modyfikatorów `final` i `abstract`. Pierwszą rzeczą, z której trzeba sobie zdać sprawę jest to, że można użyć tylko jednego z nich na raz. Klasa nie może być zadeklarowana jednocześnie `final` i `abstract`. Zadeklarowanie klasy jako `final` oznacza, że nie może ona być rozszerzona, tj. nie można zadeklarować innej klasy jako podklasy klasy `final`. Jeśli więc klasa A jest `final`, to nie będzie można zadeklarować `class B extends A`. Klasa abstrakcyjna – tj. oznaczona modyfikatorem `abstract` – to taki „interfejs z częściową implementacją”. Zadeklarowanie klasy jako `abstract` oznacza, że nie będzie możliwe utworzenie żadnej instancji tej klasy. Do czego potrzebna jest więc taka klasa? No cóż, klasy abstrakcyjne istnieją po to, by być dobrą bazą do bycia rozszerzoną przez inną klasę. Sensem istnienia klas abstrakcyjnych jest bycie rozszerzanymi, klasy `final` to takie które rozszerzane być nie mogą, oczywiste jest więc że klasa nie może być jednocześnie abstrakcyjna i `final`. Klasa abstrakcyjna zawiera pewną liczbę (być może 0) metod abstrakcyjnych i pewną liczbę (być może 0) metod z implementacją (oraz wszelkie inne deklaracje, które może zawierać klasa nie abstrakcyjna). Metody abstrakcyjne to metody, które nie posiadają implementacji; zamiast kodu ujętego w nawiasy klamrowe umieszczamy na końcu deklaracji metody średnik – tak jak przy deklaracji metod w interfejsach. Metody abstrakcyjne są dodatkowo oznaczone modyfikatorem `abstract`. Uwaga! Klasa abstrakcyjna wcale nie musi zawierać metod abstrakcyjnych, ale jeśli któraś z metod w klasie jest abstrakcyjna, to automatycznie cała klasa również staje się abstrakcyjna i musi być zadeklarowana jako `abstract`. Przykład klasy abstrakcyjnej poniżej:

```
// klasa zawiera metodę abstrakcyjną więc jest abstrakcyjna
public abstract class CollectionModifier {
    public void modifyElements(Collection collection) {
        for(Object obj : collection) {
            this.someOp(obj);
        }
    }
}
```

```
}  
  
    // metoda abstrakcyjna bez implementacji i oznaczona słówkiem abstract  
    protected abstract void someOp(Object obj);  
}
```

## DEKLARACJA INTERFEJSU

Podobnie jak w przypadku klas, interfejsy możemy oznaczać modyfikatorem widoczności `public`, lub nie używać w ogóle modyfikatora widoczności – interfejs będzie wówczas widoczny tylko w swoim pakiecie. Modyfikatory `private` oraz `protected` nie mogą być zastosowane dla interfejsów zwykłych (tj. niezagnieżdżonych), bo też nie mają w tym kontekście sensu.

Interfejs możemy także oznaczyć słówkiem kluczowym `abstract`, jednak – choć jest poprawne – nie ma to żadnego efektu i nie powinniśmy tego robić. Każdy interfejs i tak jest abstrakcyjny.

Ostatnim modyfikatorem, jaki możemy zastosować dla interfejsu niezagnieżdżonego jest `strictfp`. Stałe których wartości obliczane są w czasie kompilacji są tak czy inaczej zawsze wyliczane zgodnie z zasadami `strictfp`, tak więc modyfikator ten ma tylko ten skutek, że propaguje się na klasy zdefiniowane wewnątrz interfejsu – klasy zagnieżdżone w tym interfejsie. Skutek zadeklarowania interfejsu jako `strictfp` będzie więc taki, jak gdybyśmy wszystkie klasy wewnętrzne tego interfejsu zadeklarowali jako `strictfp`.

Interfejs może zawierać deklaracje metod abstrakcyjnych, stałych, klas, typów wyliczeniowych oraz innych interfejsów. Deklarując w interfejsie metodę nie podajemy żadnych modyfikatorów. Co prawda kod skompiluje się jeśli podamy `explicit` modyfikatory `public` albo `abstract`, ale nie mają one żadnego skutku, jako że tak czy inaczej wszystkie metody zadeklarowane w interfejsie są oznaczone tymi modyfikatorami `implicit`. Specyfikowanie tych modyfikatorów jest w złym stylu i nie należy tego robić – tak mówi specyfikacja języka Java. Deklarując metodę w interfejsie nie możemy zastosować żadnego innego modyfikatora, ale już implementując zadeklarowane metody w klasie możemy dodać modyfikatory `final`, `strictfp` lub `native`, dotyczą one bowiem implementacji a nie samego kontraktu.

Każda zmienna zadeklarowana w interfejsie musi być *de facto* stałą i jest `implicit` `public`, `static` oraz `final`. Możemy podać dowolną kombinację tych modyfikatorów `explicit`, jednak nie ma to żadnego skutku – tak czy inaczej one tam są i nie da się tego zmienić. Nie są dla zmiennych zdefiniowanych wewnątrz

interfejsów dozwolone żadne inne modyfikatory.

Klasy oraz interfejsy zadeklarowane wewnątrz deklaracji interfejsu (tj. zagnieżdżone) są *implicitnie* oznaczone jako `static` i `public`. Klasy wewnętrzne będą jeszcze obszernie omawiane w dalszych podrozdziałach.

## IMPLEMENTACJA INTERFEJSU

Co to właściwie znaczy zaimplementować interfejs? Otóż interfejs to nic innego, jak tylko zbiór deklaracji metod, zatem zaimplementować interfejs to znaczy zaimplementować wszystkie zadeklarowane w nim metody. Jeśli pewna klasa implementuje jednocześnie kilka interfejsów, to naturalnie musi implementować wszystkie metody zadeklarowane w tych interfejsach. Co to jednak oznacza zaimplementować metody zadeklarowane w interfejsie? Inaczej – jak dokładnie musi wyglądać metoda implementująca tę zadeklarowaną w interfejsie? Zacznijmy od przykładu:

```
interface SomeInf {
    Number doSomething() throws Exception;
}

class SomeClass implements SomeInf {

    // zwracany jest typ Integer a nie Number, brak też deklaracji wyjątku
    public Integer doSomething() {
        return 1;
    }
}
```

Czy powyższy kod jest poprawny? Tzn. czy metoda `doSomething()` z klasy `SomeClass` rzeczywiście implementuje tę zadeklarowaną w interfejsie? Tak, kod ten jest poprawny, albowiem sygnatury metod nie muszą być identyczne; wystarczy, że są zgodne. Implementacja metody z interfejsu jest *de facto* tym samym co nadpisanie (ang. *overriding*) metody odziedziczonej z rozszerzanej nadklasy. Możemy o tym myśleć jak o nadpisaniu metody nieposiadającej implementacji metodą z implementacją. Metody implementujące interfejsy obowiązują dokładnie te same zasady, co metody nadpisujące metody odziedziczone z nadklasy i są one szczegółowo opisane w dalszych rozdziałach.

## POPRAWNE IDENTYFIKATORY I KONWENCJE NAZEWNICZE

Każda klasa (poza klasami anonimowymi), interfejs, metoda, zmienna czy stała muszą się jakoś nazywać. Właściwe nadawanie nazw ma dwa wymiary. Po

pierwsze, nazwa musi być poprawna – bez tego nasz kod się po prostu nie skompiluje; po drugie, nazwa powinna być „ładna”, ze względu na czytelność kodu. Poprawność nazwy (identyfikatora) określają następujące zasady:

- Musi zaczynać się od litery (alfabetem jest Unicode, więc dozwolone są także litery alfabetów narodowych, w tym polskie), symbolu „\$”, albo znaku podkreślenia „\_”. Znak łącznika „-” jest niedopuszczalny. Każdy kolejny znak identyfikatora może być dodatkowo cyfrą, ale pierwszy nie!
- Nie ma ograniczenia na długość nazwy.
- Identyfikatorem nie może być słowo kluczowe języka. Upewnijmy się, że znamy wszystkie słowa kluczowe, w tym te rzadziej stosowane jak: `continue`, `goto`, `native`, `strictfp`, `transient`, `volatile`.
- Java jest wrażliwa na wielkość liter (ang. case-sensitive), litera ‘f’ nie jest w końcu tym samym co litera ‘F’, tak więc ‘Foo’ jest czym innym niż ‘foo’. Jakby się zastanowić to jest to oczywiste, w końcu typ prosty `boolean` to co innego niż klasa `Boolean`. Wartością typu `boolean` też może być tylko `true` albo `false`, a nie np. ‘TRUE’.

Innym zagadnieniem jest „ładność” (ang. naming convention) nazwy, tj. zgodność z zaleceniami Sun’a. Jeśli chodzi o identyfikatory to można owe pojęcie ograniczyć do następujących zasad:

- Nazwy klas i interfejsów powinny być zlepkiem słów pisanych wielką literą (pierwsza litera wielka a reszta mała), czyli np. „SourceManager” ale nie „sourceManager” czy „SOURCE\_MANAGER”. Taki styl nazewniczy określa się czasem mianem „camel-case”.
- Nazwy interfejsów powinny być też zazwyczaj przymiotnikami (ang. adjective), np. „Serializable”.
- Metody obowiązują te same zasady co klasy, z tym że pierwsza litera musi być mała a nie wielka, czyli np. „getData” a nie „GetData”. Dodatkowo, nazwy powinny być parami czasownik-rzeczownik (ang. verb-noun), np. „getData” czy „setCustomerId”.
- Nazwy zmiennych, tak jak nazwy metod, powinny być zlepkiem słów pisanych – z wyjątkiem pierwszego słowa – wielką literą. Poza tym, jak

zawsze jest zalecenie, aby nazwy były znaczące i możliwe, ale nie przesadnie, zwarte, np. „firstName” albo „name” ale nie samo „n”.

- Stałe to takie zmienne które oznaczono modyfikatorami `static` i `final`. Zalecane jest, aby identyfikator dla stałej składał się ze słów pisanych samymi wielkimi literami połączonymi znakiem „\_”, np. „MAX\_THREADS”.

## KLASY WEWNĘTRZNE

Klasy wewnętrzne (ang. inner classes) to klasy zdefiniowane wewnątrz innej klasy – klasy zewnętrznej. Oprócz najprostszej formy, która wygląda jak zwykła deklaracja, klasy wewnętrzne występują także w formie klas anonimowych, tj. takich, które nie posiadają nazwy. Zanim zagłębimy się w szczegółowe rozważania, zobaczmy na przykładzie, czym jest klasa wewnętrzna w swej najprostszej formie:

```
public class OuterClass {
    private String value = "Zmienna prywatna";

    class InnerClass {
        public String getOuterValue() {
            return value; // dostęp do zmiennej prywatnej
        }
    }

    public String getValue() {
        return value;
    }
}
```

Klasa `InnerClass` z powyższego przykładu to klasa wewnętrzna zdefiniowana w klasie zewnętrznej `OuterClass`. Jak widać, z klasy wewnętrznej mamy pełen dostęp do zmiennych i metod klasy zewnętrznej, także tych prywatnych. I właśnie to jest de facto sensem istnienia klas wewnętrznych – mamy możliwość wydzielenia kodu do osobnej klasy a jednocześnie zachowujemy możliwość operowania na zmiennych prywatnych. Zauważmy, że dostęp do zmiennych prywatnych z kodu klasy wewnętrznej nie jest niczym nadzwyczajnym – klasa wewnętrzna jest przecież elementem klasy zewnętrznej zupełnie tak samo jak metody, które przecież mają dostęp do zmiennych prywatnych (vide metoda `getValue()`).

Specjalna relacja między obiema klasami – wewnętrzną i zewnętrzną – dotyczy jednak de facto nie samych klas, ale ich instancji. To instancja klasy wewnętrznej ma dostęp do zmiennych i metod prywatnych instancji klasy zewnętrznej. Instancja klasy wewnętrznej w żadnym wypadku nie może istnieć bez odpowiadającej jej instancji klasy zewnętrznej, z którą jest powiązana. Zerknijmy jeszcze raz na powyższy

przykład – aby metoda `getOuterValue()` mogła działać, instancja klasy `InnerClass` musi być powiązana z instancją klasy `OuterClass`, z której pochodzi wartość zmiennej `value`. Owo powiązanie następuje w trakcie tworzenia instancji klasy wewnętrznej. Instancje klasy wewnętrznej można utworzyć zasadniczo na dwa sposoby: z kodu klasy zewnętrznej albo z kodu innej, niepowiązanej klasy. Tworzenie instancji klasy wewnętrznej z kodu klasy zewnętrznej wygląda „zwyczajnie”. Instancja klasy wewnętrznej związana jest w takim wypadku z instancją klasy zewnętrznej, dla której wywołano metodę która ją utworzyła. Zerknijmy na poniższy przykład:

```
class ExternalClass {
    public void someOp() {
        OuterClass outerInstance = new OuterClass();

        OuterClass.InnerClass innerInstance = outerInstance.getInstance();
    }
}

class OuterClass {
    private String value = "Zmienna prywatna";

    class InnerClass {
        public String getOuterValue() {
            return value;
        }
    }

    public InnerClass getInstance() {
        return new InnerClass();
    }
}
```

W metodzie `someOp()` w klasie `ExternalClass` utworzyliśmy najpierw instancję klasy `OuterClass` o nazwie `outerInstance` a następnie wywołaliśmy jej metodę `getInstance()`, która utworzyła instancję klasy `InnerClass`, powiązaną automatycznie z obiektem dla którego wykonywał się kod, a więc z instancją `outerInstance`. Zwróćmy jeszcze uwagę na sposób w jaki nazywamy klasę wewnętrzną w metodzie `someOp()`. Jeśli klasy wewnętrznej używamy poza klasą w której została ona zdefiniowana to musimy nazwę tejże poprzedzić nazwą klasy zewnętrznej, zupełnie jakby klasa zewnętrzna była pakietem, w którym zlokalizowano klasę wewnętrzną. Pełną nazwą klasy `InnerClass` jest więc `OuterClass.InnerClass`.

Tworzenie instancji klasy wewnętrznej spoza klasy w której została ona zdefiniowana (spoza klasy zewnętrznej) wymaga użycia specjalnej składni, która tworząc obiekt

pozwole jednocześnie wskazać instancję klasy zewnętrznej, z którą tworzona instancja będzie powiązana. Przykład poniżej:

```
class ExternalClass {
    public void someOp() {
        OuterClass outerInstance = new OuterClass();

        // zwróćmy uwagę na składnię użycia operatora new
        OuterClass.InnerClass innerInstance = outerInstance.new InnerClass();

        // można też tak, tworząc jednocześnie instancję klasy zewnętrznej
        innerInstance = new OuterClass().new InnerClass();
    }
}

class OuterClass {
    private String value = "Zmienna prywatna";

    class InnerClass {
        public String getOuterValue() {
            return value;
        }
    }
}
```

W normalnym przypadku chcemy jednak, aby to klasa zewnętrzna tworzyła instancje klasy wewnętrznej – klasa zewnętrzna i tylko ona. Istnieje bardzo dobry sposób, by to zagwarantować – wystarczy wszystkie konstruktory klasy wewnętrznej oznaczyć jako prywatne. Specjalna relacja, w jakiej znajdują się klasa wewnętrzna z zewnętrzną powoduje, że nie tylko klasa wewnętrzna ma dostęp do zmiennych i metod prywatnych klasy zewnętrznej, ale także klasa zewnętrzna ma pełny dostęp do „wnętrzości” klas, które w niej zdefiniowano. Relacja między klasami wewnętrznymi i zewnętrznymi działa na równi w obie strony – tj. klasy nawzajem mają dostęp do swoich prywatnych deklaracji. Ilustruje to poniższy przykład:

```
class ExternalClass {
    public void someOp() {
        OuterClass outerInstance = new OuterClass();

        // błąd! - konstruktor klasy wewnętrznej jest prywatny
        OuterClass.InnerClass innerInstance = outerInstance.new InnerClass();
    }
}

class OuterClass {
    private String value = "Zmienna prywatna";

    public class InnerClass {
        private InnerClass() {} // konstruktor jest prywatny

        public String getOuterValue() {
            return value;
        }
    }
}
```

```
    }  
}  
  
public InnerClass newInner() {  
    return new InnerClass(); // to jest ok, klasy są w specjalnej relacji  
}  
}
```

Spójrzmy teraz jeszcze raz na metodę `getOuterValue()` z powyższego przykładu. Nie dość, że zmienna `value` jest widoczna, mimo że jest zmienną prywatną innej klasy, to jeszcze widoczna jest zupełnie w taki sam sposób, jakby była zadeklarowana w klasie `InnerClass`. A co by było w przypadku, gdyby zmienna `value` była zadeklarowana zarówno w klasie zewnętrznej jak i wewnętrznej? W jaki sposób moglibyśmy wskazać, o którą z dwojga zmiennych nam chodzi? Jest to problem podobny do tego, gdy mamy zmienną o tej samej nazwie zadeklarowaną zarówno jako zmienną lokalną (w metodzie) jak i zmienną instancyjną w klasie. W takiej sytuacji, w przypadku użycia samej nazwy zmiennej odwołujemy się do zmiennej lokalnej zaś aby wskazać, że chodzi nam o zmienną zadeklarowaną w klasie nazwę zmiennej poprzedzamy słówkiem kluczowym `this`. W przypadku, gdy mamy do czynienia z kodem z klasy wewnętrznej słowo kluczowe `this` oznacza instancję klasy wewnętrznej, zaś aby wskazać, że chodzi nam o zmienną zadeklarowaną w klasie zewnętrznej słowo kluczowe `this` musimy dodatkowo poprzedzić nazwą klasy zewnętrznej. Oto przykład:

```
class OuterClass {  
    private String value = "Zmienna z klasy zewnętrznej";  
  
    public class InnerClass {  
        private String value = "Zmienna z klasy wewnętrznej";  
  
        public String getLocalValue() {  
            String value = "Zmienna lokalna";  
  
            return value; // zmienna lokalna  
        }  
  
        public String getInnerValue() {  
            String value = "Zmienna lokalna";  
  
            return this.value; // zmienna z klasy wewnętrznej  
        }  
  
        public String getOuterValue() {  
            String value = "Zmienna lokalna";  
  
            return OuterClass.this.value; // zmienna z klasy zewnętrznej  
        }  
    }  
}
```

Klasy wewnętrzne, tak jak wszystkie klasy, mogą być klasami abstrakcyjnymi albo finalnymi; mogą być także oznaczane modyfikatorem `strictfp`. Inaczej jest natomiast z modyfikatorami widoczności. Deklarując klasę zewnętrzną możemy nie specyfikować zakresu widoczności – obowiązywał będzie wówczas zakres domyślny – albo też możemy określić klasę jako publiczną z użyciem modyfikatora `public`. Klasę wewnętrzną możemy dodatkowo oznaczyć jako prywatną (modyfikator `private`) albo chronioną (`protected`). Klasę zadeklarowaną wewnątrz innej klasy możemy także oznaczyć modyfikatorem `static`, jednak wówczas będzie to już tak zwana statyczna klasa zagnieżdżona, o których będzie jeszcze mowa w osobnym podrozdziale.

## KLASY LOKALNE METODY

Klasy wewnętrzne o których mówiliśmy w poprzednim podrozdziale to klasy zadeklarowane wewnątrz klasy zewnętrznej, ale bezpośrednio w klasie, na tym samym poziomie co zmienne i metody. Można jednak zadeklarować klasę wewnętrzną także wewnątrz metody klasy zewnętrznej. Takie klasy nazywamy klasami lokalnymi metody (ang. *method-local inner classes*). Przykład poniżej:

```
public class OuterClass { // klasa zewnętrzna

    public void someOp() {
        class MethodLocalClass { // klasa lokalna metody someOp()
            public int x = 1;
        }

        // instancje klasy lokalnej metody możemy utworzyć tylko w tej metodzie
        MethodLocalClass localClassInstance = new MethodLocalClass();
    }
}
```

W powyższym przykładzie deklarujemy klasę `OuterClass` z metodą `someOp()`. Ta metoda zawiera w sobie deklarację lokalnej klasy `MethodLocalClass`, która posiada jedynie zmienną publiczną o nazwie `x`. Po deklaracji klasy w metodzie `someOp()` tworzymy jej instancję i przypisujemy ją na zmienną referencyjną `localClassInstance`. Co ważne, instancje klasy lokalnej możemy utworzyć tylko i wyłącznie w metodzie, w której tę klasę zadeklarowano – definicja takiej klasy nie jest bowiem widoczna poza tą metodą. Dodatkowo, klasa lokalna musi być zdefiniowana w metodzie powyżej jej pierwszego użycia. Zauważmy, że w przypadku klas wewnętrznych nie mamy takiego ograniczenia. Poprawna jest przecież deklaracja:

```
public class OuterClass {
```

```

    // klasa InnerClass jest zdefiniowana niżej, ale to niczemu nie szkodzi
    private InnerClass innerClassInstance = new InnerClass();

    class InnerClass {
        // dowolna implementacja klasy
    }
}

```

nie jest natomiast poprawny kod:

```

public class OuterClass {

    public void someOp() {
        // błąd! klasa MethodLocalClass jest na tym etapie jeszcze nie widoczna
        MethodLocalClass localClassInstance = new MethodLocalClass();

        class MethodLocalClass {
            // dowolna implementacja klasy
        }
    }
}

```

Typ zdefiniowany wewnątrz metody jest znany tylko w obrębie tej metody tak więc nie możemy poza tą metodą zdefiniować referencji o tym typie ani nawet zadeklarować go jako typu zwracanego przez metodę, jednak możemy zwrócić z metody instancję klasy lokalnej posługując się jej nielokalnym nadtypem, np. `Object`. Zerknijmy na poniższy przykład:

```

public class OuterClass {
    public static void main(String[] args) {
        OuterClass outerClassInstance = new OuterClass();

        // referencja obj będzie wskazywała na obiekt typu MethodLocalClass
        Object obj = outerClassInstance.someOp();

        System.out.println(obj.toString());
    }

    public Object someOp() {
        class MethodLocalClass { // każda klasa jest podklasą klasy Object
            public String toString() {
                return "Instancja klasy MethodLocalClass!";
            }
        }

        // zwracamy instancję klasy MethodLocalClass
        return new MethodLocalClass();
    }
}

```

Metoda `someOp()` z powyższego przykładu może zwrócić dowolny obiekt. Tak się akurat składa, że zwraca ona instancję klasy lokalnej `MethodLocalClass`.

W metodzie `main(...)` wywołujemy metodę `toString()` zdefiniowaną w klasie `Object` a nadpisaną w klasie `MethodLocalClass`, zatem uruchomienie programu spowoduje – zgodnie z oczekiwaniami – wyświetlenie napisu:

```
Instancja klasy MethodLocalClass!
```

Typ `MethodLocalClass` nie jest znany poza metodą w której został zdefiniowany, jednak nie przeszkadza to nam jak widać posługiwać się instancjami tej klasy poza tą metodą. Obiekty przechowywane są w pamięci na stercie (także instancje klas lokalnych), podczas gdy zmienne lokalne metod na stosie, w segmencie przypisanym do danej metody. Obiekty są usuwane ze sterty przez mechanizm oczyszczania pamięci (ang. garbage collector) dopiero gdy przestają być one użyteczne, natomiast segment stosu przechowujący zmienne lokalne metody kasowany jest natychmiast po zakończeniu się metody. Instancje klasy lokalnej mogą żyć więc dłużej (dużo dłużej) niż zmienne lokalne metody. Wynika stąd jedna bardzo ważna konsekwencja – klasy lokalne nie mają dostępu do zmiennych lokalnych metody! Rozważmy kolejny wariant metody `someOp()`:

```
public Object someOp() {
    int x = 123;

    class MethodLocalClass {
        public String toString() {
            return "x == " + x; // błąd! nie możemy odwołać się do zmiennej x
        }
    }

    return new MethodLocalClass();
}
```

Jak wiemy kod ten nie skompiluje się; wywołanie metody `toString()` na instancji klasy `MethodLocalClass` poza metodą `someOp()` nie byłoby bowiem możliwe. Zmienna `x` – składowana na stosie w segmencie przypisanym do metody `someOp()` – po zakończeniu się tej metody przecież nie istnieje – nie możemy więc pobrać wartości tej zmiennej. Od tej zasady jest jednak prosty wyjątek – możemy mianowicie odwoływać się do zmiennych finalnych. Zmienne finalne to zmienne których wartość nie może się już nigdy zmienić, zatem kompilator może po prostu w miejsce odwołania się do finalnej zmiennej lokalnej podstawić wartość tej zmiennej. Zmodyfikowanie powyższej metody `someOp()` w ten sposób, że zmienną `x` oznaczylibyśmy modyfikatorem `final` spowodowałoby więc że kod byłby poprawny. Naturalnie tak jak z kodu metody tak z kodu klasy zadeklarowanej wewnątrz tej metody mamy bezpośredni dostęp do wszystkich zmiennych i metod z klasy zewnętrznej. Pamiętajmy jednak o tym, że tak jak metoda statyczna ma dostęp tylko i wyłącznie do zmiennych statycznych tak analogicznie klasa zdefiniowana

wewnątrz metody statycznej ma dostęp tylko i wyłącznie do zmiennych statycznych klasy zewnętrznej.

Klasy lokalne metody mają z góry określony zakres widoczności tak więc nie możemy w stosunku do nich stosować modyfikatorów widoczności `public`, `protected` czy `private`. Możemy natomiast używać modyfikatorów `abstract`, `final` i `strictfp`.

## KLASY ANONIMOWE

Klasy anonimowe (ang. anonymous inner classes) to specyficzny rodzaj klas wewnętrznych – są to klasy wewnętrzne które nie posiadają nazwy. Klasę anonimową definiujemy w chwili tworzenia jej instancji, jako typ rozszerzający nadklasę nazwaną albo jako implementację pewnego interfejsu. Klasy anonimowe mogą być poręcznym narzędziem jeśli potrzebujemy utworzyć tylko i wyłącznie jedną instancję danego typu, np. tuż przed tym jak prześlemy ją do jakiejś metody. Zerknijmy na poniższy przykład:

```
public class OuterClass {
    public static void main(String[] args) {
        // tworzymy instancję klasy anonimowej implementującej Runnable
        Runnable myRunnable = new Runnable() {
            public void run() {
                System.out.println("Instancja klasy anonimowej");
            }
        }; // pamiętajmy o średniku po definicji klasy anonimowej

        runThread(myRunnable);
    }

    public static void runThread(Runnable runnable) {
        Thread thread = new Thread(runnable);

        thread.start();
    }
}
```

W metodzie `main(...)` deklarujemy zmienną referencyjną `myRunnable` typu `Runnable`. `Runnable` jest interfejsem definiującym pojedynczą metodę `run()`. Do zmiennej `myRunnable` przypisujemy instancję klasy anonimowej implementującej interfejs `Runnable`. Składnia `new Runnable() {...}`; oznacza właśnie utworzenie instancji klasy anonimowej implementującej interfejs `Runnable`, gdzie implementacja tego interfejsu jest zapisana w nawiasach klamrowych `{...}`. Po definicji klasy anonimowej, tj. za nawiasami klamrowymi obowiązkowo umieszczamy średnik.

Zwróćmy uwagę na to, że klasa anonimowa ma za zadanie tylko i wyłącznie dostarczyć implementację metod interfejsu albo nadpisać wybrane metody rozszerzanej klasy, tj. nie powinniśmy w klasie anonimowej implementować nowych metod. To że nie powinniśmy nie oznacza jednakowoż iż nie możemy tego zrobić – możemy; kod taki skompiluje się, a jedynym problemem będzie to, że metody takiej nie da się wywołać. Zerknijmy na przykład:

```
public class OuterClass {
    public static void main(String[] args) {
        class InnerClass { // klasa lokalna metody
            void printId(double num) {
                System.out.println("Klasa lokalna metody");
            }
        }

        // klasa anonimowa dziedziczy z klasy lokalnej InnerClass
        InnerClass innerClassInstance = new InnerClass() {
            // to jest przeciążenie a nie nadpisanie metody z klasy InnerClass
            void printId(int num) {
                System.out.println("Anonimowa klasa lokalna");
            }
        };

        innerClassInstance.printId(123);
    }
}
```

Powyższy kod skompiluje się i uruchomi, jednak – nie dajmy się zwieść – wywołana zostanie metoda `printId(...)` z klasy `InnerClass`. Zauważmy, że metoda `printId(...)` zdefiniowana w klasie anonimowej, której instancję przypisano do zmiennej `innerClassInstance` ma argument typu `int`, natomiast w klasie `InnerClass` typu `double`. Metoda zdefiniowana w klasie anonimowej nie jest więc nadpisanie metody z nadklasy, tylko jej przeciążeniem. Metoda ta nie jest zdefiniowana w klasie `InnerClass` i nie ma sposobu aby ją wywołać. Uruchomienie programu spowoduje więc wyświetlenie napisu:

```
Klasa lokalna metody
```

jako że zmienna `innerClassInstance` jest typu `InnerClass` a to która z przeciążonych metod zostanie wywołana zależy jest od typu referencji a nie od faktycznego typu obiektu. Więcej na ten temat dowiemy się z dalszych rozdziałów książki, poświęconych tematyce przeciążania i nadpisywania metod.

W powyższych przykładach widzieliśmy klasy anonimowe występujące w roli klas lokalnych metody, jednak możemy definiować je także na poziomie klasy – przypisując instancję klasy anonimowej do zmiennej klasowej lub instancyjnej.

W zależności od tego z którym z tych dwu przypadków mamy do czynienia klasy anonimowe obowiązują te same reguły co nazwane klasy wewnętrzne lub nazwane klasy lokalne metody. Przykładowo, klasa anonimowa zdefiniowana wewnątrz metody nie może używać nie finalnych zmiennych tej metody.

Idąc jeszcze dalej język Java umożliwia definiowanie klas anonimowych także w chwili wywoływania metody, tj. możemy wywołać metodę tworząc jednocześnie instancję definiowanej w locie klasy anonimowej. Korzystając z tej możliwości moglibyśmy pierwszy przykład z tego podrozdziału przepisać w następujący sposób:

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

        // wywołujemy metodę definiując jednocześnie klasę anonimową
        runThread(new Runnable() {
            public void run() {
                System.out.println("Instancja klasy anonimowej");
            }
        }); // średnik dopiero za instrukcją wywołania metody
    }

    public static void runThread(Runnable runnable) {
        Thread thread = new Thread(runnable);

        thread.start();
    }
}
```

## STATYCZNE KLASY ZAGNIEŹDZONE

Podobnie jak klasy wewnętrzne (ang. inner classes) klasy zagnieżdżone (ang. nested classes) definiujemy wewnątrz innej klasy – klasy zewnętrznej – jednak klasy zagnieżdżone, w odróżnieniu od wewnętrznych są to klasy statyczne. Zobaczmy na przykładzie, jak to wygląda:

```
class Notepad {
    private static Date lastCreatedDate;

    static class Note {
        private Date creationDate;

        public Date getCreationDate() {
            return creationDate;
        }

        public Date getLastCreatedDate() {
            return lastCreatedDate; // statyczna zmienna prywatna klasy zewnętrznej
        }
    }
}
```

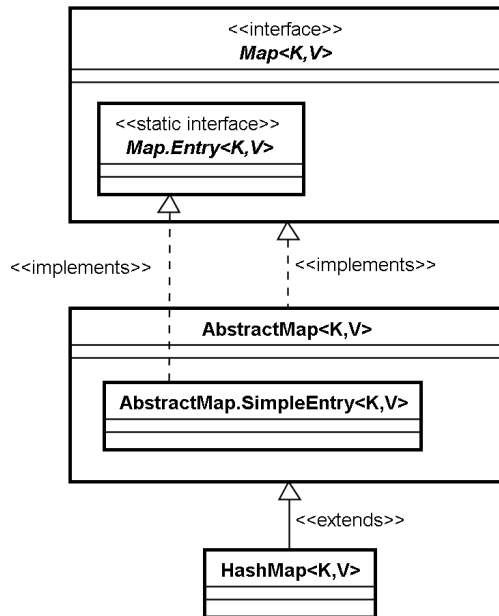
```
public static Note newNote() {
    Note art = new Note();

    art.creationDate = new Date(); // zmienna prywatna z klasy wewnętrznej
    lastCreatedDate = art.creationDate;

    return art;
}
}
```

Klasa `Notepad` jest w naszym przypadku klasą zewnętrzną a `Note` klasą zagnieżdżoną. Zauważmy, że deklarację klasy `Note` poprzedzono modyfikatorem `static`. To właśnie dodając to słówko kluczowe mówimy, że chodzi nam o klasę zagnieżdżoną a nie wewnętrzną. Podobnie jak klasę wewnętrzną, klasę zagnieżdżoną łączy z klasą zewnętrzną pewien szczególny związek, związek oparty na pełnym zaufaniu, a więc taki, który pozwala na dostęp do zmiennych prywatnych swojego partnera. Zaufanie działa w obie strony. Zarówno klasa `Note` ma dostęp do zmiennych prywatnych klasy `Notepad` (zmienna `lastCreatedDate`) jak i klasa `Notepad` do zmiennych prywatnych klasy `Note` (zmienna `creationDate`). Zwróćmy jednak szczególną uwagę na fakt, że klasy zagnieżdżone to klasy statyczne – a więc niezwiązane z żadnym obiektem klasy zewnętrznej – stąd mogą one odwoływać się tylko i wyłącznie do zmiennych (i metod) statycznych z tejże klasy zewnętrznej. Jest to sytuacja zupełnie analogiczna do tej w której jesteśmy implementując metodę statyczną, chociażby metodę `main(...)`. Możemy wówczas używać jedynie innych statycznych metod bądź statycznych zmiennych. Zmiennych instancyjnych użyć nie możemy, bo przecież nie mamy żadnej instancji!

Zanim przejdziemy dalej zastanówmy się w jakich okolicznościach ta nieco dziwna konstrukcja językowa może się przydać. Szczęśliwie, bardzo dobry przykład jest na wyciągnięcie ręki; zerknijmy na interfejs `java.util.Map<K, V>`, a szczególnie na typ wyniku metody `entrySet()`. Jest to `Set<Map.Entry<K, V>>`. A cóż to takiego `Map.Entry<K, V>`? Uwaga, uwaga... statyczny interfejs zagnieżdżony! Tak, zagnieżdżone mogą być także interfejsy i to zarówno w interfejsach jak i w klasach. Generalnie rzecz biorąc, zagnieżdżać można wszystko i we wszystkim, nawet klasy w interfejsach. Interfejs `java.util.Map<K, V>` implementuje klasa `java.util.AbstractMap<K, V>` (która z kolei jest nadklasą klasy `java.util.HashMap<K, V>`), a interfejs zagnieżdżony `Map.Entry<K, V>` klasa zagnieżdżona `AbstractMap.SimpleEntry<K, V>`. Na drzewku dziedziczenia wygląda to tak:



Jak widać z powyższych przykładów, jeśli klasy zagnieżdżonej używamy poza klasą w której została ona zdefiniowana to musimy nazwę tejże poprzedzić nazwą klasy zewnętrznej, zupełnie jakby klasa zewnętrzna była pakietem, w którym zlokalizowano klasę zagnieżdżoną. Pełną nazwą klasy `Note` jest więc `Notepad.Note`. Poniżej przykład, pokazujący jak należy identyfikować klasy zagnieżdżone, w zależności od stopnia zagnieżdżenia klasy i umiejscowienia kodu, który tej klasy używa:

```

class ExternalClass {
    void externalTest() {
        Notepad.Note.Line line = new Notepad.Note.Line();
    }
}

class Notepad {
    static class Note {
        static class Line {

        }

        void noteTest() {
            Line line = new Line();

            // można też tak - za dużo nie ma nigdy
            Notepad.Note.Line nextLine = new Notepad.Note.Line();
        }
    }
}

```

```
void notepadTest() {  
    Note.Line line = new Note.Line();  
}  
}
```

Istnieje też opcja alternatywna do każdorazowego podawania pełnej – poprzedzonej nazwą klasy zewnętrznej – nazwy klasy zagnieżdżonej. Wystarczy klasę zagnieżdżoną zaimportować; tak jak pokazano na poniższym przykładzie:

```
import my.pckg.Notepad.Note.Line;  
  
class ExternalClass {  
    void externalTest() {  
        Line line = new Line();  
    }  
}  
  
class Notepad {  
    static class Note {  
        static class Line {  
  
        }  
    }  
}
```

## TYPY WYLICZENIOWE

Typy wyliczeniowe deklarujemy za pomocą słowa kluczowego `enum` bezpośrednio w pliku, w klasie, interfejsie albo wewnątrz innego typu wyliczeniowego, ale nie w metodzie. Typ wyliczeniowy, oprócz tego, że definiuje stałe wyliczeniowe może także zawierać konstruktory, zmienne i metody oraz klasy wewnętrzne i inne typy wyliczeniowe – prawie jak klasa, ale prawie robi ogromną różnicę.

Zacznijmy od stałych wyliczeniowych. Poniżej przykład deklaracji typu wyliczeniowego, który zawiera trzy takie stałe:

```
enum Size {  
    SMALL, LARGE, HUGE  
}
```

Na tego typu deklaracjach kończy się wiedza wielu programistów, a to dopiero początek. Zacznijmy powoli, od modyfikatorów, jakich możemy użyć w deklaracji typu wyliczeniowego; dla deklaracji bezpośrednio w pliku są to tylko `public` oraz `strictfp`, a dla deklaracji wewnątrz innych typów dodatkowo `private`, `protected` i `static`. Ich znaczenie jest takie jak w przypadku klas. Idźmy dalej. Poniżej przykład deklaracji zawierającej stałe wyliczeniowe, metodę, zmienną

i konstruktor. Zwróćmy uwagę na średnik po deklaracjach stałych wyliczeniowych – jest obowiązkowy, jeśli po tych deklaracjach znajduje się coś jeszcze. Zapamiętajmy też, że stałe wyliczeniowe muszą być zadeklarowane jako pierwsze, dopiero potem, po średniku następują kolejne deklaracje:

```
enum Size {
    SMALL(1), LARGE(2), HUGE(3);

    Size(int size) {
        this.size = size;
    }

    public int getSize() {
        return size;
    }

    private int size;
}
```

To, czym typy wyliczeniowe różnią się zdecydowanie od klas jest, że mogą one zawierać jedynie konstruktory prywatne. Nie jest możliwe utworzenie instancji typu wyliczeniowego w inny sposób, jak tylko poprzez deklarację stałej wyliczeniowej. Tak, stałe wyliczeniowe są takiego typu, w jakim je zadeklarowano i są to jedyne instancje tego typu. W powyższym przykładzie stałe `Size.SMALL`, `Size.LARGE` i `Size.HUGE` są zatem typu `Size` i są to jedyne instancje tego typu jakie kiedykolwiek będą występowały w przyrodzie, tj. w JVM – innych utworzyć się nie da, także używając refleksji czy operacji `clone()`. Oznacza to między innymi, że możemy w stosunku do stałych wyliczeniowych używać operatora `==` zamiast operacji `equals()`. Mimo, że tak czy inaczej konstruktory typów wyliczeniowych są prywatne, możemy oznaczyć je słówkiem kluczowym `private` explicite. Jest to jedyny modyfikator, jakiego możemy użyć w deklaracji takiego konstruktora.

Zerknijmy teraz ponownie na deklarację stałych wyliczeniowych w powyższym przykładzie. Są to stałe i do dobrych praktyk należy, aby ich nazwy były pisane samymi wielkimi literami, ale nie jest to wymóg bezwzględny, może to być dowolny poprawny identyfikator. Tuż po nazwie stałej możemy podać parametry dla wywołania konstruktora. Jeśli argumentów nie podano – tj. użyto samej nazwy, np. „SMALL” – to dla utworzenia instancji (stałej) wywoływany jest konstruktor bezargumentowy. Tak jak w przypadku klas, możemy zdefiniować i użyć dowolny inny konstruktor.

Typy wyliczeniowe mogą implementować interfejsy, ale nie mogą dziedziczyć z innych typów wyliczeniowych. Istnieje jednak sposób na przedefiniowanie metod dla wybranych stałych wyliczeniowych, zupełnie jakby były one podtypami

zawierającego typu wyliczeniowego. Po deklaracji stałej i jej ewentualnych parametrach dla konstruktora można umieścić ciało klasy anonimowej, która dla tej jednej instancji określi typ na bazie typu zawierającego. Spójrzmy na poniższy przykład:

```
enum Size {
    SMALL {
        public String getDescription() {
            return "Taki całkiem mały";
        }
    },

    LARGE {
        public String getDescription() {
            return "Dosyć duży";
        }
    }; // nie zapomnijmy o średniku

    public abstract String getDescription();
}
```

Mimo, iż nie możemy zadeklarować, że dany typ wyliczeniowy dziedziczy z innego typu, to jak najbardziej typy wyliczeniowe również należą do wspólnej hierarchii dziedziczenia z klasą `Object` w korzeniu. Każdy typ wyliczeniowy `E` dziedziczy implicite z typu `Enum<E>`, a ten z typu `Object`. Oprócz metod odziedziczonych z `Enum<E>` każdy typ wyliczeniowy `E` ma zdefiniowane metody:

```
public static E[] values()

public static E valueOf(String name)
```

Metoda `values()` zwraca tablicę stałych wyliczeniowych zdefiniowanych w typie `E` a metoda `valueOf()` służy do konwersji z typu `String` – zwraca instancję typu `E` (jedną ze stałych wyliczeniowych) o podanej nazwie.

Jeśli chodzi o deklaracje metod, to dozwolone są wszystkie te modyfikatory, co dla deklaracji w klasach, z jednym tylko obostrzeniem dla modyfikatora `abstract` – jeśli w typie wyliczeniowym zadeklarowano metodę abstrakcyjną, to musi być ona zdefiniowana przez każdą ze stałych wyliczeniowych w tym typie. Musi przy tym być zadeklarowana co najmniej jedna taka stała. Zmienne obowiązują te same zasady, co zmienne w klasach.

Książka do nabycia w serwisie [getSCJP.pl](http://getSCJP.pl). Zapraszam.

- Autor