PRZYGOTOWANIE DO CERTYFIKACJI

OCPJP 6

MARIUSZ LIPIŃSKI

PRZYGOTOWANIE DO CERTYFIKACJI

OCPJP 6

wydanie pierwsze

kontakt: mariusz.lipinski@ocpjp.net

www.ocpjp.net

ISBN 978-83-929398-2-5

SPIS TREŚCI

WSTĘP

Celem niniejszej książki jest przygotowanie czytelnika do egzaminu na certyfikat Oracle Certified Professional, Java SE 6 Programmer (1Z0-851) skrótowo nazywanego OCPJP.

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 OCPJP.

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 OCPJP

Certyfikat OCPJP – 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…). W chwili oddawania tej książki do druku egzamin składa się z 60 pytań; limit czasowy to 150 minut. Aby zaliczyć trzeba uzyskać co najmniej 61% punktów, tak więc należy w pełni poprawnie odpowiedzieć na co najmniej 37 pytań. Liczba pytań, czas trwania egzaminu oraz wymagany procent poprawnych odpowiedzi zmieniają się okresowo. Aktualne informacje dostępne są na stronie internetowej firmy Oracle.

By móc przystąpić do egzaminu nie trzeba posiadać żadnych innych certyfikatów, tj. nie ma żadnych warunków wstępnych dopuszczenia do egzaminu. Natomiast posiadanie certyfikatu OCPJP lub jego poprzednika, tj. certyfikatu SCJP (Sun Certified Programmer for the Java Platform, Standard Edition) jest warunkiem wstępnym do przystąpienia do egzaminów dla certyfikatów specjalistycznych, takich jak Oracle Certified Expert, Java EE 6 Web Component Developer (OCEWCD) czy Oracle Certified Expert, Java EE 6 Web Services Developer (OCEWSD).

ZAKRES EGZAMINU OCPJP

Zakres materiału jaki obowiązuje na egzaminie – wiedza którą trzeba przed nim posiąść – są dość precyzyjnie określone. Firma Oracle, wystawca certyfikatu OCPJP, 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). Wytłumacz, 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,
NullPointerException,
NumberFormatException,
AssertionError,
ExceptionInInitializerError,

StackOverflowError or
NoClassDefFoundError.
Understand which of these are thrown by the virtual machine and recognize situations in which others should be thrown programatically.

Określ, jakie sytuacje mogą spowodować rzucenie wyjątków:
`ArrayIndexOutOfBoundsException`,
`ClassCastException`,
`IllegalArgumentException`,
`IllegalStateException`,
`NullPointerException`,
`NumberFormatException`,
`AssertionError`,
`ExceptionInInitializerError`,
`StackOverflowError` lub
`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), `*` (gwiazdka), `+` (plus), `?`, `\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

Niniejszy rozdział pokrywa prezentowanym zakresem wiedzy następujące wymagania egzaminu 1Z0-851:

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".

Pyt.1 **Czy poprawnym będzie zdefiniowanie w jednym pliku kodu źródłowego publicznej klasy i publicznego typu wyliczeniowego?**

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

Odp.1 **Nie! Nie możemy umieszczać dwu definicji typów publicznych w jednym pliku, niezależnie od rodzaju tych definicji.**

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.pckg;

public class SomeClass {

  // implementacja klasy
}
```

Pyt.2 **Czy możliwe jest jednoczesne zdefiniowanie klasy o nazwie *Date* i interfejsu o naziwe *Date* w tym samym pakiecie?**

Powyższy kod jest definicją klasy o nazwie `SomeClass` należącej do pakietu `my.pckg`. 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.pckg` plik ten musi znajdować się w katalogu `pckg`, 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:

Odp.2 **Nie! Nie byłoby możliwości odróżnienia jednej definicji od drugiej. Nazwa typu w parze z nazwą pakietu musi jednoznacznie identyfikować definicję.**

```
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`.

Pyt.3 **Załóżmy, że implementujemy klasę o nazwie *A* w pakiecie *com.some* i że w tym samym pakiecie znajduje się już zaimplementowana uprzednio klasa o nazwie *B*. W jaki sposób możemy w kodzie implementowanej klasy *A* użyć klasy o nazwie *B* umieszczonej w pakiecie domyślnym?**

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;
  }
}
```

Odp.3 **Nie możemy! Nie ma możliwości zaimportowania typów zdefiniowanych w pakiecie domyślnym. Użycie samej tylko nazwy *B* będzie w tym wypadku odnosiło się do typu zdefiniowanego w pakiecie *com.some*.**

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 (niezagnieżdżonej). 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 OCPJP 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 finalne to takie które rozszerzane być nie mogą, oczywiste jest więc że klasa nie może być jednocześnie abstrakcyjna i finalna. Taka klasa nie miała by żadnego zastosowania.

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.

Pyt.4 Czym w głównej mierze różni się interfejs od klasy abstrakcyjnej?

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 explicite modyfikatory `public` albo `abstract`, ale nie mają one żadnego skutku, jako że tak czy inaczej wszystkie metody zadeklarowane w interfejsie są oznaczone tymi modyfikatorami implicite. 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 implicite `public`, `static` oraz `final`. Możemy podać dowolną kombinację tych modyfikatorów explicite, 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 implicite 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:

Odp.4 **W interfejsie wszystkie metody są abstrakcyjne. W klasie abstrakcyjnej część, a nawet wszystkie metody mogą posiadać implementację.**

```
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`.

Pyt.5 **Czy można utworzyć instancję klasy abstrakcyjnej która nie zawiera żadnej abstrakcyjnej metody?**

- 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 Oracle'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 możliwie, ale nie przesadnie, zwięzłe, 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".

Odp.5 **Nie! Nie można utworzyć instancji klasy oznaczonej słówkiem kluczowym *abstract*, niezależnie od tego jaka jest implementacja tej klasy.**

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 całkiem 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 pozwoli 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 „wnętrzności" 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:

Pyt.6 **Czy klasy można zagnieżdżać wielokrotnie? Tzn. czy w klasie wewnętrznej można zdefiniować kolejną klasę wewnętrzną?**

```
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:

Odp.6 **Tak! Można. Nic nie stoi na przeszkodzie. Bardziej zewnętrzna klasa wewnętrzna będzie wówczas klasą zewnętrzną dla klasy o wyższym stopniu zagnieżdżenia.**

```
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.

Pyt.7 **Jeśli klasa WWW jest klasą wewnętrzną klasy WW, zaś klasa WW jest klasą wewnętrzna klasy W, to czy w metodzie implementowanej w klasie W możemy wywołać prywatną metodę z klasy WWW? Tj. czy relacja dająca prawo dostępu do składowych prywatnych klasy wewnętrznej jest przechodnia?**

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:

Odp.7 **Tak! Możemy, tzn. relacja ta jest przechodnia. Najlepiej będzie jeśli przekonamy się o tym sami implementując program testowy.**

```
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
      // niewidoczna
    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. Możemy jednak 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 tekstu:

```
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

Pyt.8 **Załóżmy, że w metodzie *op()* klasy *K* umieszono instrukcję tworzącą instancję klasy *K* a w kolejnych liniach zdefiniowano klasę lokalną tej metody o tej samej nazwie *K*. Instancja której klasy *K* zostanie utworzona w metodzie *op()*?**

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 przekażemy 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 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.

Odp.8 **To było pytanie podchwytliwe, sformułowane w ten sposób dla zmylenia uwagi. Żadna instancja nie zostanie utworzona, albowiem taki program się nie skompiluje. Typy zagnieżdżone nie mogą przysłaniać typów wewnątrz których są zdefiniowane. W innych przypadkach przysłanienie typów jest możliwe.**

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 nadpisaniem 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 tekstu:

```
Klasa lokalna metody
```

jako że zmienna `innerClassInstance` jest typu `InnerClass` a to która z przeciążonych metod zostanie wywołana zależne 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 niefinalnych 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 ZAGNIEŻDŻONE

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:

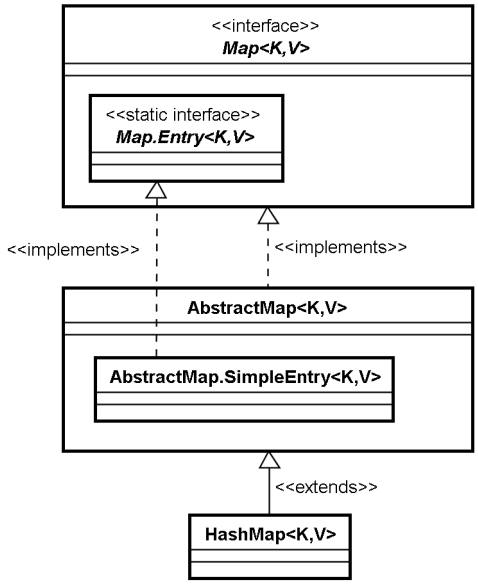
```
class Notepad {
  private static Date lastCreatedDate;

  static class Note {
    private Date creationDate;

    public Date getCreationDate() {
      return creationDate;
    }

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

  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:

Pyt.9 **Możliwe jest zdefiniowanie wewnątrz interfejsu klasy zagnieżdżonej, a czy możliwe jest zdefiniowanie klasy lokalnej w jednej z metod zaimplementowanych w interfejsie?**



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`.

Odp.9 **Nie! Przecież interfejs nie może zawierać implementacji metod. Nie mamy możliwości zdefiniowania klasy lokalnej metody, bowiem nie mamy możliwości zdefiniowania implementacji dla żadnej metody.**

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 słówka 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.

METODY, KONSTRUKTORY I ZMIENNE

Niniejszy rozdział pokrywa prezentowanym zakresem wiedzy następujące wymagania egzaminu 1Z0-851:

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

1.5 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.

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

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

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

7.6 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.

DEKLARACJA ZMIENNYCH I STAŁYCH

Java definiuje dwa rodzaje zmiennych: zmienne typu prostego (ang. primitives) i zmienne będące referencjami (ang. reference variables). Jest 8 typów prostych, są to: `char`, `boolean`, `byte`, `short`, `int`, `long`, `double` oraz `float`. Referencje są de facto wskaźnikami na obiekty, ale nie na wszystkie obiekty a tylko na te zgodne z typem referencji. Jeśli więc zadeklarujemy referencję na obiekty klasy `A`, to może ona wskazywać tylko na obiekty klasy `A` oraz na obiekty klas pochodnych.

W języku Java – w przeciwieństwie do np. C++ – wszystkie klasy uczestniczą we wspólnej hierarchii dziedziczenia z klasą `Object` w korzeniu, tak więc możemy zadeklarować referencję do obiektów klasy `Object`, która będzie mogła wskazywać na obiekt dowolnego typu, jako że wszystkie klasy są klasami pochodnymi klasy `Object`.

Zmienna typu logicznego `boolean` może przyjmować jedną z dwu wartości: `true` albo `false`. Java jest językiem wrażliwym na wielkość liter (ang. case-sensitive), a więc 'true' to co innego niż 'TRUE' i co innego niż 'True' zaś wartością zmiennej typu `boolean` może być tylko `true` albo `false` (pisane małymi literami).

Pyt.10 **Czy do zmiennej referencyjnej typu *I*, gdzie *I* jest interfejsem, możemy przypisać instancję klasy *A*, jeśli *A* dziedziczy z klasy *B* a klasa *B* implementuje interfejs *I*?**

Typ znakowy `char` reprezentowany jest na dwu bajtach i odpowiada jednemu znakowi UTF-16.

Pozostałe typy proste to typy numeryczne. Poniższa tabelka pokazuje liczbę bajtów użytą do reprezentacji liczb poszczególnych typów:

byte	short	int	long	float	double
1	2	4	8	4	8

Zmienne można deklarować w czterech różnych kontekstach. Zmienne niestatyczne zadeklarowane w klasie to zmienne instancyjne. Jeśli do zmiennej zadeklarowanej w klasie dodamy modyfikator `static` to staje się ona zmienną klasową. W ciele metody możemy zadeklarować zmienną lokalną. Ostatnim sposobem na deklarowanie zmiennych są parametry metod. W zależności od tego, z którą z powyższych deklaracji mamy do czynienia możliwe jest zastosowanie innego zestawu modyfikatorów. Przypomnijmy jeszcze, że można zadeklarować (i zainicjalizować) kilka zmiennych w jednej deklaracji, poprawne są więc deklaracje tej postaci:

```
int a = 0, b, c = 2;

Integer i = 5, j;
```

Przyjrzyjmy się teraz bliżej zmiennym instancyjnym. Są to te zmienne, które zostały zadeklarowane bezpośrednio w klasie i nie są oznaczone jako `static`. Zmienne instancyjne są inicjalizowane na domyślne wartości w chwili tworzenia instancji (a zmienne klasowe w momencie wczytywania klasy), tak więc nie muszą być inicjalizowane explicite. Zmienne te mogą mieć każdy z czterech zakresów widoczności, a więc dozwolony jest każdy z modyfikatorów: `public`, `protected` i `private`. Modyfikator widoczności może być również pominięty i wtedy obowiązywał będzie zakres domyślny. Zakresy widoczności działają analogicznie jak dla metod. Ze zmiennej takiej możemy zrobić stałą dodając modyfikator `final`. Możemy też użyć słówka kluczowego `transient` oraz `volatile`, o których za chwilkę. Legalne jest także użycie względem zmiennej zadeklarowanej w klasie słówka kluczowego `static`, ale oznacza ono właśnie tyle, że nie jest to zmienna instancyjna tylko klasowa. Zmienne klasowe mogą być oznaczone tymi samymi modyfikatorami, co zmienne instancyjne.

Odp.10 **Tak! Jeśli klasa *B* implementuje interfejs *I* to każda klasa która z niej dziedziczy także go implementuje.**

Modyfikator `transient` jest znaczący w kontekście serializacji. Zmienne instancyjne oznaczone jako `transient` są ignorowane, tj. nie są serializowane i ich stan nie jest odtwarzany przy deserializacji. Modyfikator `volatile` jest istotny w kontekście programów współbieżnych a jego znaczenie wymaga dobrego zrozumienia zagadnień współbieżności – zadanie to odkładamy na później. Póki co

zapamiętajmy tylko, że modyfikator `volatile` nie może być użyty w parze z modyfikatorem `final`; innymi słowy – stała nie może być `volatile`.

Zmienne lokalne, to zmienne zadeklarowane w ciele metod. W przeciwieństwie do zmiennych instancyjnych nie są one inicjalizowane na domyślne wartości, tak więc to programista ma obowiązek przypisać jakąś wartość zanim zmienna zostanie użyta. Zmienne lokalne mogą być oznaczone jedynie modyfikatorem `final`, co ma ten skutek, że wartość takiej zmiennej nie będzie mogła być zmieniona – będzie to stała. Zmienne będące parametrami metody również mogą być oznaczone tylko jako `final`. Ze zmiennymi lokalnymi wiąże się jeszcze jedno zagadnienie – przysłanianie zmiennych. Jeśli parametr metody, albo zmienna lokalna ma tę samą nazwę (typ nie ma znaczenia), co zmienna zadeklarowana w klasie, to zmienna zadeklarowana w klasie jest przysłonięta, tj. nazwa odnosi się do zmiennej lokalnej lub parametru metody, a odwołanie się do zmiennej zadeklarowanej w klasie wymaga użycia operatora `this`. Poniżej przykład ilustrujący to zagadnienie – efektem uruchomienia metody jest wypisanie cyfry 5 a następnie tekstu „Ola ma kota”:

```
public class TestClass {
  String var = "Ola ma kota";

  void test() {
    int var = 5;

    System.out.println(var);
    System.out.println(this.var);
  }
}
```

Pewnym szczególnym przypadkiem zmiennych są zmienne typu tablicowego. Tablice mogą być zarówno jedno jak i wielo wymiarowe. Deklarując tablicę, która będzie zawierała elementy danego typu dodajemy po nazwie typu albo po nazwie zmiennej nawiasy kwadratowe `[]`, przy czym zdecydowanie zaleca się umieszczać je po nazwie typu. Poniżej kilka przykładów poprawnych deklaracji tablic, przy czym pierwsze dwie deklaracje są zgodne z zaleceniami a kolejne, mimo że poprawne stanowią przykład deklaracji nieeleganckich:

```
String[] names;

boolean[] [] allowances;
```

```
int codes[];

String[] names[];
```

Dla tych, którzy zanim zaczęli swą przygodę z Javą, programowali w takich językach jak C czy C++ powiem jeszcze wprost, że w Javie nie jest poprawnym określenie rozmiaru tablicy w deklaracji, nie jest np. poprawną deklaracja `int[2] codes`. Deklaracja jest tylko deklaracją – pamięć na tablicę jest alokowana dopiero wówczas, gdy tworzony jest obiekt tablicy i to wówczas istotny jest jej rozmiar.

WARTOŚCI DOMYŚLNE ZMIENNYCH

Co się dzieje, gdy deklarujemy zmienną? Czy musimy przypisać jej jakąś wartość? Czy jeśli tego nie zrobimy, zmienna będzie zainicjalizowana jakąś wartością domyślną? Prosta zasada brzmi – zmienne zadeklarowane w klasie są inicjalizowane wartością domyślną zawsze a zmienne lokalne nigdy. Dodatkowo, elementy tablic (ang. arrays) są zawsze inicjalizowane wartością domyślną, niezależnie od tego czy sama tablica została zadeklarowana lokalnie czy w klasie. Zerknijmy na poniższy przykład:

```
public class Test {
  static int x;

  public static void main(String[] args) {
    System.out.println(x);
  }
}
```

Zmienna `x` została zadeklarowana w klasie i nie została zainicjalizowana explicite, wobec tego zostanie jej przypisana wartość domyślna. To, że zmienna ta jest statyczna nie ma znaczenia – istotne jest, że nie jest to zmienna lokalna, tj. nie jest zadeklarowana w metodzie lub konstruktorze. Wartość domyślna jest różna dla różnych typów. Dla typów całkowitoliczbowych, tj. `byte`, `short`, `int` i `long` wartością domyślną jest `0`. Dla typów liczbowych zmiennopozycyjnych, tj. `float` i `double` wartością domyślną jest `0.0`. Wartość domyślna dla typu logicznego `boolean` to `false`. Dla typu znakowego `char` jest to wartość `\u0000`. Zmienne referencyjne są domyślnie inicjalizowane wartością `null`. Powyższy

program skutkuje więc wypisaniem liczby 0. Zerknijmy dla kontrastu na kolejny przykład:

```
public static void main(String[] args) {
  int x;

    // błąd!
  System.out.println(x);
}
```

Nie tylko kod ten się poprawnie nie wykona, ale nawet się nie skompiluje. Próba użycia niezainicjalizowanej zmiennej lokalnej jest błędem kompilacji. Co ważne, błąd wynika z próby użycia zmiennej nie zainicjalizowanej, a nie z faktu deklaracji zmiennej bez przypisania wartości. Wartości można przecież przypisać później, aczkolwiek koniecznie przed pierwszym użyciem. I kolejny przykład:

```
public static void main(String[] args) {
  int x;

  if(args.length > 0)
    x = 1;

   // błąd!
  System.out.println(x);
}
```

Co prawda w powyższym kodzie występuje instrukcja inicjalizacji zmiennej `x`, ale jest ona wykonywana warunkowo i kompilator nie jest w stanie stwierdzić, czy rzeczywiście zostanie ona zawsze wykonana, a więc kod również się nie skompiluje. Gdyby wartość dozoru instrukcji warunkowej dała się wyliczyć statycznie i było by to `true` (np. dozór `1 < 2`) to sprawa wyglądałaby inaczej – kompilator wiedziałby, że zmienna będzie zawsze zainicjalizowana i kod by się skompilował. A teraz zobaczmy jak sprawa się ma z tablicami:

Pyt.11 **Czy zmienne zadeklarowane w klasie która jest klasą lokalną metody są automatycznie inicjalizowane?**

```
public static void main(String[] args) {
  int[] array = new int[16];

  for(int x : array)
    System.out.println(x);
}
```

Program się skompiluje i poprawnie wykona, wypisując w rezultacie uruchomienia ciąg szesnastu cyfr 0. Tak jak wcześniej napisałem – elementy tablic są zawsze inicjalizowane na wartości domyślne, niezależnie od miejsca deklaracji samej tablicy. Oczywiście samą tablicę musieliśmy utworzyć explicite. I jeszcze ostatni przykład:

```
public class Test {
  public static void main(String[] args) {
    SomeClass sc = new SomeClass();

    System.out.println("someFloat: " + sc.someFloat
        + " doubleValue: " + sc.doubleValue);
  }
}

class SomeClass {
  Float someFloat;

  double doubleValue;
}
```

Uruchomienie powyższego programu spowoduje wyświetlenie tekstu „someFloat: null doubleValue: 0.0”.

LITERAŁY

Odp.11 **Tak! Nie ma znaczenia czy klasa jest lokalna czy zagnieżdżona. Zmienne deklarowane w klasie zawsze są automatycznie inicjalizowane na wartości domyślne odpowiednie dla danego typu.**

Literały (ang. literals) to wartości wpisane bezpośrednio w kodzie źródłowym programu. W języku Java wyróżniamy 6 różnych typów literałów; są to: literały liczbowe całkowite i zmiennopozycyjne, literały logiczne, znakowe, tekstowe i literał `null`.

Literały liczbowe całkowite (ang. integer literals) mogą być wyrażone jako liczby dziesiętne, szesnastkowe i ósemkowe. Literały szesnastkowe oznaczamy przedrostkiem `0x` lub `0X`, po czym następuje liczba szesnastkowa, tzn. dowolny ciąg cyfr od 0 do 9 i liter od a do f lub A do F. Dopuszczalne jest dowolne mieszanie liter dużych i małych. Liczba może mieć także dowolną ilość zer początkowych, co naturalnie nie wpływa na jej wartość. Zer początkowych nie

może mieć natomiast liczba dziesiętna, ponieważ umieszczenie zera na początku liczby oznacza, że jest to liczba ósemkowa. Liczbę ósemkową oznaczamy właśnie w ten sposób, że umieszczamy cyfrę 0 jako przedrostek. Zer tych może być dowolnie wiele. Naturalnie cyfry ósemkowe to cyfry z zakresu od 0 do 7. Przykładowe literały dziesiętne to: `123`, `0`, `9809238`; literały ósemkowe to: `0123`, `00`, `0000077143`; a szesnastkowe: `0xABC`, `0X123`, `0xAbCd`, `0X1A2b`, `0x0`, `0x00001`. Egzamin na OCPJP co prawda nie wymaga od nas biegłości w posługiwaniu się innymi niż dziesiętny systemami liczbowymi, ale warto wiedzieć, że z pozoru dodatnie literały szesnastkowe i ósemkowe mogą reprezentować liczby ujemne, np. literał `0x80000000` reprezentuje wartość dziesiętną -2147483648, zatem wyrażenie `-0x80000000` reprezentuje wartość dodatnią 2147483648.

Domyślnie literały całkowite są typu `int`. Literały typu `long` otrzymujemy poprzez dodanie sufixu `L` lub `l`. Literały `123` czy `0xAbCd` są więc typu `int` a literały `123L` czy `0xAbCdl` typu `long`. Ze względu na podobieństwo małej litery l do cyfry 1 proponuje się używanie tylko dużej litery L.

Literały liczbowe zmiennopozycyjne mogą być wyrażone jako liczby dziesiętne i szesnastkowe. Ogólnie, literały liczbowe zmiennopozycyjne składają się kolejno z części całkowitej, kropki, części ułamkowej, symbolu wykładnika i wartości wykładnika oraz litery oznaczającej typ. Wszystkie literały zmiennopozycyjne są domyślnie typu `double`. Literały typu `float` oznaczamy poprzez dodanie na końcu litery F lub f. Możemy także explicite oznaczyć, że liczba jest typu `double` dodając literę D lub d. Litera oznaczająca typ to zatem jedna z liter: F, f, D, d.

Jeśli chodzi o dziesiętne literały zmiennopozycyjne, to wszystkie składowe są opcjonalne, przy czy musi być obecne co najmniej jedno z dwojga: część całkowita albo ułamkowa; oraz jedno z trojga: kropka albo symbol i wartość wykładnika albo litera oznaczająca typ. Część całkowita i ułamkowa to dziesiętne liczby naturalne. Symbol wykładnika to litera E lub e. Wartość wykładnika to dziesiętna liczba całkowita, dodatnia lub ujemna. Przykładowe poprawne dziesiętne literały zmiennopozycyjne to zatem: `0D`, `1.F`, `123.`, `.3450`, `23e1f`, `1e1`, `3.0E-1F`.

Literały zmiennopozycyjne szesnastkowe obowiązują inne reguły. Obowiązkową częścią jest symbol i wartość wykładnika, przy czym symbol wykładnika to litera P lub p. Obowiązkowa jest też – analogicznie jak dla systemu dziesiętnego – część

całkowita albo ułamkowa. Pozostałe części są opcjonalne. Część całkowita zapisana być musi jako liczba szesnastkowa, a więc z prefixem `0X` lub `0x`. Przykładowe poprawne szesnastkowe literały zmiennopozycyjne to: `0x3p1`, `0x2.3p0f`, `0x.1p4d`, `0xab.cdp0d`. Nie wiedzieć czemu ktoś miałby chcieć zapisywać liczby w takich postaciach, ale język Java to umożliwia a egzaminatorzy na OCPJP mogą się o tę wiedzę niestety upomnieć.

Literały logiczne to jeden z dwojga literałów: `true` albo `false`. Literały te są typu `boolean` i są to jedyne wartości, które może przyjmować zmienna tego typu. Inaczej niż np. w języku C++, literały `0` czy `1` nie mogą być użyte jako wartości logiczne.

Literały znakowe to pewna reprezentacja pojedynczego znaku ujęta w apostrofy. Znaki mogą być reprezentowane wprost – np. litera a, symbol %, czy cyfra 1 –, w postaci kodu UTF-16 poprzedzonego prefixem `\u` – czyli od `\u0000` do `\uffff` (albo `\uFFFF`) – albo jako sekwencje specjalne, tj. `\t`, `\n`, `\r`, `\\`, `\'`. Mamy jeszcze dodatkowo dwa wyjątki – nie można znaków reprezentować poprzez sekwencje `\u000a` ani `\u000d`. Odpowiadają one kolejno sekwencjom `\n` i `\r` i są niedopuszczalne ze względu na konstrukcję procesu kompilacji. Szczęśliwie w szczegóły nie potrzebujemy wnikać. Przykładowe literały znakowe to: `'a'`, `'H'`, `'*'`, `'\u1234'`, `'\n'`. Literały znakowe są zawsze typu `char`.

Literały tekstowe to ujęty w cudzysłowy dowolny ciąg znaków, reprezentowanych – z małym wyjątkiem – w taki sam sposób jak dla literałów znakowych. Tym małym wyjątkiem jest, że w literałach tekstowych można explicite umieścić znak apostrofu `'` a nie można umieścić znaku cudzysłowu `"`, tak więc nie musimy – choć możemy – używać sekwencji `\'` a z kolei musimy pisać `\"` zamiast `"`.

OBIEKTY TYPU TABLICOWEGO

Tablice w języku Java są obiektami przechowującymi sekwencję zmiennych pewnego ustalonego typu. Deklarując tablicę podajemy właśnie typ zmiennych, które będzie ona przechowywała oraz wymiar (ang. dimension) – wymiar, nie rozmiar(!). Deklaracja tablicy, czyli mówiąc precyzyjniej deklaracja

referencji na obiekt typu tablicowego to tylko deklaracja referencji – nie powoduje utworzenia żadnego obiektu. Deklarując referencję typu tablicowego nie podajemy rozmiaru tablicy, bo jest zbędny. Jest on natomiast wymagany, gdy tworzymy właściwy obiekt tablicy; musi być bowiem wiadomym, jak duży obszar pamięci trzeba zaalokować. Poniżej dwa przykłady pokazujące deklaracje i utworzenie obiektów tablic:

```
int[] nums = new int[16];

Thread[] threads = new Thread[4];
```

Elementy tablic są zawsze inicjalizowane domyślnymi wartościami. Jednak, wartością domyślną dla zmiennej referencyjnej jest `null`, tak więc druga z pokazanych deklaracji tablic bynajmniej nie spowoduje utworzenia czterech instancji obiektów klasy `Thread`. Elementy tej tablicy to referencje a nie obiekty same w sobie.

Pyt.12 **Jeśli klasa *B* dziedziczy z klasy *A*, to czy do poszczególnych elementów tablicy typu *A[]* mogę przypisywać obiekty typu *B*?**

Tablice wielowymiarowe to de facto tablice tablic. Zdaje się, że stwierdzenie to jest banałem i nic nie wnosi, ale jest zupełnie przeciwnie. Tablica dwuwymiarowa dla przykładu nie musi być bowiem tablicą „prostokątną" a trójwymiarowa nie musi być „kostką". Wszystko powinien wyjaśnić poniższy przykład:

```
public class Test {
  public static void main(String[] args) {
    int[][] myTable = new int[4][];

    myTable[0] = new int[2];
    myTable[1] = new int[4];
    myTable[2] = new int[6];
    myTable[3] = new int[8];
  }
}
```

Powyższy kod jest jak najbardziej poprawny – kompiluje się i wykonuje, choć nic ciekawego nie robi. Zacznijmy analizę od pierwszej linii. Deklarujemy w niej referencję na obiekt tablicy. Jak już powiedziano tablica przechowuje zmienne pewnego ustalonego typu – w naszym przypadku jest to typ tablicowy, przechowujący z kolei zmienne typu `int`. Tworzymy też obiekt tablicy i mówimy,

że obiekt ten ma zawierać cztery referencje do obiektów typu `int[]`. Referencje te póki co mają wartość `null`; nie określiliśmy bowiem drugiego wymiaru tablicy i kompilator nie wiedział jakie obiekty utworzyć. Obiekty te tworzymy sami w kolejnych liniach programu. Naturalnie zadanie to możemy powierzyć kompilatorowi – wystarczy, że określimy drugi wymiar, tak jak pokazuje poniższy przykład:

```
public class Test {
  public static void main(String[] args) {
    int[][] myTable = new int[4][2];

    myTable[1] = new int[4];

    for(int[] subTable : myTable) {
      for(int i : subTable)
        System.out.print(i + " ");

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

Odp.12 **Tak! Elementami tablicy typu *A[]* są przecież referencje typu *A*. Klasa *B* dziedziczy z klasy *A* więc takie przypisanie jest poprawne.**

Żeby przykład nie był zbyt banalny oraz żeby pokazać, że w istocie tablica `myTable` zawiera nic więcej jak tylko cztery referencje na obiekty typu `int[]` w drugiej linii kodu przypisuję do jednej z tych referencji nowy obiekt – tablicę zawierającą cztery zmienne typu `int`, w miejsce tablicy zawierającej dwie zmienne. Efektem uruchomienia powyższego programu będzie pokazana poniżej mozaika zer – domyślnych wartości zmiennych typu `int`:

```
0 0
0 0 0 0
0 0
0 0
```

INICJALIZACJA TABLIC

Tablice w języku Java są obiektami. Gdy potrzebujemy zmiennej typu prostego wystarczy że zadeklarujemy taką zmienną i przypiszemy jej pewną

wartość. Gdy potrzebujemy obiektu deklarujemy zmienną referencyjną, tworzymy ten obiekt i przypisujemy referencję do wspomnianej zmiennej. Jeśli tym obiektem jest tablica to dodatkowo chcielibyśmy przypisać pewne wartości elementom tej tablicy. Zerknijmy wpierw na poniższy kod, który deklaruje, tworzy i inicjalizuje tablicę:

```
public static void main(String[] args) {
  int[] myTable = new int[4];

  myTable[0] = 1;
  myTable[1] = 2;
  myTable[2] = 3;
  myTable[3] = 4;
}
```

W pierwszej linii kodu deklarujemy zmienną referencyjną o nazwie `myTable` zdolną do przechowywania referencji do obiektów typu `int[]`. Tworzymy także obiekt tablicy, który przechowuje cztery zmienne typu `int`. Referencję tego obiektu przypisujemy do zadeklarowanej zmiennej. Elementy tablic zawsze inicjalizowane są domyślnymi wartościami, tak więc wszystkie cztery elementy tablicy będą miały wartość 0. Nie tego chcemy, więc w kolejnych liniach kodu przypisujemy inne wartości. Mamy co prawda to, czego chcieliśmy, ale powyższy sposób nie jest specjalnie wygodny – da się to samo zrobić dużo zgrabniej. Popatrzmy na kolejny przykład, który robi dokładnie to samo, tyle że z użyciem specjalnej składni inicjalizacji tablic:

```
public static void main(String[] args) {
  int[] myTable = {1, 2, 3, 4};
}
```

Do zmiennej `myTable` przypisujemy zatem wartość wyrażenia `{1, 2, 3, 4}`. Ponieważ jesteśmy w kontekście inicjalizacji zmiennej referencyjnej typu tablicowego kompilator wie, że chodzi o utworzenie tablicy. Po typie zmiennej referencyjnej rozpoznaje też, że chodzi o tablicę zmiennych typu `int`. Jedyne, czego jeszcze potrzebuję do utworzenia obiektu tablicy to rozmiar, ale to wiadomo na podstawie ilości wartości podanych w nawiasach klamrowych. Elementy tablicy inicjalizowane są właśnie tymi wartościami. Jak powiedzieliśmy na początku tego akapitu kompilator wie, że chodzi o utworzenie tablicy ponieważ jesteśmy w kontekście inicjalizacji zmiennej referencyjnej typu tablicowego. W ogólności,

nawiasy klamrowe obejmujące listę wartości nie są instrukcją konstrukcji tablicy. Z tego powodu język Java udostępnia jeszcze jedną konstrukcję, której znaczenie nie jest już zależne od kontekstu. Przykład poniżej:

```
public class Test {
  public static void main(String[] args) {
    printArray(new int[] {1, 2, 3, 4});
  }

  public static void printArray(int[] someArr) {
    for(int x : someArr)
      System.out.println(x);
  }
}
```

Zerknijmy na wywołanie metody `printArray(…)` w metodzie `main(…)`. Jak widać argumentem metody `printArray(…)` jest tablica. Wywołując tę metodę nie przekazujemy jednak referencji do uprzednio utworzonej tablicy, przeciwnie, tablicę tworzymy w momencie wywołania metody (naturalnie tworzona jest ona tuż przed, ale mowa o zapisie). Zwróćmy uwagę, że tworząc tablicę w ten sposób nie podajemy explicite jej wymiaru – nie ma w nawiasach kwadratowych żadnej liczby. Rozmiar tablicy jest dedukowany na podstawie liczby elementów wymienionych między nawiasami klamrowymi. Próba określenia wprost wymiaru tablicy kończy się błędem kompilacji. Za pomocą analogicznych składni możemy także inicjalizować tablice wielowymiarowe. Zerknijmy na poniższy przykład:

```
public static void main(String[] args) {
  byte[][] myTable = {{1, 2}, {1, 2, 3}, {1, 2, 3, 4}};

  for (byte[] table : myTable) {
    for (byte x : table)
      System.out.print(x + " ");

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

W powyższym przykładzie tablica `myTable` jest tablicą dwuwymiarową – mówiąc precyzyjniej jest to tablica tablic. Zawiera ona trzy tablice: `{1, 2}`, `{1, 2, 3}` oraz `{1, 2, 3, 4}`. Efektem uruchomienia programu będzie mozaika liczb pokazana poniżej:

```
1 2
1 2 3
1 2 3 4
```

Naturalnie, w analogiczny sposób możemy inicjalizować tablice obiektów. W szczególności, obiektami tymi mogą być tablice. Poniżej przykład:

```
public class Test {
  public static void main(String[] args) {
    Number[][] myTable =
        { new Byte[] {1, 2}, new Short[] {1, 2, 3}, {1, 2} };

    for (Number[] table : myTable) {
      for (Number x : table)
        System.out.print(x + " ");

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

ZMIENNE TYPÓW PROSTYCH I REFERENCYJNYCH

Zmienne typów prostych to pewna ilość bajtów pamięci – np. 4 bajty dla typu `int` – przechowujących explicite wartość zmiennej. Zmienne typu referencyjnego to również pewna ustalona ilość bajtów pamięci, ale zapisana tam wartość nie jest bezpośrednio dla nas użyteczna. Możemy o tej wartości myśleć jak o wskaźniku na obiekt, ale bynajmniej nie musi to być adres obszaru pamięci, w którym przechowywany jest obiekt. To, jaka dokładnie wartość jest przechowywana w zmiennej referencyjnej zależy od implementacji Wirtualnej Maszyny Javy której akurat używamy, a my potrzebujemy wiedzieć tylko tyle, że wartość ta jednoznacznie reprezentuje pewien obiekt, albo referencję pustą, czyli `null`. Zerknijmy na poniższy przykład:

```
class SomeClass {
  private String descr = "default value";

  public String getDescr() {
    return descr;
  }
```

```
   public void setDescr(String descr) {
     this.descr = descr;
   }
}

public class Test {
  public static void main(String[] args) {
    testPrimitives();

    testReferences();
  }

  static void testPrimitives() {
    int x = 1;
    int y = x;

    x = 2;

    System.out.println("y: " + y);
  }

  static void testReferences() {
    SomeClass a = new SomeClass();
    SomeClass b = a;

    a.setDescr("new value");

    System.out.println("b descr: " + b.getDescr());
  }
}
```

Przypatrzmy się metodzie `testPrimitives()`. Deklarujemy w niej zmienną `x` typu prostego `int` i przypisujemy wartość literału `1`. Powoduje to alokację 4 bajtów pamięci i zapisanie w tym obszarze bitowej reprezentacji liczby 1. Następnie deklarujemy zmienną `y` i przypisujemy jej wartość zmiennej `x` – wartość zmiennej `x` a nie „zmienną `x`", cokolwiek by to mogło znaczyć. Mamy zatem kolejne 4 bajty pamięci i zapisaną w nich bitową reprezentację liczby `1`, bo taka była wartość zmiennej `x`. Następnie do zmiennej `x` przypisujemy wartość literału `2`. Powoduje to zapisanie w obszarze pamięci, który reprezentuje zmienną `x` reprezentacji bitowej liczby `2`. Co dzieje się w tym czasie ze zmienną `y` i jej wartością? Nic! Cóż by się miało dziać? Zmienne te nie są ze sobą w żaden sposób powiązane. Aby się o tym przekonać wpisujemy na koniec wartość zmiennej `y` i widzimy, że rzeczywiście jej wartość pozostała niezmieniona – program wyświetla tekst „y: 1".

A teraz zerknijmy na metodę `testReferences()` – jest ona skonstruowana analogicznie do metody opisanej powyżej, ale operuje na zmiennych referencyjnych i obiektach. W pierwszej linii deklarujemy zmienną referencyjną `a`, co powoduje alokację przestrzeni pamięci potrzebną do reprezentacji „wskaźnika" do obiektu. Tworzymy także nowy obiekt, co pociąga za sobą w skutku wiele operacji, między innymi alokację pamięci niezbędnej do reprezentowania stanu obiektu oraz inicjalizację tego stanu. W końcu „wskaźnik" obiektu jest zapisywany jako wartość zmiennej referencyjnej. W drugiej linii kodu deklarujemy kolejną zmienną. Jako wartość zmiennej `b` przypisywana jest wartość zmiennej `a` a więc „wskaźnik" na obiekt utworzony uprzednio. Następnie, używając referencji `a` wywołujemy metodę `setDescr(…)` naszego obiektu… tego samego, na który wskazuje zmienna `b`; nic zatem dziwnego, że program wypisze „b descr: new value".

Pyt.13 **Załóżmy, że klasa *A* dziedziczy a klasy *B* a klasa *B* z klasy *C*. Deklarujemy zmienną *b* typu *B* i przypisujemy do niej instancję klasy *A* a następnie wywołujemy metodę *op(C c)* przekazując jako parametr tego wywołania zmienną *b*. Jakiego typu obiekty może potencjalnie wskazywać zmienna *b* tuż po powrocie z wywołania metody *op(C c)*?**

PARAMETRY METOD

Wywołując metodę przekazujemy pewną listę – potencjalnie pustą – parametrów typu prostego i/lub referencji do obiektów. Pierwszą i w zasadzie jedyną rzeczą, jaką w tym kontekście trzeba wiedzieć jest, że w języku Java wszystkie parametry są przekazywane przez wartość – reszta to już czysta konsekwencja tego prostego faktu. Przeanalizujmy to na przykładach:

```
public class Test {
  public static void main(String[] args) {
    int x = 2;

    triple(x);

    System.out.println("x: " + x);
  }
```

```
  static void triple(int x) {
    x *= 3;
  }
}
```

Efektem uruchomienia powyższego programu będzie wyświetlenie tekstu „x: 2”. Przeanalizujmy krok po kroku jak to się dzieje. W pierwszej linii metody `main(…)` deklarujemy zmienną `x` i przypisujemy jej wartość literału `2`. Następnie wywołujemy metodę `triple(…)` przekazując jako argument „zmienną `x`”. Co to dokładnie znaczy przekazać zmienną jako argument? Zerknijmy na funkcję `triple(…)`. Widzimy, że deklaruje ona argument typu `int` o nazwie `x` – naturalnie zbieżność nazwy ze zmienną zadeklarowaną w metodzie `main(…)` nie ma żadnego znaczenia. Zmienna zadeklarowana jako argument metody traktowana jest dokładnie tak samo jak zmienne zadeklarowane w jej ciele – jest to zmienna lokalna metody. Mamy zatem zmienną typu `int` o nazwie `x` w metodzie `main(…)` oraz inną zmienną o tej samej nazwie w metodzie `triple(…)`. Mimo że nazywają się one tak samo to są to zupełnie inne zmienne, tj. odpowiadają innym obszarom pamięci.

Odp.13 **Zmienna *b* tuż po powrocie z wywołania metody *op(C c)* zawsze będzie wskazywała dokładnie na ten sam obiekt na który wskazywała tuż przed wywołaniem metody *op(C c)* a więc na obiekt typu *A*. Metoda *op(C c)* nie ma możliwości zmiany tego przypisania.**

Jak już sobie powiedzieliśmy, zmienne w języku Java zawsze przekazywane są przez wartość, zatem wywołanie metody `triple(…)` powoduje skopiowanie wartości zmiennej `x` z metody `main(…)` do zmiennej `x` w metodzie `triple(…)`. I właśnie to skopiowanie jest przekazaniem parametru wywołania! Jaki zatem efekt ma wykonanie przypisania `x *= 3` w metodzie `triple(…)`? Naturalnie potrojenie wartości zmiennej `x` zadeklarowanej w metodzie `triple(…)`, ale nie zmiennej `x` z metody `main(…)`. Nic więc dziwnego, że program wypisze „x: 2” – wypisujemy przecież wartość zmiennej lokalnej z metody `main(…)` a ta nie zmieniła się od czasu inicjalizacji liczbą 2. W powyższym przykładzie celowo nazwałem obie zmienne tak samo; w końcu moją intencją było wprowadzenie w błąd, zupełnie tak jak to ma miejsce na egzaminie na OCPJP, na co trzeba się wyczulić. Zerknijmy na kolejny przykład:

```
public class Test {
  public static void main(String[] args) {
    SomeClass obj = new SomeClass();

    triple(obj);

    System.out.println("x: " + obj.getX());
  }

  static void triple(SomeClass obj) {
    obj.setX(obj.getX() * 3);
  }
}

class SomeClass {
  int x = 2;

  public int getX() {
    return x;
  }

  public void setX(int x) {
    this.x = x;
  }
}
```

Tym razem przekazujemy referencję do obiektu, a dokładniej, kopię tej referencji. Jest to bardzo duża różnica, co pokażę w kolejnym przykładzie. Referencje to nic nadzwyczajnego – jest to zwykła zmienna przechowująca pewną wartość, tyle że ta wartość to „wskaźnik" na obiekt. Przeanalizujmy, co się dzieje w przedstawionym powyżej programie. W pierwszej linii metody `main(…)` deklarujemy zmienną referencyjną oraz tworzymy nowy obiekt. Zmienna ta inicjowana jest wartością reprezentującą wskaźnik na ten obiekt. Wywołując metodę `triple(…)` nie przekazujemy bynajmniej obiektu, tylko referencję. Przekazanie referencji oznacza skopiowanie wartości zmiennej `obj` zadeklarowanej w metodzie `main(…)` do zmiennej o tej samej nazwie zadeklarowanej w metodzie `triple(…)`. Obie zmienne nazywają się tak samo i mają tę samą wartość, jednak są to zupełnie inne zmienne, których wartości możemy zmieniać niezależnie, analogicznie jak wartości zmiennych typów prostych. Ale co oznacza instrukcja `obj.setX(…)` umieszczona w metodzie `triple(…)`? Nie jest to bynajmniej przypisanie nowej wartości zmiennej `obj`. Jest to wywołanie metody obiektu „wskazywanego" przez tę zmienną. Ale chwileczkę… przecież zmienna `obj` lokalna dla metody

`triple(…)` ma tę samą wartość, co zmienna `obj` lokalna dla metody `main(…)`, zatem „wskazuje" dokładnie ten sam obiekt. Nic więc dziwnego, że uruchomienie programu spowoduje wyświetlenie tekstu „x: 6". Zmodyfikowaliśmy przecież dokładnie ten obiekt, który utworzyliśmy w metodzie `main(…)` i którego stan następnie wyświetlamy. I jeszcze jeden przykład – klasa `SomeClass` pozostała bez zmian, więc nie pokazuję jej ponownie:

```
public class Test {
  public static void main(String[] args) {
    SomeClass obj = new SomeClass();

    triple(obj);

    System.out.println("x: " + obj.getX());
  }

  static void triple(SomeClass obj) {
    SomeClass localObj = new SomeClass();

    localObj.setX(obj.getX() * 3);

    obj = localObj;
  }
}
```

Nadal przekazujemy referencję do obiektu – metoda `main(…)` wcale się nie zmieniła – a mimo to program wypisuje „x: 2". Dlaczego? Właśnie dlatego, że referencje zupełnie tak jak zmienne typów prostych przekazywane są przez wartość, tj. przekazywane są kopie referencji. Przypisanie `obj = localObj` modyfikuje zmienną lokalną dla metody `triple(…)` i nie ma żadnego wpływu na zmienną `obj` z metody `main(…)`.

OPERATORY PRZYPISANIA DLA TYPÓW PROSTYCH

Literały całkowite domyślnie są typu `int`, ale w języku Java mamy cztery typy całkowitoliczbowe: `byte`, `short`, `int` i `long`. Jeśli na końcu literału całkowitego dodamy literę `L` lub `l`, to będzie on miał typ `long`, ale nie ma sposobu by literał miał typ `byte` lub `short`. Jak więc przypisać wartość do zmiennej jednego z tych typów? Wykorzystując konwersję domyślną, np.:

```
byte b = -128; // poprawne dla wartości od -128 do 127
short s = 32767; // poprawne dla wartości od -32768 do 32767
```

Rozważmy analogiczną sytuację dla typów zmiennopozycyjnych: `float` i `double`. Literały zmiennopozycyjne są domyślnie typu `double`, ale możemy zapisać literał typu `float` dodając sufix `F` lub `f`. Zagadkowa niekonsekwencja – nie możemy wymusić, by dany literał był typu `byte` czy `short`, ale typu `float` już tak. Literały zmiennopozycyjne są domyślnie typu `double` a mimo to możemy explicite powiedzieć, że literał jest typu `double` dodając sufix `D` lub `d`. Nie możemy tego zrobić dla typu `int`, który jest domyślnym typem literałów całkowitych. To nie koniec niekonsekwencji. Spójrzmy na poniższy kod:

Pyt.14 **Zmienna *b* jest typu *boolean*. Czy skompiluje się program zawierający instrukcję *if(b = true);*?**

```
float f = 1.1; // błąd!
```

Byłoby jeszcze nieźle, gdyby nie to, że powyższy kod się nie skompiluje. Literał `1.1` jest typu `double` i kompilator zgłasza, że nie może wykonać implicite konwersji (potencjalnie) stratnej. Niby w porządku, ale nie protestuje na tej samej zasadzie w przypadku typów całkowitych. Powyższą niepoprawną instrukcję przypisania możemy naprawić na jeden z pokazanych poniżej sposobów:

```
float f = 1.1F; // może też być małe f zamiast F
float ff = (float) 1.1;
```

Ale typami całkowitoliczbowymi także nie możemy operować zupełnie beztrosko. Przejdźmy do typów wyrażeń arytmetycznych. Zerknijmy na poniższy przykład:

```
byte a = 1;
byte b = a + 1; // błąd!

byte c = 1 + 1;
byte d = 64 + 64; // błąd!

byte e = 1 + 1L; // błąd!
```

Zacznijmy od tego, że jeśli wyrażenie arytmetyczne używa tylko operandów

całkowitych to typ wyrażenia jest również całkowity i jest to zawsze `int` albo `long`. Typem takiego wyrażenia jest `int` jeśli żaden z operandów nie jest typu `long`. Jeśli choć jeden z operandów jest typu `long` to całe wyrażenie jest też typu `long`. Wyrażenie `1 + 1` jest zatem typu `int` i kompilator godzi się wykonać konwersję automatyczną do typu `byte`, ale wyrażenie `1 + 1L` jest już typu `long`, dla którego konwersja implicite nie jest przewidziana. Wartość wyrażenia, w którym nie występują zmienne jest wyliczana w czasie kompilacji. Wartością wyrażenia `64 + 64` jest 128 co jest poza zakresem wartości typu `byte` i stąd błąd w powyższym przykładzie. Ale czemu wartość wyrażenia `1 + 1` można przypisać do zmiennej typu `byte`, a wartości wyrażenia `a + 1` już nie? Oba te wyrażenia są typu `int`, ale w tym drugim występuje zmienna, a więc w czasie kompilacji nie wiadomo, jaka jest jego wartość. Skoro nie wiadomo jaka jest wartość to nie wiadomo czy jest ona z zakresu wartości typu `byte` i stąd błąd. Aby kod ten się skompilował należy zastosować explicite operację rzutowania. Z typami `short` oraz `char` – który można traktować także jak typ liczbowy – sytuacja jest analogiczna.

Odp.14 **Tak! Instrukcja ta, choć myląca, jest poprawna. Użycie operatora przypisania zamiast operatora równości w warunku jest częstym błędem. Program taki kompiluje się, ale niekoniecznie działa zgodnie z wolą programisty.**

Jeśli wyrażenie arytmetyczne zawiera choćby jeden operand typu `float` a nie zawiera operandów typu `double` to typem takiego wyrażenia jest też `float`. Jeśli wyrażenie arytmetyczne zawiera choćby jeden operand typu `double` to typem wyrażenia jest `double`. Analogiczne przypisanie jak pokazane w ostatnim przykładzie dla typu `byte` dla typu `float` jest poprawne. Poniższy kod kompiluje się:

```
float a = 1.1f;
float b = a + 1.2f;
```

Naturalnie wszystkie te problemy związane z konwersją wartości i typów liczbowych spowodowane są troską o poprawność kodu. Co się dzieje, jeśli zmusimy kompilator do wykonania konwersji poprzez rzutowanie, a wartość jest zbyt duża by móc być przypisana do zmiennej danego typu? Nastąpi konwersja

stratna, czyli w przypadku liczby, zostaną odrzucone jej początkowe bity. Zerknijmy na poniższy program:

```
public class Test {
  public static void main(String[] args) {
    byte x = (byte)128;

    int i = 1024;
    byte y = (byte)i;

    System.out.println("x = " + x + " y = " + y);
  }
}
```

Rzutowanie jest określone explicite, więc kompilator nie protestuje; program się kompiluje. Ale jaki jest efekt jego uruchomienia? Otrzymujemy tekst „x = -128 y = 0", a więc wartości naszych zmiennych są „nieco dziwne". Nie ma cudów i trzeba o tym pamiętać. Jeśli wartość jest zbyt duża by zmieścić się w danym typie, to się tam po prostu nie zmieści.

Język Java oferuje także złożone operatory przypisania, czyli: +=, −=, *= oraz /=. Przy inicjalizacji zmiennych nie mają zastosowania, ale przy późniejszych operacjach już tak. Poniższe fragmenty kodu są sobie równoważne, ale przewaga drugiego sposobu zapisu jest oczywista:

```
b = (byte) (b + 2); // rzutowanie jest konieczne!

b += 2;
```

Prawa strona instrukcji przypisania jest zawsze wyliczana jako pierwsza, tak więc wyrażenie `x *= 1 + 2` jest równoważne wyrażeniu `x = x * (1 + 2)` a nie `x = x * 1 + 2`.

OPERATORY ARYTMETYCZNE, LOGICZNE I RELACYJNE

Operatory arytmetyczne mogące pojawić się na egzaminie OCPJP to +, −, *, / i % (czyli reszta z dzielenia) oraz operatory inkrementacji i dekrementacji, tj. ++ oraz −−. Na komentarz zasługuje operator +. Oprócz tego, że jest on operatorem dodawania może być stosowany także jako operator konkatenacji

stringów. Jeśli choć jeden z operandów operatora + jest stringiem, to + nie jest operatorem dodawania, tylko operatorem konkatenacji. Miejmy na uwadze także kolejność wykonywania działań. Dla przykładu, instrukcja:

```
System.out.println("2 plus 2 to " + 2 + 2);
```

powoduje wyświetlenie tekstu:

```
2 plus 2 to 22
```

ale już instrukcja:

```
System.out.println("2 razy 2 to " + 2 * 2);
```

daje wynik oczekiwany, tj.:

```
2 razy 2 to 4
```

Pierwszeństwo mają zawsze operatory unarne, a więc `++` i `--`. Następnie wykonywane są operacje `*`, `/` oraz `%` a operatory + i − wiążą najsłabiej. Poza tym, jeśli kolejność wykonywania działań nie jest w pełni wyznaczona przez pierwszeństwo operatorów to są one wykonywane od lewej do prawej. W wyrażeniu `"2 plus 2 to " + 2 + 2` jako pierwsza wykona się więc operacja konkatenacji `"2 plus 2 to " + 2` której wynikiem jest string `2 plus 2 to 2`. Następnie wykonana będzie konkatenacja `"2 plus 2 to 2" + 2` a więc w rezultacie otrzymamy string `2 plus 2 to 22`. Ponieważ jednak operator `*` wiąże silniej niż + w wyrażeniu `"2 razy 2 to " + 2 * 2` w pierwszej kolejności zostanie wyliczona wartość dla `2 * 2`, a potem wykonana będzie konkatenacja `"2 razy 2 to " + 4`.
Na egzaminie na OCPJP pojawiają się także operatory logiczne. Są to `&`, `|` i `^` – które to operatory są operacjami logicznymi dla operandów typu logicznego a operacjami bitowymi dla operandów typu całkowitoliczbowego – oraz `&&`, `||` i `!`. Różnica między operatorami `&` i `|` oraz `&&` i `||` zastosowanymi do operandów typu logicznego jest taka, że te pierwsze wyliczają wartość wyrażeń logicznych zachłannie a te drugie leniwie (ang. short-circuit evaluation).

Operatory relacyjne <, <=, >, >= mogą być stosowane tylko do porównywania wartości typów numerycznych i typu `char`, przy czym za wartość dla typu `char` przyjmuje się kod Unicode znaku. Operatory `==` oraz `!=` można stosować dodatkowo do porównywania wartości typu logicznego i typów referencyjnych.

Pyt.15 **Jeśli zmienna *arr* jest referencją typu *Integer[]*, to jaka może być wartość wyrażenia *arr != null & arr.length > 0*?**

Operator `instanceof` służy do testowania, czy referencja odnosi się do obiektu danego typu. Referencja jest operandem lewostronnym a nazwa klasy bądź interfejsu prawostronnym. Mówiąc precyzyjniej, wynikiem wyrażenia `x instanceof SomeClass` jest `true` gdy `x` nie jest `null` i gdy `x` może być zrzutowane do typu `SomeClass`. A co jeśli `x` ma wartość `null`? Czy w rezultacie otrzymamy wyjątek `NullPointerException`? Nie! Wyrażenie `null instanceof SomeClass` ma wartość `false` i nic złego się nie dzieje. Trzeba jeszcze zaznaczyć, że operatora `instanceof` nie możemy użyć, jeśli nie ma szans na powodzenie, tj. jeśli typ referencji `x` nie należy do tej samej hierarchii dziedziczenia co klasa `SomeClass`. Nie spełnienie tego warunku jest błędem kompilacji. Przykład poniżej:

```
public class Test {
  public static void main(String[] args) {
    Integer x = 5;

      // błąd kompilacji! to nigdy nie może być prawdą
    if(x instanceof Long) {
      System.out.println("cuda się zdarzają");
    }
  }
}
```

ZAKRES WIDOCZNOŚCI ZMIENNYCH

Zakres widoczności zmiennej (ang. variable scope) to fragment kodu w którym zmienna ta jest zdefiniowana i może być użyta. Jest bardzo prawdopodobne, że na egzaminie OCPJP pojawią się pytania zawierające kod,

który jest błędny z powodu próby użycia zmiennych poza ich zakresem widoczności. Prześledźmy to zagadnienie na przykładach.

Pierwszy przykład pokazuje błąd często popełniany przez początkujących programistów, tj. próbę odwołania się do zmiennej instancji (bez użycia instancji) z metody statycznej:

```
public class Test {
  String str = "Some text";

  public static void main(String[] args) {
    System.out.println(str); // błąd!
  }
}
```

W metodzie `main(…)` – statycznej – próbujemy wyświetlić wartość zmiennej instancyjnej `str`, tyle że nie ma żadnej instancji. Metody statyczne to metody które uruchamiane są niezależnie od instancji, tj. w kontekście metody statycznej nie istnieje obiekt domyślny `this`. Powyższy kod – napisany poprawnie – powinien wyglądać tak:

Odp.15 **Wyrażenie to, jak każde wyrażenie logiczne, może mieć wartość *true* lub *false*, ale z uwagi na zastosowanie operatora zachłannego *&* jego wyliczenie może także skutkować zgłoszeniem wyjątku typu *NullPointerException*. Aby wyeliminować tę niepożądaną ewentualność należało by operator *&* zastąpić operatorem *&&*.**

```
public class Test {
  String str = "Some text";

  public static void main(String[] args) {
    System.out.println((new Test()).str);
  }
}
```

Następny przykład przedstawia funkcję, która wylicza zadaną potęgę dla zadanej podstawy. Patrząc na taki kod przypuszczalnie skupimy uwagę na algorytmie – analizując jego poprawność – i nie weźmiemy pod uwagę, że program się nawet nie skompiluje. Na egzaminie na OCPJP powinniśmy postępować inaczej. Oto kod:

```
static long power(long base, long exponent) {
  for(int x = 1, y = 0; y < exponent; y++) {
    x *= base;
  }

  return x; // błąd!
}
```

Zmienna `x` została zadeklarowana w pętli i do tejże pętli ograniczony jest zakres jej widoczności – instrukcja `return x` jest więc niepoprawna, gdyż próbuje odwołać się do zmiennej w tym miejscu już nieistniejącej.

MODYFIKATORY WIDOCZNOŚCI METOD I ZMIENNYCH

Metody i zmienne w deklaracji klas mogą mieć każdy z czterech zakresów widoczności. Trzy zakresy widoczności odpowiadają trzem słowom kluczowym: `public`, `protected` i `private` a czwarty to zakres domyślny, który obowiązuje, jeśli nie określimy żadnego z powyższych modyfikatorów.

Przyszła pora na to, by zastanowić się dokładniej nad tym, co to właściwie oznacza „widoczność". Otóż ma ona dwa aspekty. Po pierwsze interesuje nas, czy da się odwołać do metody lub zmiennej klasy za pośrednictwem jej instancji i po drugie, czy da się do nich odwołać z kodu klasy pochodnej, a więc czy podlegają dziedziczeniu. Poniższy fragment kodu ilustruje to zagadnienie:

```
public class Parent {
  protected int x = 5;
}

public class Child extends Parent {
  int inheritanceAccess() {
    return x;
  }
  int parentInstanceAccess() {
    Parent parent = new Parent();

      // poniższa instrukcja nie zawsze jest poprawna
    return parent.x;
  }
}
```

Tak jak zasygnalizowano w komentarzu, powyższy kod nie zawsze jest poprawny, a to czy jest zależy od tego czy obie klasy znajdują się w tym samym pakiecie. Więcej na ten temat za parę zdań, póki co ważne jest żeby zrozumieć dwojakość pojęcia widoczności.

Zmienne i metody oznaczone jako `private` są widoczne tylko i wyłącznie w klasie, która je definiuje i są widoczne zarówno przy dostępie „lokalnym" jak i dostępie za pośrednictwem innej instancji tej klasy. Zmienne i metody prywatne nie są dziedziczone. Poniżej przykład poprawnego kodu ilustrującego zagadnienie:

```
public class SomeClass {
   private int x = 5;

   int instanceAccess() {
     SomeClass instance = new SomeClass();

     return instance.x;
   }

   int thisAccess() {
     return x;
   }
}
```

Jeszcze słówko komentarza, aby ułatwić zrozumienie zagadnienia. Instrukcja `return x` to de facto instrukcja return `this.x` gdzie `this` to nic innego jak wskaźnik na instancję naszej klasy dla której wywołano metodę. Instrukcja `return instance.x` jest więc z punktu widzenia widoczności tym samym co `return this.x` – obie te instrukcje odwołują się do tej samej zmiennej poprzez instancję tej samej klasy i w tej samej klasie.

Zmienne i metody oznaczone jako `public` są widoczne mówiąc najogólniej wszędzie. Oczywiście trzeba jeszcze rozważyć, czy klasa, która definiuje taką metodę lub zmienną też jest widoczna – jeśli nie, to naturalnie metoda czy zmienna też nie jest. Metody i zmienne publiczne są dziedziczone i nie ma tu żadnych udziwnień.

Zmienne i metody nieoznaczone żadnym modyfikatorem widoczności, tj. o widoczności domyślnej są widoczne tylko w klasach z tego samego pakietu, co

klasa zawierająca deklaracje. Takie metody i zmienne są również dziedziczone tylko przez klasy z tego samego pakietu.

Z modyfikatorem `protected` jest najciekawiej – tj. najtrudniej. Modyfikator ten jest rozszerzeniem domyślnego zakresu widoczności w tym sensie, że również powoduje ograniczenie widoczności do pakietu, ale ograniczenie to nie dotyczy klas, które z klasy deklarującej metodę lub zmienną `protected` dziedziczą. Jeśli więc klasa `A` deklaruje metodę o widoczności `protected` to metoda ta jest dostępna w klasie `B` tylko i wyłącznie wtedy, gdy klasa `B` znajduje się w tym samym pakiecie albo, gdy dziedziczy (wprost lub przechodnio) z klasy `A`. Zakończmy przykładem obrazującym to zagadnienie:

```
public class Parent {
   protected int x = 5;
}

public class Child extends Parent {
   int parentInstanceAccess() {
     Parent parent = new Parent();

       // działa tylko jeśli klasy są w tym samym pakiecie
     return parent.x;
   }

   int thisInstanceAccess() {
     Child child = new Child();

       // to zawsze jest ok
     return child.x;
   }

   int inheritanceAccess() {

       // zawsze ok
     return x;
   }
}
```

DEKLARACJE METOD

Modyfikatory widoczności w kontekście metod opisano w poprzednim podrozdziale, zatem znaczenie słów kluczowych `private`, `public` i

`protected` w tym kontekście mamy już wyjaśnione. Pozostało jeszcze jednak kilka modyfikatorów których możemy użyć w deklaracji metod, są to: `final`, `abstract`, `synchronized`, `native`, `strictfp` oraz `static`.

Metoda abstrakcyjna, tj. zadeklarowana jako `abstract` to metoda nieposiadająca implementacji. Jeśli klasa zawiera choć jedną metodę abstrakcyjną to sama także musi być zadeklarowana jako `abstract`. Jeśli metoda jest zadeklarowana jako `abstract` to jednocześnie może być zadeklarowana tylko jako `public` lub `protected`, modyfikator `abstract` nie może być użyty w parze z żadnym innym modyfikatorem. Przy okazji uświadommy sobie jak to jest z metodami i klasami abstrakcyjnymi w świetle dziedziczenia. Załóżmy, że klasa `A` jest klasą abstrakcyjną i zawiera abstrakcyjne metody `a()` i `b()`. Jeśli klasa `B` dziedziczy z klasy `A`, to są dwie możliwości: albo klasa `B` również jest abstrakcyjna i wtedy może implementować dowolną, obie albo żadną z metod `a()` i `b()`, albo klasa `B` nie jest abstrakcyjna i wtedy musi implementować zarówno `a()` i `b()`. Ogólna zasada jest taka, że jeśli klasa nie abstrakcyjna dziedziczy z klasy abstrakcyjnej to musi implementować wszystkie abstrakcyjne metody odziedziczone z tej nadklasy.

Modyfikator `final` oznacza, że metoda nie będzie mogła być przedefiniowana w podklasie. Generalnie, modyfikator `final` jest po to właśnie, aby zagwarantować sobie, że cokolwiek zostało raz zdefiniowane i oznaczone jako `final` nigdy nie będzie się mogło zmienić. Zmiana w kontekście metody oznacza właśnie przedefiniowanie w podklasie. Ale w kontekście deklaracji metod modyfikator `final` pojawia się raz jeszcze. Otóż modyfikatorem tym możemy także oznaczyć parametr formalny metody i ma to ten skutek, że do zmiennej lokalnej określonej tym parametrem nie będzie można przypisać żadnej wartości. Nawiasem mówiąc, `final` jest jedynym modyfikatorem, jakim można oznaczyć zmienne lokalne, a więc i parametry metod. I jeszcze przypomnienie - tak jak już napisałem powyżej, modyfikator `final` nie może wystąpić razem z modyfikatorem `abstract`.

Metodę statyczną, nazywaną także metodą klasową, deklarujemy używając słówka kluczowego `static`. Jak już napisałem, modyfikator `static` nie może wystąpić w parze z modyfikatorem `abstract`.

Słówko kluczowe `synchronized` wskazuje, że oznaczona nim metoda może być wykonywana w danym czasie co najwyżej przez jeden wątek. Mówiąc precyzyjniej, metoda taka działa jak monitor (pojęcie z programowania współbieżnego, nie chodzi o monitor w sensie ekranu). Jeśli metoda jest statyczna to monitor ten synchronizuje się na obiekcie klasy a jeśli nie to na obiekcie, dla którego metoda została wywołana. Naturalnie, słówko kluczowe `synchronized` nie może wystąpić w parze ze słówkiem `abstract`.

Modyfikator `native` oznacza, że metoda została zaimplementowana w sposób charakterystyczny dla danej platformy, czyli najpewniej przy użyciu innego języka programowania, np. C czy C++. Deklarując metodę jako `native` kończymy jej deklarację średnikiem, zupełnie tak jakbyśmy deklarowali metodę abstrakcyjną. Jakże mogłoby być inaczej, w końcu nie będziemy przecież podawać implementacji w języku C. Naturalnie, metoda oznaczona modyfikatorem `native` nie może być jednocześnie `abstract`. Nie może być także `strictfp` jako że nie będzie przecież wykonywana przez Wirtualną Maszynę Javy. Modyfikator `native` może być użyty tylko i wyłącznie w deklaracji metody. W deklaracji klasy czy zmiennej nie! Tylko metody!

Pyt.16 **Klasa *A* zawiera prywatną metodę *opA()*. Czy możliwe jest wywołanie tej metody z metody *opB()* w klasie *B*?**

O `strictfp` już było pisane, ale przypomnijmy. Modyfikator `strictfp` oznacza, że wszystkie operacje zmiennoprzecinkowe w danej metodzie będą zgodne ze standardem IEEE 754, tj. Wirtualna Maszyna Javy wykona instrukcje zmiennoprzecinkowe w sposób zgodny z tym standardem. Modyfikator ten może być zastosowany przy deklaracji metody oraz klasy, ale nigdy przy deklaracji zmiennej. Jeśli klasa jest zadeklarowana jako `strictfp` to oznacza to tyle, że wszystkie jej metody oraz klasy zagnieżdżone również są `strictfp`. Także zmienne zadeklarowane w klasie są inicjalizowane zgodnie z arytmetyką `strictfp` (choć same zmienne nie mogą być `strictfp`). Modyfikator `strictfp` nie może być użyty razem z modyfikatorem `abstract` ani `native`.

I jeszcze jedna rzecz dotycząca deklaracji metod – metody ze zmienną liczbą parametrów (ang. var-args). Począwszy od Javy w wersji 5 możemy zadeklarować,

że metoda akceptuje dowolną liczbę argumentów pewnego ustalonego typu, przy czym dopuszczone są wszystkie typy, zarówno prymitywne jak i obiektowe. Możemy więc zadeklarować `fun(int… x)` co będzie oznaczało, że metoda akceptuje dowolną liczbę argumentów typu `int`. Składnia jest właśnie taka, że po nazwie typu umieszczamy trzy kropeczki, po czym podajemy nazwę zmiennej, która de facto będzie tablicą. Metoda może akceptować także inne parametry, czyli poprawną jest deklaracja `fun(char a, long… x)`. Ważne jest, że metoda może zawierać tylko jeden parametr tego rodzaju i że musi on być zadeklarowany na końcu listy parametrów. Poniższy kod ilustruje to zagadnienie:

```
public class AnyClass {
  void testFun() {
    fun("any", 1, 2, 3, 4);
  }

  void fun(String str, Integer... ints) {
    System.out.println(str + ints.length);

    for (Integer x : ints) {
     System.out.println(x);
    }
  }
}
```

PRZESŁANIANIE METOD

Odp.16 **Wbrew pozorom tak, ale tylko wtedy gdy klasa *B* jest zagnieżdżona w klasie *A*.**

Przesłonięcie (ang. overriding) metody to implementacja na nowo metody, którą odziedziczyliśmy z klasy nadrzędnej. Jeśli implementujemy interfejs czy dziedziczymy z klasy abstrakcyjnej to nie ma wyjścia, metody trzeba zaimplementować, jeśli natomiast dziedziczymy metody wraz z implementacją to mamy wybór: możemy zaakceptować tę implementację, lub dostarczyć nową, odpowiednią dla danej podklasy. Metoda przesłaniająca metodę odziedziczoną musi być z nią zgodna co do listy argumentów, modyfikatorów dostępu, deklarowanych wyjątków i zwracanego typu, jednak zgodna to nie oznacza identyczna. Reguły są następujące:

- Modyfikator dostępu dla metody przesłaniającej nie może być bardziej restrykcyjny niż metody przesłanianej. Musi być taki sam albo mniej restrykcyjny.

- Argumenty metody przesłaniającej muszą być dokładnie takie same jak argumenty metody przesłanianej.

- Zwracany typ metody przesłaniającej musi być albo taki sam, albo być podtypem typu zwracanego przez metodę przesłanianą – musi być z nim kompatybilny pod względem przypisania.

- Metoda przesłaniana nie może deklarować wyjątków kontrolowanych (ang. checked), które nie były zadeklarowane w metodzie przesłanianej, jednak może deklarować ich mniej – zbiór wyjątków deklarowanych przez metodę przesłaniającą musi być podzbiorem wyjątków deklarowanych przez metodę przesłanianą. Ale uwaga, tutaj znowu nie chodzi o to żeby były to dokładnie te same wyjątki. Mogą być także wyjątki będące podtypami tych zadeklarowanych w metodzie przesłanianej. Dla przykładu, poniższy kod jest poprawny, ponieważ zarówno `IOException` jak i `SQLException` są podklasami klasy `Exception`:

```
class Parent {
  void test() throws Exception { }
}

class Child extends Parent {
  void test() throws IOException, SQLException { }
}
```

  Niezależnie od listy wyjątków zdefiniowanych w metodzie przesłanianej w metodzie przesłaniającej możemy zadeklarować dowolną liczbę wyjątków niekontrolowanych (ang. unchecked).

- Naturalnie, nie można przesłonić metody, która została oznaczona jako finalna, tj. została oznaczona słówkiem kluczowym `final`. Właśnie do tego, by przesłaniania zabronić to słówko kluczowe służy.

- Nie można przesłonić metody statycznej, tj. oznaczonej słówkiem kluczowym `static`. Metody takie nie podlegają polimorfizmowi, więc nie miałoby to sensu.

Ale co to właściwie oznacza, że metoda redefiniująca musi spełniać określone wymagania? Pod jakim rygorem? Jaki jest skutek nieprzestrzegania tych zasad? Rozważmy następujące klasy:

```
class Parent {
  static String getDescr() {
    return "descr for Parent class";
  }
}

class Child extends Parent {
  static String getDescr() {
    return "descr for Child class";
  }
}
```

Czy metoda `getDescr()` została prawidłowo przesłonięta? Nie! Czy powyższy kod się skompiluje? Tak! O co więc chodzi? Ano o to, że metoda `getDescr()` w klasie `Child` jest zupełnie inną metodą niż metoda `getDescr()` w klasie `Parent`. Obie metody są zdefiniowane poprawnie, tyle że jedna nie przesłania drugiej. Są to zupełnie dwie różne metody. Metody statyczne nie podlegają przesłanianiu. Zerknijmy na kolejny przykład:

```
class Parent {
  String getDescr(String str) {
    return str;
  }
}

class Child extends Parent {
  String getDescr(Object obj) {
    return obj.toString();
  }
}
```

Tym razem metoda `getDescr()` z klasy `Parent` nie jest statyczna, więc można ją przesłonić, ale czy powyższy kod jest poprawnym przesłonięciem? Nie! Rzeczywiście, nie zgadza się lista argumentów; nie jest ona identyczna. Czy kod

się kompiluje? Tak! W jakim sensie zatem przedefiniowanie to jest niepoprawne? Otóż kod jest poprawny, ale nie jest to przesłonięcie (ang. overriding) tylko przeciążenie (ang. overloading). Skutek jest więc taki, że w klasie `Child` zdefiniowane są dwie metody o tej samej nazwie, ale różniące się argumentami. Więcej o przeciążaniu metod w kolejnym podrozdziale. I jeszcze ostatni przykład:

```
class Parent {
  String getDescr() {
    return "descr for Parent class";
  }
}

class Child extends Parent {
  String getDescr() throws Exception {
    return "descr for Child class";
  }
}
```

Czy jest to poprawne przesłonięcie? Nie! Czy kod się kompiluje? Niespodzianka – też nie! Metoda w klasie `Child` deklaruje wyjątek niezadeklarowany w metodzie z klasy `Parent`, więc nie jest to poprawne przesłonięcie. Nie jest to też przeciążenie, jako że metody nie różnią się listą argumentów, zatem nie pozostaje nic innego jak tylko zgłosić błąd kompilacji.

Implementując metodę przesłaniającą możemy wywołać metodę przesłanianą, tj. metodę z nadklasy, wywołując ją ze słówkiem kluczowym `super`. Przykład poniżej:

```
class Parent {
  String getDescr() {
    return "opis dla klasy Parent";
  }
}

class Child extends Parent {
  String getDescr() {
    return "opis dla klasy Child i " + super.getDescr();
  }
}
```

Na zakończenie odnotujmy jeszcze jeden fakt dotyczący zadeklarowanej listy wyjątków. Spójrzmy na poniższy kod:

```
class Parent {
  String getDescr() throws Exception {
    return "descr for Parent class";
  }
}

class Child extends Parent {
  String getDescr() {
    return "descr for Child class";
  }
}

class Other {
  void test() {
    Parent obj = new Child();

    obj.getDescr(); // nieobsługiwany wyjątek!
  }
}
```

Kod ten nie kompiluje się. Powodem jest nieobsługiwany wyjątek. Zmienna `obj` wskazuje na obiekt klasy `Child` którego metoda `getDescr()` nie deklaruje wyjątków, ale nie ma to znaczenia. Istotny jest typ zmiennej, a jest to `Parent`. Metoda `getDescr()` w klasie `Parent` deklaruje wyjątek i musi on być obsługiwany, jeśli posługujemy się zmienną tego typu.

PRZECIĄŻANIE METOD

Przeciążanie (ang. overloading) metod to definiowanie wielu metod o tej samej nazwie a różniących się listą argumentów. W zasadzie o metodzie przeciążającej – tj. używającej tej samej nazwy, co już istniejąca metoda – można myśleć jak o metodzie zupełnie nowej. Nie ma więc żadnych zasad ograniczających zwracany typ, deklarowane wyjątki czy modyfikatory dostępu, jak to ma miejsce w przypadku przesłaniania (ang. overriding). W zasadzie można by to sformułować następująco: jeśli chcemy zdefiniować metodę która będzie się nazywała tak samo jak inna, już istniejąca metoda to musimy określić inną listę argumentów. To tyle.

Mamy więc w naszej klasie zdefiniowanych kilka metod o tej samej nazwie, skąd więc wiadomo, którą z nich wywołujemy w danej chwili? Naturalnie, na podstawie

listy parametrów, tyle, że nie zawsze sprawa jest bardzo prosta. Weźmy dla przykładu następujący program:

```
class Test {
  public static void main(String[] args) {
    Test obj = new Test();

    System.out.println(obj.getString(1));
  }

  String getString(float f) {
    return "float value: " + f;
  }

  String getString(long l) {
    return "long value: " + l;
  }
}
```

Jaki jest efekt jego działania? Która z metod `getString()` zostanie wywołana? W tym wypadku efektem działania programu będzie tekst „long value: 1”. Mówiąc ogólnie… wywoływana jest ta metoda, której argumenty „bardziej odpowiadają” przekazywanym parametrom. A precyzyjniej… dla typów prostych zasady są następujące: jeśli nie ma metody odpowiadającej idealnie typom parametrów wywołania, to wybierana jest ta metoda, której argumenty są jak najmniej „większe”, przy czym „najmniejszy” jest typ `byte` i kolejno `short`, `int`, `long`, `float` i `double`. Parametry typu `char` traktowane są jakby były typu `int`.

Co będzie jednak, gdy damy kompilatorowi wybór między „większym” typem prostym a idealnie odpowiadającym typem opakowującym, tak jak to pokazano w kolejnym przykładzie:

```
class Test {
  public static void main(String[] args) {
    Test test = new Test();

    int i = 1;

    test.someOp(i);
  }

  void someOp(double x) {
    System.out.println("someOp(double x)");
  }
```

```
    void someOp(Integer x) {
      System.out.println("someOp(Integer x)");
    }
}
```

Kompilator wybierze „większy" typ prosty. Zasada jest taka: konwersja do bardziej „pojemnego" typu prostego ma pierwszeństwo przed opakowywaniem (ang. in-boxing). Tak samo jest z argumentami wielo-arnymi (ang. var-args, variable-arity arguments). Jeśli tylko jest taka możliwość, to kompilator wybierze metodę nie zawierającą wielo-arnych argumentów. A co, jeśli kompilator ma do wyboru tylko opakowanie typu prostego w instancję odpowiadającej mu klasy albo wywołanie metody z argumentem wielo-arnym? Zerknijmy na kolejny przykład:

```
class Test {
  public static void main(String[] args) {
    Test test = new Test();

    test.someOp(1L);
  }

  void someOp(long... x) {
    System.out.println("someOp(long... x)");
  }

  void someOp(Long x) {
    System.out.println("someOp(Long x)");
  }
}
```

Kompilator zawsze preferuje opakowanie (ang. in-boxing). Uruchomienie powyższego programu spowoduje wyświetlenie tekstu „someOp(Long x)". Podsumowując: jeśli tylko się da, kompilator wybiera konwersję typów prostych do innych, „pojemniejszych" typów prostych. W dalszej kolejności kompilator próbuje znaleźć metodę, która wymaga opakowania w obiekty a dopiero, gdy to zawiedzie, rozważa metody używające argumentów wielo-arnych. Popatrzmy na jeszcze jeden przykład:

```
class Parent { }

class Child extends Parent { }

class Test {
  public static void main(String[] args) {
    Parent obj = new Child();
```

```
    test(obj);
  }

  static void test(Parent p) {
    System.out.println("method for Parent");
  }

  static void test(Child c) {
    System.out.println("method for Child");
  }
}
```

Efektem działania będzie tekst „method for Parent" mimo, że obiekt dla którego wywołano metodę `test()` jest klasy `Child`. Powód jest prosty – to, która z przeciążonych metod zostanie wywołana jest określane w czasie kompilacji, kiedy to nie wiadomo jeszcze, jaki będzie rzeczywisty obiekt, a więc o wywoływanej metodzie decyduje typ referencji nie zaś typ obiektu.

REDEFINIOWANIE METOD STATYCZNYCH

Metody statycznej nie można przesłonić (ang. override), ale można ją zredefiniować (ang. redefine). Redefiniowanie metody, to definiowanie w podklasie metody statycznej, która została już zdefiniowana w nadklasie.

Pyt.17 **Czy metoda przesłaniająca pewną metodę z nadklasy może deklarować rzucanie wyjątku *NullPointerException*, jeśli metoda w nadklasie nie deklarowała żadnego wyjątku?**

Metody i zmienne statyczne nie przynależą do obiektu. Przynależą one do klasy i z tego względu powinny być wywoływane względem klasy, ale język Java dopuszcza także składnię, gdzie metodę statyczną wywołujemy względem zmiennej referencyjnej, a więc tak jakby względem obiektu. Poeksperymentujmy trochę z tą składnią. Przykład poniżej:

```
class SomeClass {
  static void printMessage() {
    System.out.println("my message");
  }
}
```

```
public class Test {
  public static void main(String[] args) {
    SomeClass obj = null;

    obj.printMessage();
  }
}
```

Jaki będzie rezultat uruchomienia tego programu? `NullPointerException`? Otóż nie! Rezultatem będzie wypisanie tekstu „my message". Dzieje się tak dlatego, że kompilator w miejsce zmiennej referencyjnej podstawia dla statycznych odwołań klasę tejże zmiennej. Instrukcja `obj.printMessage()` z powyższego przykładu jest de facto ekwiwalentem dla `SomeClass.printMessage()`. Nic więc dziwnego, że program wykona się poprawnie. Podkreślmy jeszcze to, co powiedziano być może nie dość wyraźnie – podstawiany jest typ zmiennej referencyjnej, a nie typ obiektu na jaki ta zmienna ewentualnie wskazuje. Wydaje się to stwierdzenie być dosyć trywialnym po tym jak powyżej pokazałem przykład, gdzie obiektu w ogóle nie było – do zmiennej `obj` przypisano `null` – ale to właśnie tego typu trywialne błędy pozbawiają nas punktów na egzaminie na OCPJP.

Odp.17 **Tak, *NullPointerException* jest wyjątkiem niekontrolowanym. Metoda przesłaniająca może deklarować rzucanie dowolnych wyjątków niekontrolowanych, niezależnie od tego czy zostały one zadeklarowane w metodzie przesłanianej.**

Jeśli chodzi o kryteria, jakie musi spełniać metoda by być metodą redefiniującą to są one takie same jak dla metod przesłaniających… naturalnie z tym wyjątkiem, że dotyczy to metod statycznych. Aby zrozumieć różnicę, między przesłanianiem a redefiniowaniem przeanalizujmy poniższy przykład:

```
class Parent {
  static String getDescr() {
    return "parent";
  }
}

class Child extends Parent {
  static String getDescr() {
    return "child";
  }
}
```

```
public class Test {
  public static void main(String[] args) {
    Parent obj = new Parent();

    System.out.println(obj.getDescr());

    obj = new Child();

    System.out.println(obj.getDescr());
  }
}
```

Rezultatem wywołania tego programu będzie dwukrotne wypisanie tekstu „parent". Tak jak napisano powyżej, dla odwołań statycznych w miejsce zmiennej referencyjnej podstawiana jest klasa tej zmiennej, a więc w obu przypadkach `Parent`. Instrukcja `obj.getDescr()` jest więc równoważna instrukcji `Parent.getDescr()` niezależnie od tego, na jaki obiekt wskazuje zmienna `obj`. I to jest właśnie przysłanianie. Usuńmy natomiast modyfikatory `static` z deklaracji metod `getDescr()`. Tym razem program wypisze „parent" a potem „child". I to jest istota nadpisywania.

DEKLARACJA KONSTRUKTORA

Każda klasa ma konstruktor. Jeśli nie zdefiniowaliśmy konstruktora explicite to jest on generowany przez kompilator – jest to tak zwany konstruktor domyślny. Konstruktor domyślny generowany jest tylko wtedy, gdy nie zdefiniowano żadnego explicite. Domyślny konstruktor, to konstruktor bezparametrowy. Jedyne, co robi konstruktor wygenerowany przez kompilator to wywołanie konstruktora bezparametrowego klasy nadrzędnej, a więc wykonanie instrukcji `super()`. Jeśli chcemy zmienić to zachowanie to możemy konstruktor bezargumentowy po prostu jawnie zaimplementować. Pytanie sprawdzające – czy poprawnym jest poniższy kod:

```
public class NewClass {
  private int x;

  public NewClass(int x) {
    this.x = x;
  }
}
```

```
public class OtherClass extends NewClass {

}
```

Nie! Dla klasy `OtherClass` nie zdefiniowano żadnego konstruktora a więc kompilator wygenerował konstruktor domyślny. Konstruktor domyślny – jak napisano powyżej – zawiera wywołanie `super()`. Dla klasy `NewClass` zdefiniowano konstruktor jednoargumentowy, więc kompilator nie wygenerował konstruktora domyślnego. Nie został też taki konstruktor zdefiniowany explicite a więc wywołanie `super()` znajdujące się w wygenerowanym dla klasy `OtherClass` konstruktorze domyślnym a będące wywołaniem nieistniejącego konstruktora bezargumentowego z nadklasy `NewClass` nie jest prawidłowe.

Konstruktor może mieć dowolną listę parametrów, która jest legalna dla zwykłych metod. Konstruktor deklarujemy analogicznie do tego jak deklarujemy metodę, tyle że nie podajemy zwracanego typu – jeśli określimy typ to jest to już metoda a nie konstruktor.

Konstruktor musi się nazywać dokładnie tak jak klasa, ale uwaga, metoda także może się nazywać dokładnie tak jak klasa. Zanim więc orzekniemy, że dana deklaracja jest konstruktorem – sugerując się nazwą – zwróćmy uwagę czy zadeklarowany jest zwracany typ.

Jeśli chodzi o modyfikatory, to dozwolone są wszystkie modyfikatory widoczności, jak i brak modyfikatora widoczności, co oznacza zasięg domyślny.

Konstruktory nigdy nie są dziedziczone. Znaczenie modyfikatorów widoczności, z dokładnością do dziedziczenia było już opisywane. Inaczej niż dla metod, dla konstruktorów nie są dozwolone żadne inne modyfikatory. Jeszcze tylko jeden komentarz co do widoczności. Zdawałoby się, że skoro konstruktory nie podlegają dziedziczeniu, to nie ma różnicy między zakresem `protected` i zakresem domyślnym. Poniższy przykład pokazuje, że jednak taka różnica jest. Jeśli poniższe klasy są w różnych pakietach, to kod jest poprawny dla konstruktora `protected`, a dla zakresu domyślnego już nie:

```
public class NewClass {
  protected NewClass() { }
}
```

```
public class OtherClass extends NewClass {
  public OtherClass() {
    super();
  }
}
```

KONSTRUKTORY I INICJALIZACJA

Każda klasa ma konstruktor. Podkreślmy to jeszcze raz – każda, w tym także abstrakcyjna! Konstruktory mają także typy wyliczeniowe, ale interfejsy nie. Po co konstruktor w klasie abstrakcyjnej, której instancji przecież nie można skonstruować? Wszystko wyjaśni się już za chwilę.

Konstruktory uruchamiane są, gdy tworzymy nowe obiekty a więc najczęściej w wyniku użycia operatora `new` (wyjątkiem są stałe wyliczeniowe). Jednak, aby utworzyć instancję danej klasy nie wystarczy wywołać konstruktora z tej klasy; trzeba wywołać także konstruktory z wszystkich nadklas, do klasy `Object` włącznie. Poniższy przykład pomoże nam zrozumieć, dlaczego:

```
abstract class Parent {
  private String str;

  Parent() {
    this.str = "some data";
  }

  public String getStr() {
    return str;
  }
}

class Child extends Parent { }

public class Test {
  public static void main(String[] args) {
    Child child = new Child();

    System.out.println(child.getStr());
  }
}
```

Co robi ten program? Oczywiście wypisuje tekst „some data”. Dzieje się tak tylko dlatego, że oprócz wygenerowanego przez kompilator, domyślnego konstruktora

z klasy `Child` wywołany został także konstruktor z klasy `Parent`. W celu zapewnienia poprawnej inicjalizacji obiektów obowiązuje prosta zasada – pierwszą instrukcją każdego konstruktora musi być wywołanie `super(…)` albo `this(…)` (z dowolną liczbą parametrów), przy czym jeśli nie umieściliśmy jednego z tych wywołań kompilator automatycznie doda wywołanie `super()` (bez parametrów).

Instrukcja `super(…)` to wywołanie konstruktora z nadklasy a `this(…)` to wywołanie innego, przeciążonego konstruktora z klasy bieżącej, aczkolwiek któryś z kolei konstruktor będzie musiał w końcu wywołać konstruktor z nadklasy. Ilustruje to poniższy przykład:

```
abstract class Parent {
  private String str;

  Parent(String str) {
    this.str = str;
  }

  public String getStr() {
    return str;
  }
}

class Child extends Parent {
  Child() {
    this("some text");
  }

  Child(String str) {
    super(str);
  }
}

public class Test {
  public static void main(String[] args) {
    Child child = new Child();
    System.out.println(child.getStr());

    child = new Child("another text");
    System.out.println(child.getStr());
  }
}
```

Efektem działania tego programu będą teksty kolejno „some text” i „another text”. Podkreślę jeszcze, że konstruktora nie można wywołać tak jak metody, posługując

się jego nazwą – zawsze posługujemy się konstrukcją `super(…)` lub `this(…)`. Konstruktora nie można także wywołać explicite inaczej jak tylko z innego konstruktora, tak więc instrukcje `super(…)` i `this(…)` nie są dozwolone w metodach. To, że instrukcje `super(…)` i `this(…)` muszą być pierwszymi instrukcjami w konstruktorze oznacza również, że w każdym konstruktorze możemy użyć tylko jednej z nich i tylko co najwyżej raz. W przeciwnym wypadku któraś z nich nie byłaby pierwsza. Skądinąd, wielokrotne występowanie tych instrukcji nie miałoby sensu.

Wywołanie przeciążonego konstruktora z klasy bieżącej, albo konstruktora z nadklasy musi być bezwzględnie pierwszą instrukcją i dopóki instrukcja ta nie zostanie wykonana obiekt nie jest jeszcze skonstruowany. Z tego względu, dopóki nie zostanie wykonana instrukcja `super(…)` nie jest możliwe wykonywanie operacji na właśnie konstruowanej instancji. Mówiąc wprost, dopóki nie zostanie wykonana (niewywołana, wykonana!) instrukcja `super(…)` nie można wywoływać nie statycznych metod ani używać nie statycznych zmiennych. Parametrami tych wywołań mogą być natomiast zmienne statyczne i wartości zwracane z wywołań statycznych metod. Poniżej przykład, który przy okazji pokazuje, jak wykonać pewne operacje przed wywołaniem super-konstruktora:

```
class Child extends Parent {
  Child() {
    super(doSomethingBefore());
  }

  static String doSomethingBefore() {
    System.out.println("before call to super()");
    return "some text";
  }
}
```

I jeszcze słówko, co do domyślnego konstruktora generowanego przez kompilator w przypadku, gdy programista nie zapewnił żadnego. Oprócz tego, że jest to konstruktor bez argumentowy i zawierający jedynie bezargumentowe wywołanie `super()` trzeba wiedzieć, że ma on modyfikator dostępu taki jak jego klasa. I ostatni przykład:

```
class Parent {
  String str = "some value";
}
```

```
class Child extends Parent {
  String childStr = "some other value";
}
```

Jak można się domyślić chodzi o inicjalizację zmiennych instancyjnych. Instrukcje przypisania wartości tym zmiennym też muszą się przecież kiedyś i gdzieś wykonać. I jak najbardziej, wykonują się one jako część konstruktora. Aby zrozumieć, kiedy to się dzieje prześledźmy kolejne operacje wykonywane w efekcie utworzenia nowej instancji klasy `Child`, tj. wywołania `new Child()`. Oto one:

- Wywoływany jest domyślny konstruktor klasy `Child`. Jego pierwszą instrukcją jest wywołanie `super()`.

- Wywoływany jest domyślny konstruktor klasy `Parent`. Jego pierwszą instrukcją jest wywołanie `super()`.

- Wywoływany jest konstruktor w klasie `Object`, ta nie ma już nadklas, więc po wykonaniu kodu konstruktora następuje powrót do konstruktora z klasy `Parent`.

- Wykonywana jest inicjalizacja zmiennych instancyjnych w klasie `Parent`, a więc przypisanie `str = "some value"`, po czym następuje powrót do konstruktora z klasy `Child`.

- Wykonywana jest inicjalizacja zmiennych instancyjnych w klasie `Child`, a więc przypisanie `childStr = "some other value"`, co jest ostatnim krokiem. Obiekt został utworzony i zainicjalizowany.

BLOKI INICJALIZACYJNE

Bloki inicjalizacyjne (ang. initialization blocks) występują w dwu wariantach: jako bloki instancyjne i bloki statyczne. Blok inicjalizacyjny to fragment kodu, który wykonuje się – w przypadku bloków statycznych – gdy

ładowana jest klasa, bądź – w przypadku bloków instancyjnych – gdy tworzona jest instancja. Zerknijmy na poniższy przykład:

```
public class Test {
  public Test() {
    System.out.println("constructor");
  }

  static {
    System.out.println("static init block");
  }

  {
    System.out.println("instance init block");
  }

  public static void main(String[] args) {
    System.out.println("start main");

    Test test = new Test();

    System.out.println("end main");
  }
}
```

Kod znajdujący się między konstruktorem a metodą `main(…)` to właśnie bloki inicjalizacyjne. Ten oznaczony słówkiem kluczowym `static` to blok statyczny. Ten drugi to blok instancji. Bloki statyczne – jak już powiedzieliśmy – wykonywane są w momencie ładowania klasy a więc zawsze jest to kod, który wykona się jako pierwszy.

O blokach instancyjnych powiedzieliśmy sobie, że wykonują się w momencie tworzenia instancji, ale w momencie tworzenia instancji klasy wykonują się też konstruktory. Jaka jest zatem kolejność? Wpierw konstruktor czy blok inicjalizacyjny? Otóż ani to, ani to. Jak wiemy z poprzedniego podrozdziału pierwszą instrukcją konstruktora jest zawsze wywołanie konstruktora nadklasy albo innego konstruktora tej samej klasy. Właśnie zaraz po tym wywołaniu – przed wykonaniem jakiejkolwiek innej instrukcji konstruktora – wykonywany jest kod bloków inicjalizacyjnych instancji. Efekt uruchomienia powyższego programu będzie zatem taki:

```
static init block
start main
```

```
instance init block
constructor
end main
```

W poprzednim akapicie pisałem o blokach inicjalizacyjnych w liczbie mnogiej i bynajmniej nie był to mój błąd. W każdej klasie może występować dowolna ilość bloków inicjalizacyjnych. O kolejności ich wykonania decyduje kolejność w jakiej są one zapisane, patrząc od góry do dołu. Spójrzmy na kolejny przykład:

```
class SuperClass {
  public SuperClass() {
    System.out.println("superclass constructor");
  }

  {
    System.out.println("superclass instance init block");
  }
}

public class Test extends SuperClass {
  public Test() {
    System.out.println("constructor");
  }

  static {
    System.out.println("static init block");
  }

  {
    System.out.println("instance init block #1");
  }

  {
    System.out.println("instance init block #2");
  }

  public static void main(String[] args) {
    Test test = new Test();
  }
}
```

Jak widzimy pojawiła się większa ilość bloków inicjalizacyjnych oraz nadklasa. Przekonajmy się, że kolejność wywoływania kodu bloków inicjalizacyjnych versus konstruktory jest taka właśnie jak to opisałem. Uruchomienie tego programu spowoduje wypisanie następującej sekwencji:

```
static init block
superclass instance init block
superclass constructor
instance init block #1
instance init block #2
constructor
```

A co się stanie, jeśli kod bloku inicjalizującego spowoduje wyjątek, np. odwołamy się do niewłaściwego indeksu tablicy? Wyjątek zostanie zwrócony… w postaci opakowanej w `java.lang.ExceptionInInitializerError`.

IN-BOXING I OUT-BOXING

Pyt.18 **Jeśli implementując klasę *B* nie zadeklarowano dziedziczenia z żadnej innej klasy, oraz nie zaimplementowano w niej żadnego konstruktora, to czy tworzenie instancji klasy *B* spowoduje wywołanie jakichś konstruktorów?**

Co to jest in-boxing i out-boxing wyjaśnimy sobie na przykładzie. Poniżej pokazana klasa implementuje dwie metody równoważne pod względem wykonywanych operacji, z tą różnicą, że druga z nich używa in- oraz out- (auto-) boxingu a pierwsza nie:

```
public class Test {
  private Integer counter;

  public void noAutoBoxing() {
    if(counter == null)
      counter = new Integer(0);

    counter = new Integer(counter.intValue() + 1);
  }

  public void useAutoBoxing() {
    if(counter == null)
      counter = 0; // in-boxing

    counter++; // out-boxing, inkrementacja, in-boxing
  }
}
```

Zaobserwujmy przy okazji jedną ciekawą rzecz – dzięki zastosowaniu typu obiektowego dla zmiennej `counter`, tj. `Integer` a nie `int` mamy możliwość

odróżnienia zmiennej niezainicjalizowanej od zmiennej posiadającej wartość 0 (czy inną wybraną wartość specjalną).

Metoda `equals(…)` dla typów opakowujących działa zgodnie z oczekiwaniami, tj. obiekty uważane są za równe, jeśli są tego samego typu i reprezentują tę samą wartość, ale za to operatory równości (==) i różności (!=) zadziwiają. Zerknijmy na poniższy przykład:

```
public class Test {
  public static void main(String[] args) {
      // odpowiada instrukcji 'Integer a = new Integer(1024);'
    Integer a = 1024;
    Integer b = 1024;

    if(a != b)
      System.out.println("a i b to różne obiekty");

    Integer c = 16;
    Integer d = 16;

    if(c == d)
      System.out.println("c i d to ten sam obiekt");
  }
}
```

O dziwo, zostaną wypisane obydwie sentencje, tj. „a i b to różne obiekty” oraz „c i d to ten sam obiekt”. Hmmm? Zgadza się! Jest to zgodne ze specyfikacją języka, a powodem jest jak można się domyślać chęć optymalizacji zużycia pamięci. Zasada jest taka: jeśli opakowywana (przez automatyczny in-boxing) jest wartość typu `boolean`, `byte` lub `short` czy `integer` z przedziału -128 do 127, albo `char` z przedziału \u0000 do \u007f to używana jest dla danego typu i danej wartości zawsze ta sama instancja obiektu – to wyjaśnia dlaczego „c i d to ten sam obiekt”. Co do tego, że „a i b to różne obiekty” to sprawa nie jest jasna. W specyfikacji języka Java wcale nie jest powiedziane, że `a` i `b` muszą odwoływać

Odp.18 **Tak! W klasie *B* zostanie wygenerowany konstruktor domyślny, którego jedyną instrukcją będzie wywołanie konstruktora z klasy *Object*, tak więc każdorazowo przy tworzeniu instancji klasy *B* wywołane będą dwa konstruktory – konstruktor z klasy *B* oraz konstruktor z klasy *Object*.**

się do różnych obiektów – przeciwnie, powiedziane jest, że dobrze byłoby gdyby te same wartości zawsze dawały tę samą instancję, ale nie muszą.

ZAGADNIENIA PROGRAMOWANIA OBIEKTOWEGO

Niniejszy rozdział pokrywa prezentowanym zakresem wiedzy następujące wymagania egzaminu 1Z0-851:

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

7.3 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.

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

HERMETYZACJA, ZALEŻNOŚĆ, SPÓJNOŚĆ

Znajomość języka Java, umiejętność definiowania klas czy interfejsów to tylko narzędzie programisty. O końcowym efekcie pracy, oprócz jakości i stopnia znajomości samego narzędzia decyduje jednak sposób, w jaki narzędzie to będzie używane. Java jest narzędziem a zasady programowania obiektowego określają, w jaki sposób należy tegoż używać, dla osiągnięcia jak najlepszych rezultatów.

Hermetyzacja (ang. encapsulation), zależność (ang. coupling) i spójność (ang. cohesion) to pojęcia którymi opisujemy jakość programu obiektowego. Zależność powinna być możliwie mała (ang. loose coupling) a spójność możliwie duża (ang. high cohesion). Klasy powinny być także hermetyczne (ang. tight encapsulation).

Hermetyczność oznacza, że klasy powinny ukrywać swoje wewnętrzne struktury i operacje a udostępniać tylko te, dla których udostępnienia zostały powołane, tj. swoje dobrze określone interfejsy. Dla przykładu, jeśli dla implementacji wymaganych mechanizmów posłużyliśmy się pewną strukturą danych, albo kilkoma strukturami to nie powinny być one dostępne z zewnątrz; jest bowiem wewnętrzną sprawą klasy, jak jej mechanizmy zostały zaimplementowane. Poprzez nie ujawnianie tych szczegółów chcemy zagwarantować, że sposób implementacji będzie można w przyszłości zmienić bez obawy o konieczność modyfikacji innych klas. Tutaj uwaga! Niska hermetyczność klas to tylko potencjalne ryzyko pojawienia się nieprzewidzianych i niepożądanych zależności, natomiast stan faktyczny istnienia takowych to zupełnie inna sprawa.

W języku Java hermetyzację osiągamy poprzez odpowiednie stosowanie modyfikatorów dostępu – w miarę możliwości jak najbardziej restrykcyjne.

Przejawem dążenia do wysokiej hermetyzacji są też właściwości (ang. properties) znane ze specyfikacji JavaBeans – wszystkie zmienne instancyjne powinny być prywatne i z zewnątrz dostępne poprzez metody typu get i set.

Klasy powinny być hermetyczne także dlatego, aby nie było możliwe tworzenie instancji reprezentujących niespójny, niepożądany stan. Rozważmy poniższy przykład, klasę reprezentującą wniosek kredytowy:

```
class LoanApplication {
  public float profitMargin;

  public long amount;

  public LoanApplication(long amount) {
    if(amount < Config.AMOUNT_LOW) {
      profitMargin = Config.PROFIT_HIGH;
    } else {
      profitMargin = Config.PROFIT_LOW;
    }
  }
}
```

Intencją powyższego kodu jest, aby wnioski kredytowe opiewające na małą sumę skutkowały wysoką marżą a dopiero powyżej pewnej kwoty marża byłaby odpowiednio niższa. Jednak, ponieważ zmienna `profitMargin` jest publiczna, możliwa jest dowolna zmiana jej wartości zaraz po utworzeniu obiektu. Wbrew intencjom legalny jest więc poniższy kod:

```
class GoodTeller {
  private LoanRegistry loanRegistry;

  public LoanID submitApplication(long amount, Customer cust) {
    LoanApplication loanApplication = new LoanApplication(amount);

      // przypisanie nie powinno być możliwe
    loanApplication.profitMargin = 0;

    if(cust.isTrustworthy()) {
      return loanRegistry.grantLoan(loanApplication, cust);
    } else {
      return null;
    }
  }
}
```

Klasy nie powinny być we wzajemnych skomplikowanych zależnościach. Kod jednej klasy nie powinien wykorzystywać cech innej klasy, które wynikają z jej konkretnej implementacji a nie są elementem zaprojektowanego interfejsu. Zagadnienie to jest ściśle powiązane z hermetyzacją w tym sensie, że skomplikowane zależności mogą wystąpić tylko wtedy, kiedy tej hermetyzacji brakuje. W kontekście egzaminu na OCPJP (choć oczywiście nie tylko) ważne jest, aby zrozumieć różnicę między stanem faktycznym istnienia ścisłych zależności a ryzykiem wystąpienia takowych, które powstaje jeśli brak hermetyzacji. Brak

hermetyzacji jedynie umożliwia zaistnienie tych zależności, ale niczego nie implikuje.

Wymóg spójności oznacza, że klasy powinny implementować dobrze określony, wąski zakres funkcjonalności. Oznacza to, że tam gdzie tylko ma to sens należy używać delegacji odpowiedzialności. Zerknijmy na poniższy przykład:

```
class Employee {
  private Company company;

  private int salary;

  void requestPayRise(int amount) {
     salary += company.requestPayRise(this, amount);
  }
}

class Company {
  private Boss boss;

  int requestPayRise(Employee emp, int amount) {
     return boss.requestPayRise(emp, amount);
  }
}

class Boss extends Employee {
  int requestPayRise(Employee emp, int amount) {
     if(emp instanceof Boss) {
          // szef dostanie podwyżkę o jaką prosił
        return amount;
     } else {
          // a zwykły pracownik… nie
        return 0;
     }
  }
}
```

Zwróćmy uwagę na klasę `Employee`. Jej metoda `requestPayRise()` deleguje wykonanie właściwych operacji do klasy `Company` a ta z kolei dalej do klasy `Boss`. I jest to słuszne, to w końcu nasz szef decyduje, czy dostaniemy podwyżkę czy nie.

WIELODZIEDZICZENIE

Język Java nie wspiera wielodziedziczenia klas, tj. klasa może dziedziczyć jednocześnie co najwyżej z jednej nadklasy. Powód jest prosty – chęć uniknięcia problemów. Wyobraźmy sobie, że zarówno klasa `A` jak i `B` implementują metodę `anyOp()`. Jeśliby teraz dopuścić wielodziedziczenie, to moglibyśmy zadeklarować klasę `C`, która rozszerza zarówno `A` i `B` co prowadzi do pytania – którą implementację metody `anyOp()` mamy w klasie `C`? Tę z klasy `A` czy z `B`? Jak widać, problem wynika z konieczności wyboru implementacji, a więc nie dotyczy interfejsów. Tak, język Java dopuszcza wielodziedziczenie interfejsów – interfejs może rozszerzać dowolną liczbę innych interfejsów. Klasa może rozszerzać co najwyżej jedną klasę ale jeśli chodzi o implementację interfejsów nie ma ograniczeń. Klasy mogą implementować dowolną liczbę interfejsów.

POLIMORFIZM

Zacznijmy od paru prostych konstatacji. Obiekty to są instancje klas. Do obiektów tych odwołujemy się za pomocą zmiennych, które są de facto referencjami. Każda referencja, tj. zmienna, ma swój typ. Raz zadeklarowana zmienna nie może później zmieniać swego typu, ale może wskazywać różne obiekty, także obiekty różnych typów – tych typów, które są zgodne pod względem przypisania (ang. assignment compatible) z typem zmiennej. Typ zmiennej może być określony poprzez klasę, interfejs lub typ wyliczeniowy.

Jeśli `A` jest pewną klasą, to zgodne pod względem przypisania ze zmienną typu `A` są obiekty klasy `A` oraz jej klas pochodnych, zaś jeśli `A` jest interfejsem to obiekty klas implementujących ten interfejs. Dla przykładu `A a = new B()` jest poprawnym przypisaniem, jeśli np. `B extends A` albo `B implements A`, albo jeśli `B` rozszerza czy też implementuje `A` nie bezpośrednio. Typy wyliczeniowe są tutaj jak klasy, z dokładnością do tego, że nie mogą one rozszerzać innych typów wyliczeniowych. I jeszcze jedno – to typ zmiennej, a nie typ faktycznego obiektu określa jakie operacje możemy wykonać dla obiektu za pośrednictwem tej zmiennej. Zerknijmy na poniższy przykład:

```
class Parent {
   void testFromParent() {}
}

class Child extends Parent {
   void testFromChild() {}
}

class Test {
   void test() {
     Parent p = new Child();

     p.testFromParent();

       // błąd!
     p.testFromChild();
   }
}
```

Zmienna `p` w powyższym przykładzie jest typu `Parent` a więc nie możemy dla niej wywołać metody `testFromChild()`, mimo że rzeczywisty obiekt wskazywany przez zmienną `p` jest typu `Child` i jego klasa posiada implementację tej metody. Gdybyśmy natomiast powiedzieli kompilatorowi, że zmienna `p` wskazuje na obiekt klasy `Child` stosując rzutowanie, to byłoby już poprawnie. Instrukcję `p.testFromChild()` trzeba więc w powyższym przykładzie zastąpić przez `((Child) p).testFromChild()`. Zwróćmy jeszcze uwagę na nawiasy przy rzutowaniu – wszystkie są konieczne, a może ich zabraknąć w godzinie próby, tj. w czasie egzaminu na OCPJP. Nie dajmy się na to nabrać.

Pyt.19 **Czy to typ zmiennej, czy może typ referencji decyduje o tym jakie metody możemy wywołać dla danego obiektu?**

O tym, jakie metody możemy wywołać z użyciem danej referencji decyduje typ referencji a nie typ wskazywanego przez nią obiektu, ale to, która implementacja danej metody zostanie wybrana zależy już od faktycznie wskazywanego obiektu. Zerknijmy na przykład:

```
class A {
   void test() {
     System.out.println("A");
   }
}
```

```
class B extends A {
   void test() {
     System.out.println("B");
   }
}

public class Test {
   public static void main(String[] args) {
     A var = new A();
     var.test();

     var = new B();
     var.test();
   }
}
```

Odp.19 **Decyduje oczywiście typ referencji. Referencja może wskazywać na obiekty różnych typów (zgodnych pod względem przypisania) a instrukcja wywołania metody musi być poprawna w każdym przypadku, niezależnie od typu obiektu faktycznie wskazywanego przez tę referencję.**

Uruchomienie powyższego programu spowoduje wypisanie litery „A" a następnie litery „B", mimo że dwukrotnie wywołano tę samą instrukcję `var.test()`, a więc zdawać by się mogło tę samą metodę. To jest właśnie istota polimorfizmu! Polimorfizm to zachowanie polegające na tym, że wybór metody która zostanie wykonana zależy nie od typu zmiennej tylko od typu faktycznego obiektu wskazywanego przez tę zmienną. W powyższym przykładzie dwukrotnie wywołaliśmy metodę `test()` na zmiennej `var` typu `A` a jednak za drugim razem wywołała się metoda zdefiniowana w klasie `B`. Właśnie dlatego, że za drugim razem zmienna `var` wskazywała na obiekt klasy `B`. A teraz uwaga! Polimorfizm dotyczy tylko metod i to tylko metod instancji, tj. nie statycznych. Spróbujmy dopisać słówko kluczowe `static` w deklaracjach metod `test()` w klasach `A` i `B`. Polimorfizm znika – program wypisuje dwukrotnie literę „A", przestaje się bowiem liczyć typ obiektu, ważny jest typ zmiennej. I jeszcze jeden przykład. Zmodyfikujmy klasy `A` i `B` tak jak pokazano poniżej:

```
class A {
   String id = "A";

   void test() {
     System.out.println(id);
   }
}
```

```
class B extends A {
   String id = "B";
}

public class Test {
   public static void main(String[] args) {
     A var = new A();
     var.test();

     var = new B();
     var.test();
   }
}
```

Jaki będzie teraz efekt działania programu? Będzie to dwukrotnie litera „A”. Na koniec jeszcze jeden przykład. Zastąpmy klasy `A` i `B` następującymi:

```
class A {
   void test() {
     System.out.println(getId());
   }

   String getId() {
     return "A";
   }
}

class B extends A {
   String getId() {
     return "B";
   }
}

public class Test {
   public static void main(String[] args) {
     A var = new A();
     var.test();

     var = new B();
     var.test();
   }
}
```

Tym razem program wypisze „A” a potem „B”. Metoda `getId()` wykonuje się przecież dla obiektu `this`, a obiektem tym za drugim razem jest obiekt klasy `B`.

RZUTOWANIE REFERENCJI

Wiemy już, że typ zmiennej referencyjnej i typ obiektu wskazywanego przez tę zmienną nie koniecznie są tym samym typem – typ obiektu może być także podtypem zmiennej, albo dowolnym typem implementującym, jeśli zmienna ma typ pewnego interfejsu. Zerknijmy na poniższy przykład:

```
class Parent {
  void doSomething() { }
}

class Child extends Parent {
  void doMore() { }
}

public class Test {
  public static void main(String[] args) {
    Parent obj = new Child();

    obj.doSomething();
    ((Child)obj).doMore();
  }
}
```

Zmienna `obj` jest typu `Parent`, ale wskazuje na obiekt typu `Child`. Jak już wiemy, o tym jakie metody możemy wywołać na danym obiekcie decyduje typ referencji, a nie typ obiektu. Aby więc wywołać metodę `doMore()` zdefiniowaną w klasie `Child` trzeba wykonać rzutowanie (ang. casting), tak jak to pokazano w powyższym przykładzie. Jeśli byśmy tego rzutowania nie wykonali, to kod by się nie skompilował, byłby to więc błąd kompilacji. Ale z rzutowaniem związane są także błędy wykonania. Rozważmy następującą modyfikację klasy `Test`:

```
public class Test {
  public static void main(String[] args) {
    Parent obj = new Parent();

    ((Child)obj).doMore(); // błąd wykonania!
  }
}
```

W czasie kompilacji wszystko jest w porządku, natomiast uruchomienie programu skutkuje rzuceniem wyjątku `java.lang.ClassCastException`.

Rzeczywiście, obiekt wskazywany przez zmienną `obj` jest klasy `Parent` a nie `Child`. Kompilator zakłada, że wiemy co robimy wykonując rzutowanie i ślepo ufa, że w czasie wykonania zmienna będzie wskazywała na obiekt odpowiedniego typu. Nie ma zresztą innego wyjścia, wykonanie takiego sprawdzenia w czasie kompilacji było by często zbyt skomplikowane, a nawet nie możliwe… aczkolwiek możliwa jest statyczna weryfikacja w pewnym zakresie. Kompilator zaprotestuje, jeśli wskazane rzutowanie nie ma szans na powodzenie, tj. jeśli typ rzutowania nie jest podtypem albo nadtypem typu zmiennej. Ilustruje to poniższy przykład:

```
public class Test {
  public static void main(String[] args) {
    Number num = new Integer(1);

    String str = ((String)num); // błąd kompilacji!
  }
}
```

Obiekt typu `Number` oraz żaden jego podtyp nigdy nie może być potraktowany jako `String`, tak więc powyższy kod skutkuje błędem kompilacji.

Pyt.20 **Dlaczego metody statyczne nie podlegają polimorfizmowi?**

Myśląc o rzutowaniu zwykle mamy na myśli rzutowanie uszczegółowiające (ang. downcasting), tj. rzutowanie zmiennej do pewnego podtypu, ale możliwe jest też rzutowanie uogólniające (ang. upcasting) do typu nadrzędnego, tak jak to pokazano poniżej, choć nie ma ono większego sensu:

```
public class Test {
  public static void main(String[] args) {
    Number num = new Integer(1);

    ((Object)num).toString();
  }
}
```

ZWIĄZKI TYPU IS-A ORAZ HAS-A

Podstawowym elementem budulcowym programu jest klasa, jednak program nie jest luźną kolekcją klas – przeciwnie, klasy i ich instancje są ściśle ze

sobą powiązane. Klasy mogą być ze sobą powiązane poprzez kompozycję – znajdują się wówczas w relacji HAS-A – oraz poprzez dziedziczenie; mamy wtedy do czynienia z relacją IS-A.

IS-A oznacza „jest". Warzywo jest rośliną, marchewka jest warzywem, ale rośliną również. Oprócz tego warzywo jest jadalne, ale roślina już nie koniecznie. Jak to przekłada się na programowanie w języku Java? Rozważmy następujące klasy i interfejsy:

```
class Plant {
}

interface Eatable {
}

  // Vegetable IS-A Plant, Vegetable IS-A Eatable
class Vegetable extends Plant implements Eatable {
}

  // Carrot IS-A Vegetable, Carrot IS-A Eatable, Carrot IS-A Plant
class Carrot extends Vegetable {
}
```

Związki IS-A są oparte na dziedziczeniu i implementacji interfejsów. Klasa `A` jest w relacji IS-A z klasą `B`, jeśli `A` jest klasą pochodną – bezpośrednio lub pośrednio – klasy `B`, ewentualnie klasa `A` IS-A `B` jeśli `B` jest interfejsem i `A` implementuje `B`. Mówiąc jeszcze inaczej: `A` IS-A `B` jeśli wyrażenie `(new A()) instanceof B` ma wartość `true`.

HAS-A oznacza „ma". Firma ma szefa, który jest pracownikiem. Jeśli chcemy czegoś od firmy, np. podwyżki, to uruchamiamy procedury w firmie, ale decyzję podejmuje jednak szef. Firma deleguje tego typu zadania właśnie szefom. W języku Java wygląda to tak:

Odp.20 **Metody statyczne nie są wywoływane dla obiektów, tylko dla klas, zaś jeśli wywołujemy je za pośrednictwem referencji, to jest to równoznaczne z wywołaniem dla klasy która jest typem tej referencji. Polimorfizm natomiast polega na wywoływaniu metody odpowiedniej dla typu obiektu. W przypadku wywołania metody statycznej w ogóle nie ma mowy o obiektach. Są tylko klasy.**

```
class Company {
   private Boss boss; // Company HAS-A Boss

   int requestPayRise(Employee emp, int amount) {
     return boss.requestPayRise(emp, amount);
   }
}

class Boss extends Employee { // Boss IS-A Employee
   int requestPayRise(Employee emp, int amount) {
     if(emp instanceof Boss) {
       return amount;
     } else {
       return 0;
     }
   }
}

class Employee {
   private Company company; // Employee HAS-A Company

   private int salary;

   void requestPayRise(int amount) {
     salary += company.requestPayRise(this, amount);
   }
}
```

Związek HAS-A oparty jest na kompozycji. Klasa `A` jest w relacji HAS-A z klasą `B`, tj. `A` HAS-A `B`, jeśli klasa `A` posiada referencję do obiektów klasy `B`. Z relacjami typu HAS-A związany jest kolejny ważny aspekt – delegacja odpowiedzialności. W powyższym przykładzie klasa `Company` posiada metodę `requestPayRise()`, jednak jest ona w całości oddelegowana do powiązanego obiektu klasy `Boss`. To, jaki jest rezultat działania metody `requestPayRise()` w klasie `Company` zależy więc w pełni od implementacji w klasie `Boss`.

PĘTLE, ITERATORY I INSTRUKCJE WARUNKOWE

Niniejszy rozdział pokrywa prezentowanym zakresem wiedzy następujące wymagania egzaminu 1Z0-851:

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

2.2 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). Wytłumacz, w jaki sposób zmienia się wartość licznika pętli w trakcie wykonania i jaka jest wartość tego licznika po wykonaniu pętli.

INSTRUKCJA WARUNKOWA IF

Instrukcja warunkowa `if` służy do oznaczenia pewnego fragmentu kodu, który ma się wykonać wtedy i tylko wtedy, gdy spełniony będzie określony warunek. Ów warunek to dowolne wyrażenie typu logicznego. Pamiętajmy o tym, że w języku Java literały `1` i `0` nie odpowiadają wartościom logicznym `true` i `false`, tak jak to ma miejsce w niektórych językach programowania. Nieprawidłowy jest więc kod:

```
if(1) { // warunek musi być wyrażeniem typu boolean
  // blok kodu
}
```

Jeślibyśmy chcieli – z jakichś dziwnych pobudek – zapisać instrukcję warunkową której warunek jest zawsze prawdziwy to zamiast powyższego powinniśmy napisać:

```
if(true) {
  // blok kodu
}
```

Instrukcja `if` nie należy do szczególnie skomplikowanych ale nawet ona może zostać na egzaminie certyfikacyjnym użyta do wprowadzenia nieuważnych w błąd. Popularne są dwa triki. Pierwszy polega na użyciu w warunku operatora przypisania zamiast operatora równości, np. w taki sposób:

```
public static void main(String[] args) {
  boolean b = true;

  if(b = false) { // operator przypisania zamiast operatora równości
    b = false;
  }

  System.out.println(b);
}
```

Program ten skompiluje się a uruchomiony wykona się poprawnie, tyle, że rezultat może być nieco inny niż spodziewany – zmienna `b` będzie miała wartość `false`.

Drugi trik polega na złośliwym, wprowadzającym w błąd formatowaniu zagnieżdżonych instrukcji `if-else` – zerknijmy na poniższy przykład:

```
public static void main(String[] args) {
  if (1 == 2)
    if (2 == 3)
      System.out.println("1 == 2 & 2 == 3");
  else
    System.out.println("1 != 2");
}
```

Na pierwszy rzut oka wygląda na to, że uruchomienie programu spowoduje wyświetlenie tekstu „1 != 2”, ale jest inaczej. Niezależnie od tego, jakie wcięcia zrobimy w naszym kodzie, `else` dotyczy drugiej, zagnieżdżonej instrukcji `if`. Zapamiętajmy prostą zasadę – `else` zawsze należy do najbardziej zagnieżdżonej instrukcji `if`, do jakiej może należeć. Prawidłowo sformatowany kod wygląda więc tak:

```
public static void main(String[] args) {
  if (1 == 2)
    if (2 == 3)
      System.out.println("1 == 2 & 2 == 3");
    else
      System.out.println("1 != 2");
}
```

INSTRUKCJA WYBORU SWITCH

Instrukcji wyboru `switch` używamy, jeśli chcemy w zależności od wartości pewnego wyrażenia wykonać jeden z kilku fragmentów kodu. Poniższy program używa instrukcji `switch` do określania parzystości liczby przekazanej jako argument wywołania:

```
public static void main(String[] args) {
  int x = Integer.parseInt(args[0]);

  switch (x % 2) { // x % 2 to tzw. wyrażenie wyboru
    case 0: // wariant dla wartości 0
      System.out.println(x + " jest liczbą parzystą");
      break;
```

```
    case 1: // wariant dla wartości 1
      System.out.println(x + " jest liczbą nieparzystą");
      break;

    default: // wariant domyślny
      System.out.println("nieprzewidziana sytuacja");
      break;
  }
}
```

Operator `%` (tj. operator reszty z dzielenia) zaimplementowany w języku Java może zwracać ujemne wartości, stąd wyrażenie `x % 2` oprócz wartości 1 i 0 może zwrócić także wartość -1. Stanie się tak dla każdej ujemnej liczby nieparzystej. Powyższy program nie rozpozna prawidłowo takiej sytuacji i wyświetli komunikat „nieprzewidziana sytuacja". Poprawmy go więc w następujący sposób:

```
public static void main(String[] args) {
  int x = Integer.parseInt(args[0]);
  switch (x % 2) {
    case 0:
      System.out.println(x + " jest liczbą parzystą");
      break;

    case -1: // dodano wariant dla wartości -1
    case 1:
      System.out.println(x + " jest liczbą nieparzystą");
      break;

    default:
      System.out.println("nieprzewidziana sytuacja");
      break;
  }
}
```

Instrukcja `switch` działa w ten sposób, że w pierwszej kolejności wyliczana jest wartości wyrażenia wyboru a następnie wykonywany jest skok do wariantu oznaczonego tą wartością. Co ważne, wariant jest tylko punktem początkowym wykonania i kod będzie się wykonywał przechodząc kaskadowo przez kolejne bloki `case` aż do napotkania instrukcji `break`. Wykorzystując tę własność mogliśmy w powyższym programie opatrzyć ten sam blok kodu dwiema etykietami `case`, z wartością 1 i -1.

Wariant oznaczony etykietą `default` to wariant domyślny, wykonywany gdy dla wyliczonej wartości nie zdefiniowano wariantu typu `case`. Wariant domyślny – tj.

etykieta `dafault` – jest opcjonalny i może w obrębie danej instrukcji `switch` występować co najwyżej raz. Jeśli nie zdefiniowano wariantu domyślnego, a wartość wyrażenia wyboru nie odpowiada żadnemu ze zdefiniowanych wariantów to instrukcja `switch` zakończy się nie wykonując żadnego kodu, tj. sterowanie przejdzie do kodu umieszczonego za instrukcją `switch`.

Dla zwiększenia czytelności kodu zaleca się aby wariant domyślny umieszczać jako ostatni, jednak nie ma takiego wymogu technicznego – etykieta `default` może znajdować się na dowolnej pozycji, także jako pierwsza, przed etykietami `case` lub gdzieś pomiędzy nimi.

Możemy zdefiniować co najwyżej jeden wariant domyślny i podobnie, nie jest dopuszczalne zdefiniowanie więcej niż jednego wariantu `case` dla tej samej wartości. Przykładowo, nie skompiluje się kod zawierający instrukcje:

```
switch (x) {
  case 1: // błąd - dwa warianty dla tej samej wartości
    System.out.println("jeden");
    break;

  case 1: // błąd - dwa warianty dla tej samej wartości
    System.out.println("one");
    break;
}
```

Instrukcja `switch` może operować na wartościach typu `char`, `byte`, `short` i `int` a także na wartościach ich typów opakowujących – tj. `Character`, `Byte`, `Short` i `Integer` – oraz na typach wyliczeniowych.

Argumenty dla poszczególnych wariantów (tj. dla etykiet `case`) muszą mieć typ dający się automatycznie zrzutować do typu wyrażenia wyboru. W szczególności, jeśli wyrażenie wyboru jest typu `byte`, to nie możemy zdefiniować wariantu dla wartości 128, gdyż wartość ta nie mieści się w zakresie dopuszczalnym dla tego typu. Przykładowo, kod:

```
public static void main(String[] args) {
  byte x = 4;

  switch (x) {
    case 127:
```

```
      System.out.println("sto dwadzieścia siedem");
      break;

    case 128: // błąd! wartość 128 nie może być przypisana do typu byte
      System.out.println("sto dwadzieścia osiem");
      break;
  }
}
```

się nie skompiluje, jako że zmienna `x` jest typu `byte`, który przyjmuje wartości z zakresu od -128 do 127. Dodatkowo, wartością wariantu w żadnym wypadku nie może być `null`.

Pyt.21 **Czy możliwe jest użycie w instrukcji *switch* wyrażenia wyboru typu *String*?**

Argumenty dla wariantów muszą być wartościami znanymi w czasie kompilacji, a więc poza literałami, mogą to być tylko zmienne finalne, z wartością przypisaną w czasie deklaracji. Przykładowo, kod:

```
public static void main(String[] args) {
  final byte a = 64;
  final byte b;

  b = 32;

  switch (127) {
    case a:
    case b: // błąd!
      System.out.println("potęgi dwójki");
  }
}
```

się nie skompiluje, jako że zmienna `b` nie jest inicjalizowana w czasie deklaracji a została użyta jako argument wariantu.

PĘTLE WHILE I DO-WHILE

Pętla `while` służy do cyklicznego wykonywania pewnego fragmentu kodu tak długo, jak długo określony warunek jest prawdziwy, tj. dopóty dopóki dozór pętli ma wartość `true`. Przykładowo, uruchomienie programu:

```
public static void main(String[] args) {
  int x = 1;

  while(x < 4) {
    System.out.println(x);

    x++;
  }
}
```

spowoduje wyświetlenie ciągu liczb:

```
1
2
3
```

Podobnie jak w przypadku instrukcji warunkowej `if` dozór pętli musi być wyrażeniem typu logicznego, tj. wyliczać się do wartości `true` lub `false`.

Odp.21 **Nie! Dopuszczalne są tylko i wyłącznie typy *char*, *byte*, *short* i *int*, ich typy opakowujące oraz typy wyliczeniowe.**

Pętla `do-while` to mutacja pętli `while`. Odwrócona jest tutaj kolejność sekwencji w której wpierw sprawdzany jest warunek a dopiero potem, jeśli jest on prawdziwy wykonywany jest blok kodu umieszczony w pętli. W przypadku pętli `do-while` warunek sprawdzany jest po pierwszym wykonaniu kodu. Przykładowo, uruchomienie programu:

```
public static void main(String[] args) {
  int x = 4;

  do {
    System.out.println(x);

    x++;
  } while (x < 4); // ten warunek nigdy nie jest spełniony
}
```

spowoduje wyświetlenie tekstu:

```
4
```

Zwróćmy uwagę na średnik umieszczony na końcu instrukcji pętli. Jest on niezbędny do tego aby kod się skompilował.

PĘTLA FOR

Pętla `for` składa się z trzech oddzielonych średnikami sekcji w części sterującej pętli, oraz kodu umieszczonego w tejże pętli. Przyjęło się mówić, że pierwsza sekcja służy do deklaracji zmiennych, druga to warunek pętli – a więc wyrażenie typu logicznego – a trzecia to instrukcja inkrementacji zmiennych. Zazwyczaj tak właśnie się tych sekcji używa, ale w ogólności sekcja pierwsza może zamiast deklaracji zawierać dowolną instrukcję. W sekcji trzeciej także możemy umieścić dowolną instrukcję. Sekcja druga to dowolne wyrażenie typu `boolean`. Dodatkowo, każda z sekcji może być zupełnie pusta. Przykładowo, poprawną jest pętla:

```
for (System.out.print("sekcja 1."); ; System.out.print("sekcja 3.")) {
  break;
}
```

Jeśli druga sekcja jest pusta – tak jak w powyższym przykładzie – to pętla zachowuje się tak, jakby był tam umieszczony literał `true`. Jest to więc pętla nieskończona, a właściwie to byłaby to pętla nieskończona, gdyby nie instrukcja `break` umieszczona w ciele pętli.

Sekcja pierwsza to kod, który uruchamiany jest jako pierwszy, zawsze dokładnie raz, niezależnie od wartości sekcji drugiej. Przykładowo, uruchomienie programu:

```
public static void main(String[] args) {
  int x = 1, y = x;

  for (System.out.print("sekcja 1."); x != y;
         System.out.print("sekcja 3.")) {
    System.out.print("ciało pętli");
  }
}
```

spowoduje wyświetlenie tekstu:

```
sekcja 1.
```

Po wykonaniu sekcji pierwszej, przeznaczonej na inicjalizację zmiennych sterujących pętli, obliczana jest wartość wyrażenia z sekcji drugiej. Jeśli wyrażenie

to ma wartość `true` to wykonuje się ciało pętli. Na końcu wykonywany jest kod z sekcji trzeciej nagłówka pętli, przeznaczonej na kod inkrementujący wartości zmiennych sterujących. Co ważne, jeśli w sekcji pierwszej pętli zadeklarowano zmienną, to widoczność tej zmiennej ograniczona jest do tej pętli. Przykładowo, kod:

```
public static void main(String[] args) {
  for (int x = 1; x < 2; x++) {
    System.out.println(x);
  }

  System.out.println(x); // błąd! zmienna x istnieje tylko w obrębie pętli
}
```

nie skompiluje się, jako że zmienna `x` widoczna jest tylko i wyłącznie w obrębie pętli w której została zadeklarowana.

INSTRUKCJE BREAK I CONTINUE

Instrukcja `break` służy do natychmiastowego przerwania wykonania pętli – sterowanie zostaje wówczas przeniesione do pierwszej instrukcji za pętlą. Efektem uruchomienia programu:

```
public static void main(String[] args) {
  int i = 0;

  for (System.out.println("sekcja 1."); condition();
       System.out.println("sekcja 3.")) {

    System.out.println("ciało pętli");

    if(i++ > 0)
      break;
  }

  System.out.println("po pętli");
}

public static boolean condition() {
  System.out.println("sekcja 2. (true)");

  return true;
}
```

jest wyświetlenie sekwencji:

```
sekcja 1.
sekcja 2. (true)
ciało pętli
sekcja 3.
sekcja 2. (true)
ciało pętli
po pętli
```

Instrukcja `continue` przerywa bieżące wykonanie pętli, a więc tylko wykonanie ciała pętli w bieżącej iteracji. Zwróćmy uwagę, że instrukcja `continue` nie powoduje zaniechania wykonania trzeciej sekcji nagłówka pętli. Efektem uruchomienia programu:

```
public static void main(String[] args) {
  for (int i = 0; i < 10; i++) {
    if (i % 2 == 1)
      continue;

    System.out.println(i + " jest liczbą parzystą");
  }
}
```

jest wyświetlenie sekwencji:

```
0 jest liczbą parzystą
2 jest liczbą parzystą
4 jest liczbą parzystą
6 jest liczbą parzystą
8 jest liczbą parzystą
```

W przypadku umieszczenia instrukcji `break` lub `continue` w pętli zagnieżdżonej w innej pętli mają one skutek tylko dla pętli w której bezpośrednio się znajdują, ale możliwe jest wskazanie – poprzez etykietę – pętli bardziej zewnętrznej. Przykładowo, uruchomienie programu:

```
public static void main(String[] args) {
  outerLoop:
  for (int j = 0; ; j += 100) {
    for (int i = 0; i < 5; i++) {
      if ((i + j) % 2 == 1)
        continue;
```

```
      if (j > 100)
        break outerLoop; // przerywa wykonanie także pętli zewnętrznej

      System.out.println(i + j + " jest liczbą parzystą");
    }
  }
}
```

skutkuje wyświetleniem sekwencji:

```
0 jest liczbą parzystą
2 jest liczbą parzystą
4 jest liczbą parzystą
100 jest liczbą parzystą
102 jest liczbą parzystą
104 jest liczbą parzystą
```

Instrukcja `continue` z powyższego przykładu nie została oznaczona żadną etykietą, dlatego dotyczy ona wewnętrznej pętli `for`. Instrukcja `break` z etykietą `outerLoop` powoduje natomiast przerwanie wykonania obydwu pętli, jako że etykietą tą została oznaczona pętla zewnętrzna.

PĘTLA FOR-EACH

Pętla `for-each` służy do iteracji po kolejnych elementach tablicy lub kolekcji. Przykładowo, aby wykonać jakąś akcję dla kolejnych elementów tablicy – w tym przypadku dla tablicy liczb typu `int` – napiszemy:

```
for (int i : new int[] { 1, 3, 5, 7, 11 }) {
  System.out.print(i + " ");
}
```

Przed dwukropkiem w nagłówku pętli deklarujemy zmienną do której będą przypisywane kolejne wartości tablicy lub kolekcji po której iterujemy. Po dwukropku umieszczamy dowolne wyrażenie, którego wartością jest kolekcja lub tablica. W powyższym przykładzie jest to instrukcja tworząca nową tablicę, ale może to być także wywołanie metody czy zmienna referencyjna odnosząca się do obiektu pewnej kolekcji, np.:

```
public static void main(String[] args) {
  List<Integer> someList = new LinkedList<Integer>();

  someList.add(1);
  someList.add(2);
  someList.add(3);

  for (int i : someList) {
    System.out.print(i + " ");
  }
}
```

Zwróćmy uwagę, że typ zmiennej pętli musi być zgodny (zgodny pod względem przypisania) z typem elementów kolekcji. Nie oznacza to, że musi to być ten sam typ. Działa auto-boxing, co widać w powyższym przykładzie; typ zmiennej może być także nadtypem typu elementów kolekcji.

Analogicznie jak w przypadku tradycyjnej pętli `for` zmienna pętli `for-each` ma zakres widoczności ograniczony do tej pętli, tj. próba użycia jej za tą pętlą zakończy się błędem kompilacji.

WYJĄTKI

Niniejszy rozdział pokrywa prezentowanym zakresem wiedzy następujące wymagania egzaminu 1Z0-851:

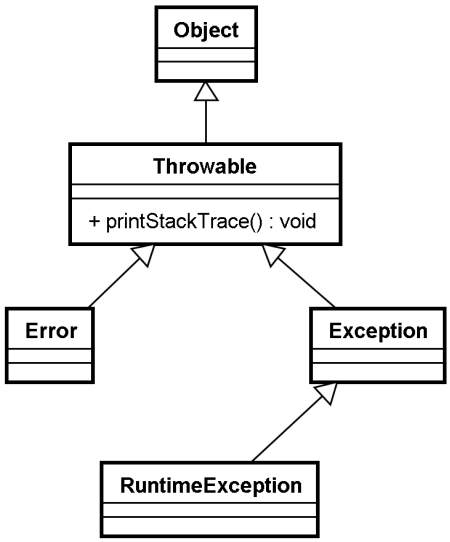
2.3 Napisz kod w którym używasz wyjątków i zaimplementuj kod obsługi wyjątków (`try`, `catch`, `finally`) oraz zadeklaruj metody i metody przysłaniające, które rzucają wyjątki.

2.4 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").

2.5 Określ, jakie sytuacje mogą spowodować rzucenie wyjątków:
`ArrayIndexOutOfBoundsException`,
`ClassCastException`,
`IllegalArgumentException`,
`IllegalStateException`,
`NullPointerException`,
`NumberFormatException`,
`AssertionError`,
`ExceptionInInitializerError`,
`StackOverflowError` lub
`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.

RZUCANIE I OBSŁUGA WYJĄTKÓW

Wyjątek w języku Java to obiekt, który opisuje pewną sytuację błędną lub nieprawidłową – wyjątkową. Jest to obiekt odpowiedniego typu, tj. obiekt klasy `Throwable` lub jej dowolnej podklasy. Poniższy diagram przedstawia trzon hierarchii dziedziczenia dla klas wyjątków:



Każda z klas `Error`, `Exception` i `RuntimeException` ma jeszcze wiele innych podklas, oznaczających konkretny rodzaj sytuacji wyjątkowej. Z punktu widzenia programisty wyjątki dzielą się na dwa rodzaje: kontrolowane (ang. checked) i niekontrolowane (ang. unchecked). Wyjątki kontrolowane tym różnią się od niekontrolowanych, że muszą być jawnie obsłużone w metodach, tj. albo muszą być złapane, albo zadeklarowane na liście wyjątków rzucanych przez metodę. Zerknijmy na poniższy przykład:

```
class SomeClass {
  void someOp() {
    throw new NullPointerException();
  }
```

```
  void someOtherOp() throws IOException {
    throw new IOException();
  }

  void nextOp() {
    try {
      throw new IOException();
    } catch (IOException ioExc) {
      ioExc.printStackTrace();
    }
  }
}
```

W metodzie `someOp()` rzucamy wyjątek `NullPointerException`. Jest to wyjątek niekontrolowany, więc nic specjalnego nie musimy robić. W metodzie `someOtherOp()` rzucamy wyjątek `IOException`. Jest on wyjątkiem kontrolowanym, więc nie możemy go zignorować – deklarujemy, że metoda rzuca ten wyjątek. W metodzie `nextOp()` również pojawia się wyjątek `IOException`, jednak w tym wypadku od razu go łapiemy i obsługujemy.

Wyjątki niekontrolowane to instancje klas `Error` i `RuntimeException` oraz ich dowolnych podklas. Wyjątki kontrolowane to instancje klas `Throwable` i `Exception` oraz ich podklas, oczywiście z wyłączeniem klas `Error` i `RuntimeException`. Wyjątków niekontrolowanych nie musimy deklarować czy obsługiwać, ale możemy, poprawny jest więc poniższy kod:

```
class SomeClass {
                  // ta deklaracja jest nadmiarowa ale dozwolona
  void someOp() throws Error {
    throw new Error();
  }
}
```

Obsługa wyjątków polega na tym, że kod który może rzucić wyjątek ujmujemy w klauzulę `try`. Bezpośrednio po klauzuli `try` następuje seria klauzul `catch` a po klauzulach `catch` klauzula `finally`. Klauzula `finally` jest opcjonalna. Opcjonalne są także klauzule `catch`, ale jeśli nie ma żadnej klauzuli `catch`, to wymagana jest klauzula `finally`. Jeśli nie ma klauzuli `finally` to wymagana jest co najmniej jedna klauzula `catch`. W klauzuli `finally` umieszczamy kod, który ma się wykonać zawsze, po wykonaniu kodu objętego klauzulą `try`. Jest to dobre miejsce na umieszczenie kodu sprzątającego, który powinien się uruchomić

niezależnie od tego, co się stało w klauzuli `try`. To, że kod ten uruchamia się zawsze demonstruje dobrze poniższy przykład:

```
public class Test {
  public static void main(String[] args) {
    someOp();
  }

  public static int someOp() {
    try {
      return 1;
    } finally {
      System.out.println("klauzula finally");
    }
  }
}
```

Uruchomienie programu spowoduje wyświetlenie tekstu „klauzula finally" – a mogłoby się wydawać, że po wykonaniu instrukcji `return 1` wykonanie programu powraca natychmiast do metody `main(…)`. Tak nie jest.

Klauzul `catch` może być dowolna ilość. Co prawda moglibyśmy zadeklarować jedną klauzulę `catch` która łapie wszystkie wyjątki, ale przeważnie chcemy wykonać różne akcje w zależności od typu wyjątku, przynajmniej dla niektórych z nich. Klauzule obsługujące konkretne wyjątki umieszczamy jako pierwsze a w dalszej kolejności występują klauzule bardziej ogólne. Demonstruje to poniższy przykład:

```
class SubException extends Exception {}

class NextException extends Exception {}

public class Test {
  public void someOp() {
    try {
      throwingOp();
    } catch (SubException subExc) {
      // jakieś specyficzne operacje
    } catch (Exception exc) {
      // operacje ogólne, np. logowanie błędu
    }
  }

  public void throwingOp() throws SubException, NextException {}
}
```

Wyjątek `SubException` obsługujemy w pewien szczególny sposób, więc umieściliśmy dedykowaną klauzulę `catch`. Następna klauzula obsługuje wszystkie inne wyjątki, w tym wyjątki klasy `NextException`. Umieszczenie tych klauzul w odwrotnej kolejności byłoby błędem kompilacji – klauzule bardziej ogólne muszą znajdować się za tymi wyspecjalizowanymi.

Zobaczmy teraz jak fakt rzucenia wyjątku przez pewien fragment kodu (metodę którą wywołujemy) wpływa na przepływ sterowania w programie. Zerknijmy na poniższy przykład:

```
public void someOp() {
  try {
    System.out.println("przed wywołaniem throwingOp()");

    throwingOp();

    System.out.println("po wywołaniu throwingOp()");
  } catch (Exception exc) {
    System.out.println("w bloku catch");
  } finally {
    System.out.println("w bloku finally");
  }

  System.out.println("za blokiem try-catch");
}

public void throwingOp() throws Exception { }
```

Metoda `throwingOp()` deklaruje, że może rzucić wyjątek (może, ale nie musi), jednak aktualnie jej ciało jest puste, tak więc wyjątek nie będzie podniesiony; instrukcje będą wykonane w swej naturalnej kolejności. Uruchomienie metody `someOp()` spowoduje wyświetlenie tekstu:

```
przed wywołaniem throwingOp()
po wywołaniu throwingOp()
w bloku finally
za blokiem try-catch
```

Metoda `throwingOp()` nie rzuciła wyjątku w związku z czym sterowanie doszło do linii wyświetlającej tekst „po wywołaniu throwingOp()". Nie był rzucony także żaden inny wyjątek w obrębie bloku `try` tak więc klauzula `catch` nie została wykonana. Wykonały się za to – jak zawsze – instrukcje z klauzuli

`finally`. Za blokiem try-catch-finally była jeszcze jedna instrukcja wyświetlająca tekst i ona także została wykonana. Zobaczmy teraz jak na przebieg programu wpłynie wyjątek rzucony z metody `throwingOp()`. Zmodyfikujmy w tym celu tę metodę tak jak pokazano poniżej:

```
public void throwingOp() throws Exception {
  throw new Exception();
}
```

Uruchomienie metody `someOp()` tak jak poprzednio, tyle że ze zmodyfikowaną implementacją metody `throwingOp()` spowoduje wyświetlenie tekstu:

```
przed wywołaniem throwingOp()
w bloku catch
w bloku finally
za blokiem try-catch
```

Jak widać wystąpienie wyjątku powoduje przerwanie wykonania kodu i przeskoczenie do klauzuli obsługi błędu `catch`. Instrukcja umieszczona za wywołaniem metody `throwingOp()` nie zostanie więc nigdy wykonana. Po wykonaniu kodu obsługi wyjątku wykonana będzie klauzula `finally` i kod umieszczony za blokiem try-catch-finally. Błąd został obsłużony i nie rzutuje na dalsze wykonanie programu. Podsumujmy więc – jeśli pewna instrukcja rzuca wyjątek, to wykonanie programu jest natychmiast wstrzymywane i zaczyna się obsługa wyjątku, tak więc instrukcje znajdujące się za instrukcją która rzuciła wyjątek – a przed kodem który wyjątek obsłuży – nigdy się nie wykonają. Po wykonaniu kodu obsługi błędu program wznawia swe normalne wykonanie.

Pyt.22 **Jeśli klasa *A* dziedziczy z klasy *Exception* a klasa *AA* z klasy *A*, to czy poprawnym będzie zadeklarowanie w pewnej metodzie, że rzuca ona wyjątek *A* i rzucanie w niej wyjątku *AA*?**

POPULARNE TYPY WYJĄTKÓW

Bodaj najpopularniejszym wyjątkiem jest `NullPointerException`. Dziedziczy on bezpośrednio z klasy `RuntimeException` a więc jest wyjątkiem

niekontrolowanym (ang. unchecked). Wyjątek ten rzucany jest przez Wirtualną Maszynę Javy (ang. akr. JVM) przy próbie odwołania się do obiektu (np. uruchomienie metody) z użyciem referencji, która ma wartość `null`.

Pamięć operacyjna wykonującego się programu składa się z dwóch obszarów: stosu (ang. stack) i sterty (ang. heap). Na stercie znajdują się obiekty, a na stosie kontekst wykonywanego kodu, tj. zmienne lokalne. Gdy wywołujemy metodę, to jej zmienne lokalne, a więc i przekazywane parametry zapisywane są na stosie. Pamięć ta zwalniana jest dopiero po zakończeniu wykonania metody. A co jeśli metody nigdy się nie kończą, za to wywołują (rekurencyjnie) kolejne metody? Wtedy właśnie, po pewnym czasie wyczerpuje się pamięć stosu i Wirtualna Maszyna Javy rzuca wyjątek `StackOverflowError`. Wyjątek ten jest podklasą (nie bezpośrednią) klasy `Error` a więc jest to również wyjątek niekontrolowany. Zresztą, wszystkie opisywane tu wyjątki są wyjątkami niekontrolowanymi, a więc podklasami klasy `Error` albo `RuntimeException`.

Odp.22 **Tak! Klasy *A* i *AA* dziedziczą z typu *Exception* zatem są to wyjątki kontrolowane, jednak *AA* jest podtypem *A*, więc rzucanie go przy deklaracji rzucania *A* jest poprawne.**

Wyjątkiem błędu formatu mogą zakończyć się metody konwersji ze stringa na typy numeryczne, np. metoda `parseInt(…)` z klasy `Integer`. Jeśli tekst nie może być przekonwertowany na liczbę, a dzieje się tak gdy np. próbujemy konwertować tekst „sto jeden” a nie „101”, to metody te rzucają wyjątek `NumberFormatException`. Wyjątek ten jest podklasą klasy `IllegalArgumentException`. Widzimy na tym przykładzie jak hierarchia dziedziczenia klas wyjątków odzwierciedla hierarchię rodzajów błędów. Wyjątek `IllegalArgumentException` oznacza, że argument metody jest z pewnych powodów nieprawidłowy, a jej podklasa `NumberFormatException` precyzuje, że argument jest nieprawidłowy, bo nie może być przekonwertowany do wartości liczbowej.

Wyjątek `ArrayIndexOutOfBoundsException` rzucany jest gdy nastąpi odwołanie do indeksu tablicy, który jest z poza dopuszczalnego zakresu – tj. jeśli indeks jest liczbą ujemną lub większą bądź równą (numerujemy od zera) od ilości elementów w tablicy.

Wyjątek `ClassCastException` jest efektem próby rzutowania obiektu na typ z nim nie kompatybilny.

Wyjątek `IllegalStateException` oznacza, że stan urządzeń bądź zewnętrznych systemów których nasz program używa jest nieprawidłowy i wykonywana operacja nie może być z tego powodu zakończona.

Wyjątek `AssertionError` informuje, że nie jest prawdziwa jedna z asercji. Asercje są szczegółowo omówione w odrębnym podrozdziale.

Wyjątek `ExceptionInInitializerError` jest rzucany gdy wystąpi błąd w trakcie statycznej inicjalizacji zmiennej, bądź w trakcie wykonania bloku inicjalizacyjnego.

Wyjątek `NoClassDefFoundError` jest rzucany gdy Wirtualna Maszyna Javy – a mówiąc precyzyjniej mechanizm ładowania klas – nie może znaleźć wszystkich klas niezbędnych do tego by załadować definicję klasy do której odwołujemy się w kodzie. Przeważnie błąd ten spowodowany jest brakiem odpowiedniej biblioteki na ścieżce przeszukiwania (ang. classpath).

API

Niniejszy rozdział pokrywa prezentowanym zakresem wiedzy następujące wymagania egzaminu 1Z0-851:

3.1 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 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 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 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), `*` (gwiazdka), `+` (plus), `?`, `\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`).

KLASY OPAKOWUJĄCE TYPÓW PROSTYCH

Każdy typ prosty (ang. primitive type) z języka Java posiada swój odpowiednik w postaci klasy w pakiecie `java.lang`. Dla typu `int` jest to `java.lang.Integer`, dla typu `char` `java.lang.Character` a dla pozostałych klasy nazywają się tak samo jak typ prosty, tyle, że nazwa klasy zaczyna się wielką literą. Dla przykładu, dla typu `float` będzie to `java.lang.Float`. Do czego służą te klasy i dlaczego ich potrzebujemy? Z dwu powodów. Po pierwsze, aby móc używać wartości prostych w operacjach, które wymagają obiektów. Zerknijmy na poniższy przykład:

```
public class Test {
  public static void main(String[] args) {
    Storage storage = new Storage();

    int x = 1;

    storage.addToStorage(new Integer(x));
  }
}

class Storage {
  private Set<Object> storage;

  public void addToStorage(Object obj) {
    this.storage.add(obj);
  }
}
```

Deklarując kolekcję – zbiór – w klasie `Storage` mieliśmy intencję taką, aby móc w tej kolekcji przechowywać dowolne byty. Tyle, że obiekty języka Java to nie są wszystkie byty jakie nas interesują – są jeszcze przecież wartości proste, które obiektami nie są. Aby więc móc dodać wartość zmiennej `x` typu `int` do kolekcji musimy utworzyć obiekt, który tę wartość będzie reprezentował. W powyższym przykładzie jest to konstrukcja `new Integer(x)`. Co prawda odkąd mamy w Javie (a mamy począwszy od Javy 5) "in-boxing" i "out-boxing" nie musimy robić tego explicite, ale nie zmienia to faktu, że taka operacja musi być wykonywana.

Omawiane klasy – oprócz tego, że są obiektowymi opakowaniami dla typów prostych – udostępniają szereg przydatnych operacji konwersji. Jako operację

konwersji możemy również postrzegać samą operację konstrukcji. Żadna z omawianych klas nie udostępnia konstruktora bezargumentowego. Wszystkie za to udostępniają konstruktor jednoargumentowy z argumentem typu prostego odpowiadającego klasie, czyli np. `Boolean(boolean b)` czy `Character(char c)`. Dodatkowo, wszystkie klasy oprócz klasy `Character` udostępniają konstruktor akceptujący obiekt typu `String` – jest to funkcjonalność konwersji z typu tekstowego. Wyjątkowo, klasa `Float` implementuje jeszcze konstruktor, który akceptuje wartości typu `double`. Bardzo ważną rzeczą jest, że obiekty klas opakowujących reprezentują jedną konkretną wartość danego typu. Oznacza to, że wartość reprezentowana np. przez obiekt klasy `Integer` nigdy nie może być zmieniona – nie istnieją operacje „setValue(int val)" czy „increase()". Z tego też powodu nie ma konstruktorów bezargumentowych – liczba czy wartość logiczna, albo litera nieposiadające wartości nie mają sensu, a późniejsze ustawienie tej wartości nie jest możliwe. Ilustruje to dobitnie poniższy przykład:

```
public static void main(String[] args) {
  Integer i = new Integer(1);
  Integer j = i;

  if(i == j) // warunek jest spełniony
    System.out.println("zmienne i oraz j reprezentują ten sam obiekt");

  i = i - 1; // równoważne z instrukcją 'i = new Integer(i.intValue() - 1)'

  if(i == j) // warunek już nie jest spełniony
    System.out.println(
        "zmienne i oraz j nadal reprezentują ten sam obiekt");
}
```

Rezultatem uruchomienia tego programu będzie wypisanie tylko pierwszego ze zdań, albowiem po operacji `i = i - 1` referencja `i` nie odnosi się już do tego samego obiektu. Wartości reprezentowanej przez obiekt, do którego odnosi się zmienna `i` nie da się zmienić, zatem trzeba utworzyć nowy obiekt. Instrukcja `i = i - 1` jest de facto równoważna instrukcji `i = new Integer(i.intValue() - 1)`. Na zakończenie tematu konstruktorów jeszcze słowo komentarza o konwersji z typu tekstowego, a więc o konstruktorach akceptujących obiekt typu `String`. Wszystkie one – dla typów numerycznych, a więc z wyłączeniem klasy `Boolean` – zgłaszają wyjątek `NumberFormatException` w przypadku, gdy konwersja nie jest możliwa.

Taki wyjątek otrzymamy np. jeśli spróbujemy uruchomić konstruktor `new Integer("jeden")` – niestety, Java nie umie jeszcze czytać po polsku. W innych językach też nie umie, ogranicza się do czytania cyfr. Analogiczna konwersja dla typu `Boolean` zawsze kończy się sukcesem, tj. konstruktor tej klasy nie zwraca wyjątków. Obiekt klasy `Boolean` reprezentujący wartość `true` uzyskamy z tekstu "true", przy czym nie ważna jest wielkość liter, zaś obiekt reprezentujący wartość `false` z każdego innego tekstu, oraz gdy zamiast obiektu klasy `String` przekażemy `null` (!). Uwaga! Instrukcja `new Boolean("1")` także powoduje utworzenie obiektu reprezentującego wartość `false`.

Bliskim kuzynem konstruktorów są metody `valueOf(…)`. Są to statyczne metody-fabryki klas opakowujących, których zadaniem jest tworzenie instancji tychże klas. Jaka jest więc różnica między tymi metodami a konstruktorami? Różnica głównie tkwi w tym, że metody `valueOf(…)` nie zawsze tworzą nowy obiekt. Zamiast tego – w celu optymalizacji – używane są istniejące, ale już niewykorzystywane instancje. Szczegóły nie są dla nas ważne, ale trzeba zapamiętać, że jeśli nie zależy nam na utworzeniu nowego obiektu to powinniśmy używać tych metod zamiast konstruktorów. Pomijając względy wydajnościowe – metody `valueOf(…)` dają te same możliwości tworzenia obiektów, co konstruktory (poza tym, że nie ma metody `Float.valueOf(double d)`, ale to szczegół) a oprócz tego, pozwalają na parsowanie liczb zapisanych w systemie innym niż dziesiętny. Zerknijmy na poniższy przykład:

Pyt.23 **Czy do zmiennej typu *boolean* możemy przypisać wartość *null*?**

```
public static void main(String[] args) {
  System.out.println(Integer.valueOf("100", 2));
}
```

Rezultatem uruchomienia tej metody będzie wypisanie liczby 4, albowiem w systemie dziesiętnym taka jest właśnie wartość liczby 100 zapisanej w systemie dwójkowym. Metody `valueOf(String value, int radix)`, służące do konwersji z jednoczesną zmianą systemu liczbowego są zaimplementowane tylko dla typów całkowitoliczbowych, a więc dla typów: `Byte`, `Short`, `Integer` i `Long`.

Bardzo podobne do metod `valueOf(…)` są metody `parseXXX(…)` gdzie *XXX* to nazwa odpowiedniego typu prostego. Są to również metody statyczne do konwersji ze stringów, tyle, że do typów prostych a nie instancji klas opakowujących. Metody `parseXXX(…)` występują w dwu odmianach: jednoargumentowej i dwuargumentowej. Wariant jednoargumentowy – z argumentem typu `String` – zaimplementowany jest we wszystkich klasach poza klasą `Character`; wariant dwuargumentowy tylko w klasach reprezentujących typy całkowitoliczbowe. Pierwszy argument to liczba a drugi (typu `int`) to podstawa systemu liczbowego (ang. radix), w którym ta liczba jest zapisana.

Sprawę tworzenia obiektów z wartości prostych oraz konwersji stringów na obiekty i wartości prymitywne mamy już z głowy. Została jeszcze kwestia przejścia w drugą stronę, a więc pobieranie wartości prostej z obiektu oraz konwersja wartości reprezentowanej przez obiekt na stringa. Aby przejść od typu prostego do stringa można użyć statycznej metody `valueOf(…)` z klasy `String`.

Odp.23 **Nie! Typ *boolean* jest typem prostym. Wartość *null* możemy przypisywać tylko do referencji.**

Aby pobrać wartość prostą reprezentowaną przez daną instancję klasy typu opakowującego należy użyć jednej z metod postaci *xxx*`Value()`. Naturalnie, w odróżnieniu od wcześniej omawianych metod są to metody instancyjne. W klasie `Boolean` zdefiniowano tylko metodę `booleanValue()`, podobnie w klasie `Character` jest tylko metoda `charValue()` ale w pozostałych klasach, tj. w klasach reprezentujących typy numeryczne jest już komplet metod numerycznych, tj. `byteValue()`, `shortValue()`, `intValue()`, `longValue()`, `floatValue()` oraz `doubleValue()`. Można zatem obiekt klasy – dajmy na to – `Byte` przekonwertować wywołaniem jednej funkcji do wartości typu `float`.

Została jeszcze konwersja z obiektu liczbowego na `String` i będzie po wszystkim… a nie było lekko. Przede wszystkim, instancje klas opakowujących są obiektami, a więc mamy do dyspozycji metodę instancyjną `toString()`. Dodatkowo, wszystkie typy numeryczne implementują jednoargumentową statyczną metodę `toString(…)` która akceptuje odpowiadającą typem (np. `int` w klasie `Integer`, `float` w klasie `Float` itd.) wartość prymitywną a zwraca

stringa. W klasie `Integer` oraz `Long` jest jeszcze dwuargumentowy wariant statycznej metody `toString(…)`. Drugi, dodatkowy argument typu `int` określa system liczbowy, w którym zapisana ma być liczba. Trochę jakby nadmiarowo, klasy `Integer` oraz `Long` implementują też jednoargumentowe metody `toBinaryString(…)`, `toOctalString(…)` oraz `toHexString(…)`, które robią dokładnie to samo, co wspomniane przed chwilą metody dwuargumentowe `toString(…)`, tyle, że system liczbowy określony jest w nazwie metody a nie w postaci drugiego argumentu.

KLASA STRING

Zacznijmy od rozwiania ewentualnych wątpliwości natury fundamentalnej – typ tekstowy w języku Java nie jest typem prostym. Co prawda przywykliśmy do tego, że obiekty tworzymy z użyciem operatora `new`, a więc inaczej niż zazwyczaj tworzone są stringi, ale nie zmienia to faktu, że każdy string jest obiektem. Zerknijmy na poniższy kod:

```
public class MyClass {
  public static void main(String[] args) {
    String a = new String("Ola ma kota"); // oczywiście można tak

    String b = "A Ula psa"; // ale możemy użyć także skróconej formy

    System.out.print(b.getClass());
  }
}
```

Uruchomienie tego programu spowoduje wyświetlenie tekstu „class java.lang.String”. Czy jednak obydwa pokazane powyżej sposoby utworzenia obiektu tekstu są równoważne? Otóż nie. Konstrukcja `new String(…)` zawsze skutkuje utworzeniem nowego obiektu, natomiast forma skrócona, z bezpośrednim użyciem literału spowoduje utworzenie nowego obiektu tylko wówczas, gdy obiekt reprezentujący dany tekst jeszcze nie istnieje. A gdzie ma istnieć? Otóż Wirtualna Maszyna Javy utrzymuje pulę stringów – w specjalnym, przeznaczonym do tego celu obszarze pamięci – z nadzieją, że przyczyni się to do optymalizacji wykonania i ilości potrzebnej pamięci. Pierwsze użycie literału tekstowego spowoduje utworzenie nowego obiektu oraz umieszczenie go w owej puli, zaś każde następne

użycie odnosić się będzie do tegoż właśnie obiektu z puli (kolejny obiekt nie jest tworzony). Przekonajmy się o tym analizując poniższy program:

```
public class MyClass {
  public static void main(String[] args) {
    String a = "Ola";
    String b = "Ola";

    System.out.println("a == b : " + (a == b));

    String c = new String("Ola");
    String d = new String("Ola");

    System.out.println("c == d : " + (c == d));
  }
}
```

który uruchomiony wyświetli tekst:

```
a == b : true
c == d : false
```

Zatem zgadza się. Referencja `a` i `b` wskazują na ten sam obiekt (wyrażenie `a == b` ma wartość `true`) zaś `c` i `d` już nie (wyrażenie `c == d` ma wartość `false`).

Kolejnym ważnym faktem odnośnie obiektów klasy `String` jest, że są one niemodyfikowalne (ang. immutable). Obiekt klasy `String` reprezentuje pewien stały, dokładnie jeden tekst. Jeśli potrzebujemy obiektu, który reprezentuje inny tekst, to najzwyczajniej w świecie potrzebujemy nowego obiektu klasy `String`. Zerknijmy na poniższy przykład:

```
public class MyClass {
  public static void main(String[] args) {
    String a = "Ola";
    String b = a;

    System.out.println("a == b : " + (a == b));

    a += " ma kota";

    System.out.println("a == b : " + (a == b));
  }
}
```

który uruchomiony wyświetli tekst:

```
a == b : true
a == b : false
```

Prześledźmy teraz, jak to się dzieje. W pierwszej linii deklarujemy referencję `a` i przypisujemy jej nowo utworzony obiekt reprezentujący tekst „Ola". Obiekt ten ląduje w puli obiektów o której mówiliśmy przed chwilą. Następnie deklarujemy referencję `b` i przypisujemy jej ten sam obiekt, który przypisaliśmy do zmiennej `a`. Nic więc dziwnego, że pierwszą linią wyświetloną przez program jest „a == b : true". Następnie, do zmiennej `a` chcemy dodać sufiks „ ma kota". Wiemy, że obiekty klasy `String` są niemodyfikowalne a więc operator += nie może zmienić wartości obiektu wskazywanego przez zmienną `a`. Co zatem się dzieje? Otóż tworzony jest nowy obiekt inicjowany wartością „Ola ma kota" i to ten właśnie obiekt jest przypisywany do zmiennej `a`. Zmienna `a` wskazuje zatem teraz na nowy, zupełnie inny obiekt niż zmienna `b`, stąd porównanie `a == b` ma wartość `false`.

Pyt.24 **Czy można zmienić łańcuch znaków reprezentowany przez typ *String*? Tzn. czy można sprawić, by ten sam obiekt typu *String* reprezentował inny tekst niż ten, który reprezentował uprzednio?**

String to klasa zupełnie podstawowa, stąd niezbędna jest – także z perspektywy egzaminu na OCPJP – znajomość jej najpopularniejszych metod. Poniżej zaprezentowano listę tychże, wraz z opisem i przykładem wywołania:

`public char` **`charAt`**`(int index)` - zwraca znak znajdujący się na zadanej argumentem wywołania pozycji w stringu, przy czym należy pamiętać, że tak jak w tablicach, indeks zaczyna się od liczby 0. Wyrażenie:

```
"abcdef".charAt(1)
```

ma więc wartość 'b'.

`public String` **`concat`**`(String sufix)` - zwraca nowy obiekt typu `String`, który ma wartość taką jak obiekt dla którego wywołano metodę z doklejonym jako sufiks argumentem wywołania, np. wyrażenie:

```
"abcdef".concat("ghij")
```

ma wartość „abcdefghij”.

`public boolean` **`equalsIgnoreCase`**`(String str)` – metoda pozwala stwierdzić, czy porównywane teksty (ten, dla którego wywołano metodę i ten przekazany jako argument) są równe (leksykograficznie), przy czym przy porównywaniu ignoruje się wielkość liter (litery małe i duże są utożsamiane), np. wyrażenie:

```
"abc".equalsIgnoreCase("AbC")
```

ma wartość `true`.

`public int` **`length`**`()` - metoda ta zwraca długość tekstu reprezentowanego przez obiekt, tj. ilość jego znaków, np. wyrażenie:

```
"abc".length()
```

ma wartość 3.

Odp.24 **Nie! Nie można. Obiekty typu *String* są niemodyfikowalne. Zawsze, gdy mamy wrażenie że udało nam się zmodyfikować istniejący obiekt typu *String* de facto obiekt ten pozostał bez zmian, natomiast nowy tekst jest reprezentowany przez inny, nowy obiekt typu *String*.**

`public String` **`replace`**`(char old, char new)` - metoda zwraca nowy obiekt typu `String`, powstały poprzez zastąpienie (w stringu dla którego wywołano metodę) każdego wystąpienia znaku przekazanego jako pierwszy argument, przez znak przekazany jako drugi argument, np. wyrażenie:

```
"abababab".replace('b', 'c')
```

ma wartość „acacac”.

`public String` **`substring`**`(int begin)` - zwraca stringa powstałego po odrzuceniu początkowych znaków ze stringa dla którego wywołano metodę; pierwszym znakiem zwracanego stringa będzie ten, który znajduje się na pozycji przekazanej jako argument wywołania, np. wyrażenie:

```
"abcdef".substring(1)
```

ma wartość „bcdef”.

`public String` **`substring`**`(int begin, int end)` - zachowuje się jak metoda opisana powyżej, z tym, że obcina także końcówkę tekstu; wartość przekazana jako drugi argument wywołania to indeks pierwszego znaku który nie znajdzie się w zwracanym stringu, np. wyrażenie:

```
"abcdef".substring(1, 3)
```

ma wartość „bc”.

`public String` **`toLowerCase`**`()` - zwraca stringa, który powstał po zamienieniu wszystkich dużych liter stringa dla którego wywołano metodę na odpowiadające litery małe, np. wyrażenie:

```
"aBcDe".toLowerCase()
```

ma wartość „abcde”.

`public String` **`toUpperCase`**`()` - metoda analogiczna do metody opisanej powyżej, z tym, że litery małe zamieniane są na duże, np. wyrażenie:

```
"aBcDe".toUpperCase()
```

ma wartość „ABCDE”.

`public String` **`trim`**`()` - zwraca tekst powstały ze stringa użytego do wywołania metody po obcięciu białych znaków znajdujących się na początku i końcu tegoż stringa, np. wyrażenie:

```
"  przed tabulacja, po spacja ".trim()
```

ma wartość „przed tabulacja, po spacja”.

KLASY STRINGBUFFER I STRINGBUILDER

Klasa `String` reprezentuje sekwencję znaków, typ tekstowy. Jak wiemy z poprzedniego podrozdziału, obiekty klasy `String` są niemodyfikowalne (ang. immutable); każda operacja, w wyniku której chcemy otrzymać inny tekst na bazie istniejącego wymaga utworzenia nowego obiektu, do reprezentowania tego nowego tekstu.

Konieczność tworzenia nowego obiektu przy każdorazowej operacji na tekście jest to coś, czego czasem chcielibyśmy uniknąć. Potrzebujemy typu, który reprezentowałby modyfikowalną sekwencję znaków i właśnie tym typem jest klasa `StringBuffer`.

Aby zapewnić bezpieczne przetwarzanie wielowątkowe (ang. thread-safe) operacje klasy `StringBuffer` są synchronizowane. Ułatwia to znakomicie programowanie, ale w sytuacji gdy wiemy, że nasza aplikacja nie będzie wykonywana jednocześnie przez więcej niż jeden wątek, synchronizacja jest zbędna i wprowadza bezsensowne straty wydajnościowe. Z tego powodu do specyfikacji Java SE w wersji 5 dodano klasę `StringBuilder`, która różni się od klasy `StringBuffer` tylko tym, że jej metody synchronizowane już nie są.

Poniżej opisano metody tych klas – te ważne z perspektywy egzaminu na OCPJP – na przykładzie klasy `StringBuilder`, ale jak wiemy klasa `StringBuffer` ma metody analogiczne, tyle że synchronizowane:

`public StringBuilder` **`append`**`(String str)` - Metoda ta zmodyfikuje łańcuch znaków reprezentowany przez obiekt dla którego ją wywołano w ten sposób, że do jego końca doklei tekst przekazany jako argument. Metoda ta ma wiele wariantów różniących się typem parametru; istnieją między innymi wersje dla parametrów typu: `int`, `char`, `boolean` czy `Object`. Metoda ta zwraca obiekt dla którego została wywołana (nie nowy obiekt jak w klasie `String`) aby umożliwić wywołania łańcuchowe (ang. chained invocations). Uruchomienie metody:

```
public void someOp() {
  StringBuilder stringBuilder = new StringBuilder();

  System.out.println(
      stringBuilder.append("mam ").append(5).append(" lat"));
}
```

spowoduje wyświetlenie tekstu „mam 5 lat".

`public StringBuilder` **`delete`**`(int start, int end)` - Metoda ta zmodyfikuje obiekt dla którego ją wywołano w ten sposób, że usunie ciąg znaków określony argumentami wywołania. Pierwszy argument to pozycja pierwszego znaku który ma być usunięty (jak zawsze liczymy od 0-ra) a drugi to pozycja pierwszego znaku który usunięty nie będzie. Metoda ta także zwraca obiekt dla którego została uruchomiona. Uruchomienie metody:

```
public void someOp() {
  StringBuilder stringBuilder = new StringBuilder("mam 55 lat");

  System.out.println(stringBuilder.delete(4, 5));
}
```

spowoduje wyświetlenie tekstu „mam 5 lat".

`public StringBuilder` **`insert`**`(int offset, String str)` - Metoda ta zmodyfikuje obiekt dla którego ją wywołano w ten sposób, że na pozycji określonej pierwszym argumentem wywołania zostanie wklejony string przekazany jako drugi argument. Analogicznie jak metoda `append(…)` metoda `insert(…)` występuje w wielu odmianach, akceptujących jako drugi argument typy takie jak (między innymi): `int`, `float`, `char`, `Object`. Zwrócony obiekt to obiekt dla którego wywołano metodę. Uruchomienie operacji:

```
public void someOp() {
  StringBuilder stringBuilder = new StringBuilder("mam 5 lat");

  System.out.println(stringBuilder.delete(4, 5).insert(4, "niewiele"));
}
```

spowoduje wyświetlenie tekstu „mam niewiele lat".

`public StringBuilder` **`reverse`**`()` – Metoda ta odwraca kolejność

znaków w obiekcie dla którego ją wywołano. Tak jak dla innych metod, zwrócony obiekt to obiekt dla którego wywołano metodę. Uruchomienie metody:

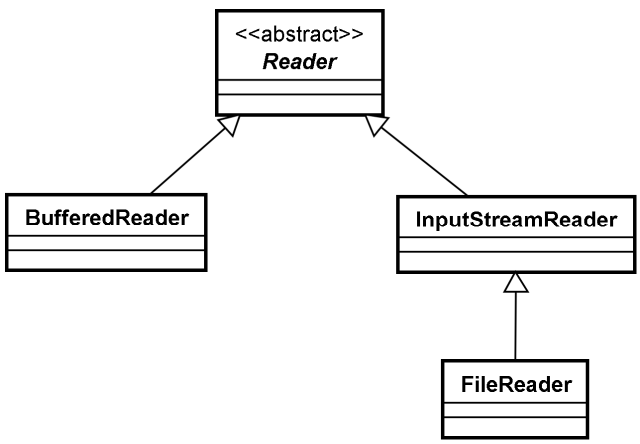
```
public void someOp() {
  StringBuilder stringBuilder = new StringBuilder("abcdef");

  System.out.println(stringBuilder.reverse());
}
```

spowoduje wyświetlenie tekstu „fedcba”.

KLASY PAKIETU JAVA IO

Pakiet `java.io` zawiera definicje kilkudziesięciu klas i interfejsów, ale na egzaminie na OCPJP wymagana jest znajomość tylko kilku z nich. W tym kontekście interesujące są dla nas hierarchie klas `Reader` i `Writer` oraz klasy `File` i `Console`. Klasy `Reader` i `Writer`, a w zasadzie ich podklasy – bo te są klasami abstrakcyjnymi – służą odpowiednio do odczytu i zapisu strumieni znaków. Wymienione klasy nie są ze sobą powiązane w sensie dziedziczenia i wszystkie są bezpośrednimi podklasami klasy `java.lang.Object`.

Strukturę dziedziczenia dla klasy `Reader`, uwzględniającą tylko te podklasy których znajomość wymagana jest na egzaminie, ilustruje poniższy diagram:



Podstawową operacją klasy `Reader` i jej klas pochodnych jest operacja `read(…)` służąca do odczytu ze źródła danych pewnej ilości znaków. Dla klasy `InputStreamReader` źródłem danych jest dowolny strumień bajtów (tj. `InputStream`) a dla klasy `FileReader` strumień danych oparty na pliku dyskowym. Klasa `BufferedReader` służy do minimalizacji liczby odczytów ze źródła danych. Dane pobierane są dużymi porcjami i przechowywane w buforze, dzięki czemu liczba faktycznych odczytów może być wielokrotnie mniejsza niż liczba wywołań metody `read(…)`. Taka optymalizacja ma szczególne znaczenie, jeśli charakterystyka źródła danych powoduje, że operacje odczytu są kosztowne, co ma miejsce gdy źródłem tym jest plik dyskowy. Instancje klasy `BufferedReader` tworzymy na podstawie instancji podklas klasy `Reader` z użyciem jednego z dwu konstruktorów:

```
public BufferedReader(Reader in)

public BufferedReader(Reader in, int bufferSize)
```

Instancje klasy `InputStreamReader` możemy skonstruować na bazie instancji podklas klasy `InputStream`, specyfikując jeszcze ewentualnie kodowanie jakiego należy użyć do konwersji strumienia bajtów na strumień znaków. Do dyspozycji mamy cztery wersje konstruktora (trzy sposoby na określenie kodowania), ale nam wystarczą dwie:

```
public InputStreamReader(InputStream in)

public InputStreamReader(InputStream in, Charset charset)
```

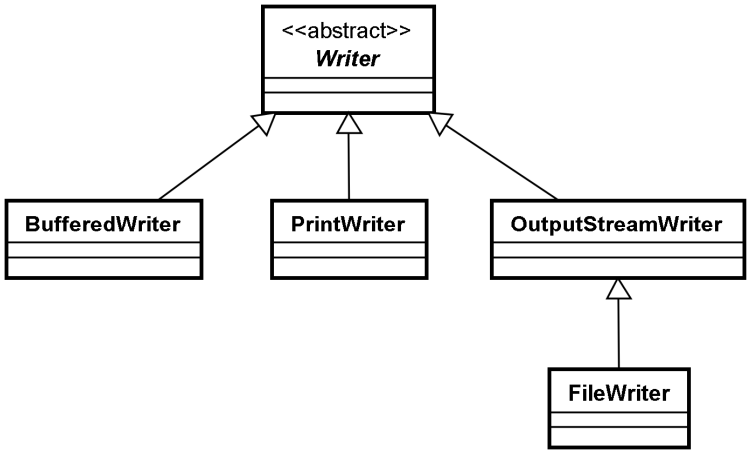
Klasy z grupy `InputStream` służą do odczytu strumienia bajtów ze źródła danych. Z perspektywy egzaminu OCPJP nie są dla nas interesujące same w sobie.

Instancje klasy `FileReader` tworzymy w oparciu o plik określony albo jako obiekt klasy `File` (ewentualnie `FileDescriptor`, ale to jest już poza zakresem naszych zainteresowań) albo poprzez nazwę pliku, a więc obiekt klasy `String`. Sygnatura konstruktorów jest następująca:

```
public FileReader(File file) throws FileNotFoundException

public FileReader(String fileName) throws FileNotFoundException
```

Klasa `File` reprezentuje abstrakcyjny zbiór dyskowy; chciałoby się powiedzieć plik, ale niekoniecznie, może to być bowiem także katalog. O tym, czy dana instancja klasy `File` reprezentuje plik czy katalog możemy się przekonać wywołując jedną z metod `isFile()` albo `isDirectory()`, które zwracają wartość typu `boolean`. Nie bez znaczenia jest także stwierdzenie, że klasa `File` reprezentuje abstrakcyjny (!) zbiór dyskowy. Zbiór ten jest abstrakcyjny w tym sensie, że może to być zbiór jeszcze nie istniejący; instancji obiektu klasy `File` możemy użyć do utworzenia nowego zbioru (metoda `createNewFile()` lub `mkdir()`), albo do sprawdzenia czy dany zbiór istnieje (metoda `exists()`).

Struktura dziedziczenia dla klasy `Writer` jest podobna do struktury dla klasy `Reader`, aczkolwiek obecna tu klasa `PrintWriter` nie ma tam swojego odpowiednika. Strukturę tę przedstawia poniższy diagram (klasa `OutputStreamWriter` jest poza zakresem egzaminu):



Podstawową operacją klasy `Writer` i jej klas pochodnych jest operacja `write(…)` (tudzież `append(…)`, która ma dokładnie to samo działanie co `write(…)`) służąca do zapisu pewnej ilości znaków do źródła danych. Dla klasy `OutputStreamWriter` źródłem danych jest dowolny strumień bajtów (tj. `OutputStream`) a dla klasy `FileWriter` strumień danych oparty na pliku dyskowym. Klasa `BufferedWriter` służy do minimalizacji liczby zapisów do

źródła danych poprzez buforowanie. Dane zapisywane za pomocą metody `write(…)` są więc de facto zapisywane do bufora i stąd – gdy zachodzi taka potrzeba – do faktycznego źródła danych. Zapisywaniem bufora zarządza w pełni automatycznie klasa `BufferedWriter` ale możemy w razie potrzeby wymusić zapis wywołując metodę `flush()`. Cała zawartość bufora jest także zapisywana gdy wywołujemy metodę `close()`. Metoda `close()` zdefiniowana jest we wszystkich klasach grupy `Reader` i `Writer`. Jest to metoda którą powinniśmy wywołać zawsze gdy kończymy pracę ze strumieniem danych – ma ona za zadanie „posprzątać", tj. zwolnić używane zasoby, zamknąć otwarte połączenia czy właśnie zapisać dane z bufora.

Instancje klasy `BufferedWriter` tworzymy na podstawie instancji klasy z grupy `Writer` z użyciem jednego z dwu konstruktorów:

```
public BufferedWriter(Writer out)

public BufferedWriter(Writer out, int bufferSize)
```

Instancje klasy `OutputStreamWriter` możemy skonstruować na bazie instancji podklas klasy `OutputStream`, analogicznie jak instancje klasy `InputStreamReader` tworzymy na bazie instancji podklas klasy `InputStream`.

Instancje klasy `FileWriter` tworzymy w oparciu o plik określony albo jako obiekt klasy `File` (ewentualnie `FileDescriptor`) albo poprzez nazwę pliku, a więc obiekt klasy `String`. Sygnatura konstruktorów jest następująca:

```
public FileWriter(File file) throws IOException

public FileWriter(String fileName) throws IOException
```

Klasa `PrintWriter` służy do zapisywania sformatowanych ciągów znaków do tekstowego strumienia danych. Klasa ta dodaje do podstawowej metody `write(…)` odziedziczonej z klasy `Writer` metody `format(…)`, `print(…)`, `printf(…)` i `println(…)`, które dostarczają funkcji wyższego poziomu. Instancje klasy `PrintWriter` tworzymy z użyciem jednego z ośmiu

konstruktorów, ale dla dostatecznego zorientowania się w tej tematyce wystarczy przyjrzeć się czterem:

```
public PrintWriter(Writer out)

public PrintWriter(OutputStream out)

public PrintWriter(File file) throws FileNotFoundException

public PrintWriter(String fileName) throws FileNotFoundException
```

Klasa `Console` to jedyna z pośród prezentowanych klas pakietu `java.io`, której nie instancjonujemy sami. Klasa ta nie posiada publicznego konstruktora a jej jedyną instancję – która reprezentuje konsolę skojarzoną z Wirtualną Maszyną Javy – uzyskujemy wywołując statyczną metodę `System.console()`. Jeśli z Maszyną Wirtualną nie jest skojarzona żadna konsola metoda ta zwraca `null`.

W klasie `Console` zdefiniowano metody `reader()` i `writer()`, które zwracają odpowiednio obiekty typu `Reader` i `PrintWriter` powiązane z tą konsolą, jednak do większości zastosowań zapewne wystarczą metody `format(…)` czy `printf(…)` – analogiczne jak te zdefiniowane w klasie `PrintWriter` – oraz `readLine(…)` i `readPassword(…)`. Metody `readLine(…)` oraz `readPassword(…)` nie posiadają swego odpowiednika w klasie `Reader` – są specyficzne dla klasy `Console` – i to właśnie na nich należy się skupić studiując tę klasę. Metoda `readLine(…)` zwraca `String` reprezentujący ciąg znaków wpisany na konsoli. Metoda `readPassword(…)` nie ogranicza się do tego, że nie wyświetla wprowadzanego ciągu znaków na konsoli. Metoda ta zwraca wpisane hasło nie jako `String` tylko jako tablicę znaków (`char[]`) – ze względów bezpieczeństwa. O ile nie jest możliwe zniszczenie obiektu `String` na zawołanie, o tyle tablicę znaków możemy wypełnić tuż po weryfikacji hasła jakimiś wartościami (np. zerami) i tym samym zapewnić, że hasło nie zostanie odczytane przez hakera z użyciem techniki skanowania pamięci. Metody `readLine(…)` i `readPassword(…)` zdefiniowano w dwu wariantach: bezargumentowym oraz wieloargumentowym, który umożliwia odczytywanie danych sformatowanych – analogicznie jak metoda `format(…)`.

OPERACJE NA PLIKACH

Pliki i katalogi dyskowe w języku Java reprezentowane są przez klasę `java.io.File`. Klasy tej używamy aby sprawdzić, czy interesujący nas plik lub katalog istnieje (metoda `exists()`), do tworzenia nowych plików (metoda `createNewFile()`) oraz katalogów (metoda `mkdir()`) a także aby sprawdzić, czy dana instancja klasy `File` reprezentuje plik (metod `isFile()`) czy katalog (metoda `isDirectory()`). Klasa ta umożliwia także kasowanie oraz zmianę nazwy (ale nie ma kopiowania per se) a także udostępnia kilka mniej istotnych operacji, natomiast – co trzeba zapamiętać – nie służy do modyfikowania czy odczytywania zawartości samego pliku (do tego używamy strumieni). Zerknijmy na poniższy fragment kodu:

```
public void createFile() throws IOException {
  File file = new File("newfile.txt");

  System.out.println(file.exists());

  System.out.println(file.createNewFile());
}
```

W pierwszej linii tworzymy obiekt reprezentujący plik o nazwie „newfile.txt". Nie podaliśmy pełnej ścieżki dostępu, więc chodzi o plik znajdujący się w bieżącym katalogu. Załóżmy, że plik ten nie istnieje. Utworzenie obiektu `File` nie powoduje utworzenia pliku na dysku, więc kolejna linia kodu wyświetli tekst „false". W ostatniej linii wywołujemy metodę `createNewFile()` i tym samym tworzymy plik. Metoda ta zwraca wartość logiczną (typu `boolean`) informującą czy plik został faktycznie utworzony – w naszym wypadku będzie to wartość „true". Ponowne uruchomienie tego programu spowoduje w pierwszej kolejności wyświetlenie tekstu „true", jako że plik już istnieje, a następnie tekstu „false" – plik już istniał więc nie został utworzony tym wywołaniem. Przyjrzyjmy się jeszcze jednemu przykładowi:

Pyt.25 **Jak nazywa się metoda służąca do kopiowanie plików?**

```
public void createFiles() throws IOException {
  File directory = new File("dir");
  System.out.println(directory.mkdir());
```

```
    directory = new File("dir/subdir");
    System.out.println(directory.mkdir());

      // katalog możemy określić jako osobny obiekt typu File
    File file = new File(directory, "newfile.txt");
    System.out.println(file.createNewFile());

      // albo tekstowo, w formie ścieżki dostępu
    file = new File("dir/subdir/nextfile.txt");
    System.out.println(file.createNewFile());

    directory = new File("nextdir/subdir");
    System.out.println(directory.mkdirs());
  }
```

Załóżmy, że przed uruchomieniem powyższej metody żaden z wymienionych plików ani katalogów nie istniał. W pierwszych dwu liniach kodu tworzymy w katalogu bieżącym podkatalog o nazwie „dir". Metoda `mkdir()` podobnie jak `createNewFile()` zwraca wartość typu logicznego informującą czy katalog został utworzony. W kolejnych liniach tworzymy podkatalog o nazwie „subdir" w katalogu utworzonym poprzednio. Metoda `mkdir()` może utworzyć tylko jeden katalog na raz, tj. utworzenie katalogu „dir/subdir" nie powiodłoby się jeśli byśmy uprzednio nie utworzyli katalogu „dir". Aby utworzyć kilka (zagnieżdżonych w sobie) katalogów na raz należy użyć metody `mkdirs()` tak jak to pokazano w ostatnich liniach powyższego przykładu. Co to dokładnie oznacza, że utworzenie katalogu nie powiodłoby się? Czyżby wyjątek `IOException`? Otóż nie – metoda `mkdir()` po prostu zwróciłaby wartość `false`, zupełnie tak samo, jak w przypadku gdyby katalog nie został utworzony z powodu tego, że już istniał. Inaczej jest w przypadku, gdy w nieistniejącym katalogu spróbujemy utworzyć plik. Niezależnie od tego, czy katalog określimy tekstowo – w formie ścieżki dostępu poprzedzającej nazwę pliku – czy jako obiekt `File`, próba utworzenia pliku w katalogu który nie istnieje spowoduje rzucenie wyjątku `IOException`.

Odp.25 **Taka metoda nie istnieje. Aby skopiować plik należy utworzyć pusty plik docelowy i skopiować do niego zawartość pliku źródłowego.**

Do kasowania zbiorów dyskowych, zarówno plików jak i katalogów – ale tylko i wyłącznie pustych – służy metoda `delete()`. Podobnie jak metody tworzące metoda ta zwraca wartość typu `boolean` informującą czy zbiór dyskowy został rzeczywiście skasowany. Do zmiany nazwy używamy jednoargumentowej operacji

`renameTo(…)`, która jako argument akceptuje nie `String`, który miałby być nową nazwą, tylko obiekt `File`. W ten sam sposób zmieniamy nazwy plików i katalogów, przy czym inaczej niż w przypadku kasowania zmienić nazwę można także katalogowi który pusty nie jest. Przykład poniżej:

```
public void renameFiles() throws IOException  {
  File directory = new File("dir");
  directory.mkdir();

  File file = new File(directory, "newfile.txt");
  file.createNewFile();

  File newDirectory = new File("newdir");
  directory.renameTo(newDirectory);
}
```

Ostatnią operacją klasy `File` której potrzebujemy się przyjrzeć bliżej jest operacja `list()` która zwraca tablicę nazw plików i podkatalogów znajdujących się w danym katalogu. Przykład użycia poniżej:

```
public void listFiles() throws IOException  {
  File directory = new File("dir");
  directory.mkdir();

  File file = new File(directory, "newfile.txt");
  file.createNewFile();

  file = new File(directory, "otherfile.txt");
  file.createNewFile();

  for(String fileName : directory.list())
      System.out.println(fileName);
}
```

Do odczytu bądź modyfikacji zawartości pliku używamy strumieni, tj. klas z grupy `Reader` i `Writer`, `InputStream` i `OutputStream`. Zerknijmy na przykład użycia klas `FileReader` i `FileWriter`:

```
public class FileIO {
  public static void main(String[] args) throws IOException {
    FileIO io = new FileIO();
    io.writeToFile("newfile.txt", "tekst do pliku");

    System.out.println(io.readFromFile("newfile.txt"));
  }
```

```
  public String readFromFile(String fileName) throws IOException {
    FileReader reader = new FileReader(fileName);

    char[] buffer = new char[64];

      // nigdy nie wiadomo ile znaków wczytamy!
    int size = reader.read(buffer);

    reader.close();

      // nie cały bufor to znaki wczytane z pliku!
    return new String(buffer, 0, size);
  }

  public void writeToFile(String fileName, String txt) throws IOException {
      // jeśli plik nie istniał to ta instrukcja go utworzy!
    FileWriter writer = new FileWriter(fileName);

    writer.write(txt);

    writer.close(); // nie zapomnijmy zamknąć strumienia!
  }
}
```

Analizę kodu zacznijmy od metody `writeToFile(…)`. W pierwszej linii tworzymy instancję klasy `FileWriter` posługując się bezpośrednio nazwą pliku do którego chcemy zapisać tekst (można też użyć konstruktora akceptującego instancję klasy `File`). Jeśli plik ten nie istniał, to w wyniku wykonania tej instrukcji zostanie on utworzony. Zapamiętajmy, że utworzenie instancji klasy `File` nie powoduje utworzenia pliku (ani katalogu) na dysku, natomiast utworzenie obiektu strumienia do zapisu do pliku (niezależnie czy będzie to `FileWriter` czy inny strumień) zawsze pociąga skutek utworzenia pliku, o ile oczywiście ten jeszcze nie istniał. Tak samo jednak jak w przypadku użycia metody `createNewFile()` plik może być utworzony tylko w istniejącym katalogu (katalog nie zostanie utworzony automatycznie) – w przeciwnym wypadku otrzymamy wyjątek `IOException`. Operacja `write(…)` spowoduje zapisanie tekstu do tego pliku, nadpisując całą jego dotychczasową zawartość. Jeślibyśmy chcieli, aby tekst został do pliku dopisany, to należałoby w inny sposób utworzyć obiekt `FileWriter`, przekazując jako drugi argument wywołania konstruktora wartość `true`, mówiącą o tym, że pracujemy w trybie dopisywania. Egzamin na OCPJP nie wymaga jednak abyśmy znali wszystkie te szczegóły, więc nie wnikajmy w nie zbyt mocno. Po skończonej pracy strumień obowiązkowo zamykamy. Zawartość pliku odczytujemy z użyciem metody `read(…)` z klasy

`FileReader`. Treść zapisywana jest do bufora (tj. do tablicy znaków) a zwracana wartość informuje o liczbie odczytanych znaków z tym jednym wyjątkiem, że wartość `-1` oznacza, że nie odczytano żadnych znaków i że osiągnięto koniec pliku. Aby wczytać zawartość całego pliku powinniśmy więc w normalnej sytuacji wywoływać operację `read(…)` w pętli, aż otrzymamy wartość `-1`. Zamiast jednak trudzić się z konstrukcją w pełni poprawnego algorytmu (naturalnie możemy, ale to już poza przygotowaniami do OCPJP) przyjrzyjmy się klasie `BufferedReader`. Użycie metody `readLine()` zdefiniowanej w tejże upraszcza implementację operacji `readFromFile` do następującej postaci:

```
public String readFromFile(String fileName) throws IOException {
  FileReader reader = new FileReader(fileName);

  BufferedReader bufferedReader = new BufferedReader(reader);

  String txtLine = bufferedReader.readLine();

  bufferedReader.close(); // zamyka także strumień reader

  return txtLine;
}
```

Metoda `readLine()` zwraca linię tekstu albo wartość `null` jeśli osiągnięto koniec pliku. Nie jest to wczytanie całej treści pliku jednym wywołaniem, tak wiec aby to zrobić nadal należałoby zastosować pętlę, ale jest już znacznie łatwiej. Linia tekstu to naturalnie ciąg znaków zakończony sekwencją końca linii. Jak wiemy, sekwencja końca linii jest różna w zależności od systemu operacyjnego. Klasa `FileWriter` udostępnia w miarę wygodne w użyciu metody do zapisu do pliku z tym właśnie jednym brakiem, że sekwencję końca linii musimy skomponować z pojedynczych znaków (`\n`, `\r`) sami, wiedząc z jakim systemem operacyjnym mamy do czynienia. Analogicznie jak w przypadku odczytu z pomocą przychodzi klasa wyższego poziomu, klasa `BufferedWriter`. Zobaczmy jak mogłaby wyglądać implementacja kodu zapisującego do pliku kilka linii tekstu z wykorzystaniem metody `newLine()` i klasy `BufferedWriter`:

```
public void writeToFile(String fileName, String... txts)
    throws IOException {

  FileWriter writer = new FileWriter(fileName, false);

  BufferedWriter bufferedWriter = new BufferedWriter(writer);
```

```
  for(String line : txts) {
    bufferedWriter.write(line);

    bufferedWriter.newLine(); // zapisuje sekwencję nowej linii
  }

  bufferedWriter.close(); // zamyka także strumień writer
}
```

Funkcji wysokiego poziomu, umożliwiających zapisanie do pliku (i nie tylko do pliku) sformatowanych ciągów znaków dostarcza klasa `PrintWriter`. Oprócz standardowej operacji `write(…)` udostępnia ona metody `format(…)`, `printf(…)`, `print(…)` i `println(…)`. Metody `print(…)` i `println(…)` to nic nadzwyczajnego – akceptują jeden argument dowolnego typu, zarówno prostego jak i obiektowego i zapisują do swego źródła danych jego tekstową reprezentację. Metoda `println(…)` zapisuje dodatkowo sekwencję końca linii. Przykład poniżej:

```
public void writeToFile(String fileName) throws IOException {
  PrintWriter printWriter = new PrintWriter(fileName);

  printWriter.println(3.4F);
  printWriter.println("Zapisze wszystko");
  printWriter.print(new Object());

  printWriter.close();
}
```

Metody `format(…)` i `printf(…)` akceptują argument typu `String` określający formatowanie oraz listę argumentów dla tego formatowania. Poniżej przykład:

```
public void writeFormated() {
  PrintWriter printWriter = new PrintWriter(System.out);

  printWriter.format("PI to około %f", Math.PI);

  printWriter.close();
}
```

W miejscu powyższej metody `format(…)` moglibyśmy równie dobrze użyć metody `printf(…)`; są one najzupełniej wzajemnie równoznaczne. Obydwie zapisują sformatowany ciąg znaków do swojego strumienia, obydwie akceptują

takie same argumenty i działają w ten sam sposób. Wywołanie powyższego kodu spowoduje zapisanie do strumienia `System.out`, a więc na standardowe wyjście, tekstu:

```
PI to około 3,141593
```

Formatowanie odbywa się w ten sposób, że w miejsca gdzie występują instrukcje zaczynające się od znaku `%` wstawiane są wartości przekazanych kolejno argumentów. W powyższym przykładzie, w miejsce instrukcji `%f` wstawiona będzie wartość `Math.PI`. Znak `%` jak powiedziano rozpoczyna instrukcję formatującą. Literka `f` określa, że wartość `Math.PI` należy sformatować jako wartość zmiennoprzecinkową (z angielskiego floating point). Literka `b` określa konwersję binarną a `d` całkowitoliczbową. Pełna, długa lista i opis szczegółów formatowania w dokumentacji API dla platformy Java SE. Na potrzeby egzaminu powinny wystarczyć trzy wspomniane. Flaga formatowania binarnego działa w ten sposób, że jeśli argument jest typu innego niż binarny (kiedy to prawda jest prawdą a fałsz fałszem) to odpowiada to zawsze wartości `true`, chyba że argument ten to `null`, wtedy zostanie sformatowany do wartości `false`. Przykładowo, uruchomienie metody:

```
public void writeFormated() {
  PrintWriter printWriter = new PrintWriter(System.out);

  printWriter.format("%b", Math.PI);

  printWriter.close();
}
```

spowoduje wyświetlenie tekstu:

```
true
```

Flaga formatowania binarnego jest więc wyrozumiała w tym sensie, że można się ją posłużyć do formatowania wartości dowolnego typu i nie spowoduje to żadnego błędu. Inaczej jest z flagą konwersji całkowitoliczbowej. Uruchomienie powyższego przykładu z flagą `%d` w miejscu flagi `%b` zakończyłoby się błędem wykonania `IllegalFormatConversionException`. Flaga ta służy tylko i wyłącznie to formatowania wartości całkowitoliczbowych.

Formatowanie nie ogranicza się jednak li tylko do podstawiania wartości w miejsce instrukcji formatujących. Instrukcje formatujące można parametryzować i w ten sposób określać, jak dokładnie wartości te mają być wyświetlone. Flaga `%f` jak widzieliśmy poprzednio spowodowała wyświetlenie tylko pierwszych 6-ciu cyfr dziesiętnych liczby `Math.PI`. A gdybyśmy chcieli zmienić precyzję tak aby wyświetlić 10 cyfr? Możemy to zrobić podając przed znakiem konwersji pożądaną liczbę miejsc dziesiętnych poprzedzoną kropką. Aby więc uzyskać precyzję 10-cio cyfrową należałoby użyć formatowania `%.10f`. Przed elementem określającym precyzję możemy dodatkowo podać liczbę określającą minimalną długość liczby, przy czym jeśli liczba jest krótsza niż podana minimalna długość to wolne znaki zostaną uzupełnione spacjami. Przykładowo, uruchomienie metody:

```
public void writeFormated() {
  PrintWriter printWriter = new PrintWriter(System.out);

    // pamiętamy, że \n to znak nowej linii
  printWriter.format("PI to %10.4f\nE to %11.4f", Math.PI, Math.E);

  printWriter.close();
}
```

spowoduje wyświetlenie tekstu:

```
PI to     3,1416
E to      2,7183
```

Jako jeszcze wcześniejszy element instrukcji formatującej możemy podać flagi formatujące. I tak, jeśli chcemy aby liczba była zawsze poprzedzona znakiem (+ lub −), niezależnie czy jest to liczba ujemna czy dodatnia to należy dodać jako flagę symbol +. Jeśli chcemy aby wartość była wyrównana do lewej, a nie do prawej, w przypadku gdy określiliśmy minimalną długość liczby i jest ona większa niż faktyczna jej długość, to podajemy flagę −. Jeśli brakujące znaki liczby, krótszej niż podana minimalna długość, mają być wypełniane zerami a nie spacjami, to należy podać flagę `0`. Jeśli chcemy aby liczby ujemne były oznaczane poprzez ujęcie w nawias a nie poprzez poprzedzenie znakiem minusa to należy podać jako flagę nawias otwierający `(`.

Ostatnim elementem instrukcji formatującej, który nie dotyczy już formatowania sensu stricte jest numer argumentu którego wartość należy w miejsce danej

instrukcji podstawić. W poprzednich przykładach nie wykorzystywaliśmy tego elementu, opieraliśmy się na zachowaniu domyślnym, że pierwsza instrukcja dotyczy argumentu pierwszego, druga drugiego itd. Czasami możemy jednak chcieć uzyskać inny efekt. Możemy wtedy bezpośrednio po znaku `%` podać numer argumentu którego dana instrukcja ma użyć; za tym numerem musimy dodać znak `$`. Argumenty numerowane są inaczej niż elementy tablicy, począwszy od 1-ki. Przykład poniżej:

```
public void writeFormated() {
  PrintWriter printWriter = new PrintWriter(System.out);

  printWriter.format("E to %2$.4f a PI to %1$.4f", Math.PI, Math.E);

  printWriter.close();
}
```

Instrukcja formatująca rozpoczyna się od znaku `%`, następnie podajemy numer argumentu, którego wartość zostanie podstawiona w miejscu tej instrukcji i za tym numerem dodajemy znak `$`. Argumenty numerowane są począwszy od 1-ki, tak więc element `2$` z powyższego przykładu odnosi się więc do wartości `Math.E` a `1$` do wartości `Math.PI`. Element `.4` określa precyzję liczby, tj. 4 cyfry po przecinku, zaś literka `f` określa konwersję zmiennopozycyjną. Uruchomienie powyższej metody spowoduje wyświetlenie tekstu:

```
E to 2,7183 a PI to 3,1416
```

DATY I CZAS

Daty i czas w języku Java reprezentowane są przez klasę `Date` z pakietu `java.util`. U zarania dziejów, tj. jeszcze za czasów Javy w wersjach wcześniejszych niż 1.1 klasa ta miała pokrywać wszystkie potrzeby związane z przetwarzaniem dat a więc zaimplementowano w niej metody do modyfikacji czy formatowania, ale szybko okazało się, że nie tędy droga. Wprowadzono więc dwie nowe klasy: `java.util.Calendar` oraz `java.text.DateFormat`. Pierwsza z nich służy do konstruowania obiektów klasy `Date` reprezentujących interesujący nas moment w czasie a druga do budowania ich tekstowych reprezentacji oraz z powrotem – parsowania, tj. konwersji tekstowej reprezentacji

dat na obiekty klasy `Date`. Większość metod klasy `Date` została oznaczona jako wycofane (ang. deprecated) i nie należy już ich używać – jedyna funkcja jaką spełnia obecnie klasa `Date` to reprezentowanie określonego momentu w czasie. Zerknijmy na poniższy przykład, który ukazuje pełnię możliwości tej klasy – przynajmniej tych o których musimy wiedzieć:

```
public void dateTest() {
  Date currentTime = new Date();
  long currentTimeInMillis = currentTime.getTime();

  Date nextDay = new Date(currentTimeInMillis + 24 * 3600000);
  System.out.println(nextDay.toString());

  nextDay.setTime(nextDay.getTime() + 24 * 3600000);
  System.out.println(nextDay.toString());
}
```

Jak widać klasę `Date` można instancjonować na dwa sposoby: z użyciem konstruktora bezargumentowego – obiekt reprezentuje wówczas moment w czasie w którym został utworzony; oraz z użyciem konstruktora jednoargumentowego z argumentem typu `long` – obiekt reprezentuje wtedy datę i czas określone przez ten argument. Ten argument to ilość milisekund które upłynęły od dnia 1 stycznia 1970 roku, godziny 00:00:00. De facto czas w języku Java to właśnie wartość typu `long` – ilość milisekund które upłynęły od wspomnianej daty. Tę właśnie wartość zwraca metoda `getTime()` i to tę wartość ustawiamy wywołując metodę `setTime(…)`. Metoda `toString()` formatuje datę w ten sposób, że uruchomienie powyższego programu w dniu pisania niniejszego akapitu spowodowało wyświetlenie następującego tekstu:

```
Sat Feb 07 18:50:30 CET 2009
Sun Feb 08 18:50:30 CET 2009
```

Klasa `Calendar` jest klasą abstrakcyjną. Jej konkretnym podtypem jest klasa `java.util.GregorianCalendar`. Bezpośrednie użycie tej klasy jest możliwe, ale nie należy tego robić bez dobrego powodu. W normalnej sytuacji instancje klasy `Calendar` uzyskujemy wywołując jej statyczną metodę `getInstance(…)`. Utworzona w ten sposób instancja kalendarza reprezentuje bieżącą datę i czas. Konwersję między typami `Date` a `Calendar` umożliwiają metody `getTime()` oraz `setTime(Date date)`. Przykład poniżej:

```
public void dateTest() {
  Calendar calendar = Calendar.getInstance();

  Date date = calendar.getTime();
  System.out.println(date.toString());

  calendar.add(Calendar.YEAR, 2);

  date = calendar.getTime();
  System.out.println(date.toString());
}
```

Datą reprezentowaną przez kalendarz możemy manipulować przy użyciu metody `add(…)`. Pierwszym operandem metody jest wielkość, którą chcemy dodać (np. `Calendar.YEAR`, `Calendar.MONTH`, `Calendar.HOUR`) a drugim krotność wielkości. W powyższym przykładzie do aktualnej daty dodajemy dwa lata. Metody `add(…)` możemy użyć także do odejmowania; wystarczy że krotność będzie liczbą ujemną. Uruchomienie powyższej metody w chwili pisania tego akapitu spowodowało wyświetlenie tekstu:

```
Wed Feb 11 19:41:39 CET 2009
Fri Feb 11 19:41:39 CET 2011
```

Podobną do metody `add(…)` jest metoda `roll(…)`. Akceptuje ona takie same argumenty i także służy do dodawania bądź odejmowania wybranych składników daty a różnica polega na tym, że przesunięcia czasowe są cykliczne. Przykładowo, dodanie 24 godzin z użyciem operacji `roll(…)` nie ma żadnego skutku, podczas gdy użycie operacji `add(…)` spowodowałaby przesunięcie daty do tej samej godziny, ale następnego dnia.

Do formatowania i parsowania tekstowych reprezentacji dat służy klasa `DateFormat` z pakietu `java.text`. Podobnie jak `Calendar` jest to klasa abstrakcyjna a jej instancje tworzymy za pomocą statycznej metody-fabryki, tyle że w tym wypadku mamy do dyspozycji kilka takich metod. Jeśli chcemy aby formatowanie pomijało czas w ramach dnia, tj. interesuje nas tylko data sensu stricte to powinniśmy użyć metody-fabryki `getDateInstance(…)`. Jeśli przeciwnie, tj. interesuje nas tylko godzina, to do pobrania instancji formatera użyjemy metody `getTimeInstance(…)`. Instancję która formatuje wszystkie składniki daty zwróci metoda `getDateTimeInstance(…)`. Wszystkie

wymienione metody występują w formie bezargumentowej, która zwraca instancje używające formatowania domyślnego, oraz w formach akceptujących argumenty dzięki którym możemy określić jakiego stylu ma używać formater. Ostatnią dostępną metodą-fabryką jest `getInstance()`, która występuje tylko w wersji bezargumentowej i która zwraca instancję taką jak metoda `getDateTimeInstance(…)` poinstruowana do używania stylu `DateFormat.SHORT`. Aby użyć tak skonstruowanego formatera do konwersji obiektu daty do tekstu używamy metody `format(Date date)`:

```
public void formatTest() {
  Date currentDate = new Date();

  DateFormat shortFormat = DateFormat.getInstance();
  DateFormat defaultDateFormat = DateFormat.getDateInstance();
  DateFormat longDateFormat = DateFormat.getDateInstance(DateFormat.LONG);
  DateFormat fullFormat = DateFormat.getTimeInstance(DateFormat.FULL);

  System.out.println(shortFormat.format(currentDate));
  System.out.println(defaultDateFormat.format(currentDate));
  System.out.println(longDateFormat.format(currentDate));
  System.out.println(fullFormat.format(currentDate));
}
```

Dostępne style formatowania daty to `FULL`, `LONG`, `MEDIUM` oraz `SHORT`. Styl domyślny, a więc taki który zostanie użyty jeśli nie określimy żadnego stylu to najczęściej `MEDIUM`, ale nie jest to wyspecyfikowane. Zdefiniowanie własnych styli formatowania jest możliwe, lecz wymaga użycia innej klasy i jest to temat wykraczający poza zakres egzaminu na OCPJP. Uruchomienie powyższego kodu w chwili pisania niniejszego akapitu spowodowało wyświetlenie tekstu:

```
16.02.09 18:48
2009-02-16
16 luty 2009
18:48:28 CET
```

Aby sparsować tekst do obiektu klasy `Date` używamy metody `parse(String source)`. Przykład poniżej:

```
public void parseTest() throws ParseException {
  DateFormat mediumFormat = DateFormat.getDateInstance(DateFormat.LONG);
  Date date = mediumFormat.parse("19 luty 2008");
```

```
    DateFormat shortFormat = DateFormat.getInstance();
    date = shortFormat.parse("19.02.08 18:48");
  }
```

Zwróćmy uwagę, że obiekt formatera musi być odpowiednio dobrany pod kątem parsowanej daty. Jeśli chcemy dla przykładu sparsować datę „19 luty 2008" to musimy użyć formatera który nie wymaga określenia czasu i który używa stylu `DateFormat.LONG`. Jeśli styl daty nie będzie zgodny ze stylem obiektu formatera to parsowanie się nie powiedzie i metoda `parse(…)` rzuci wyjątek `ParseException`.

Analizując powyższe przykłady można by odnieść wrażenie, że twórcy standardu Java SE postanowili zunifikować sposób zapisu dat oraz nazwy miesięcy i że co więcej, za wzór przyjęto normy obowiązujące w Polsce. Jakże by inaczej wytłumaczyć to, że program prawidłowo sparsował datę „19 luty 2008", podczas gdy próba parsowania daty wyrażonej w języku angielskim – „February 19, 2008" – skończyłaby się błędem? Otóż klasa `DateFormat` jest bardziej uniwersalna niż mogło się początkowo wydawać – potrafi formatować i parsować daty w sposób właściwy dla wybranej lokalizacji, języka czy regionu – tyle tylko, że trzeba tę lokalizację jawnie zdefiniować. Jeśli lokalizacja nie zostanie zdefiniowana explicite to zostanie użyta lokalizacja domyślna, czyli w przypadku autora książki lokalizacja Polska. Lokalizację w języku Java reprezentuje klasa `Locale` z pakietu `java.util`. Instancje tej klasy tworzymy z użyciem jednego z trzech konstruktorów, z czego dla nas interesujące są dwa:

```
public Locale(String language)

public Locale(String language, String country)
```

Kilka najpopularniejszych lokalizacji zdefiniowano w klasie `Locale` jako stałe statyczne, dzięki temu dla określenia przykładowo języka angielskiego czy niemieckiego, oraz państw Wielka Brytania czy Niemcy możemy użyć wartości takich jak `Locale.ENGLISH`, `Locale.GERMAN`, `Locale.UK`, `Locale.GERMANY`. Lokalizację formatera określamy w momencie tworzenia obiektu i jest to własność niezmienialna. Jeśli potrzebujemy identycznego formatera, ale dla innej lokalizacji, to nie ma wyjścia, musimy utworzyć nowy obiekt. Zobaczmy teraz, jak należałoby zmodyfikować metodę `parseTest()` z

ostatniego przykładu aby parsowała datę zapisaną w językach… fińskim i francuskim:

```
public void parseTest() throws ParseException {
  DateFormat fiFormat =
    DateFormat.getDateInstance(DateFormat.LONG, new Locale("fi"));

  Date date = fiFormat.parse("19. helmikuuta 2009");

  DateFormat frFormat =       // może być także new Locale("fr")
    DateFormat.getDateInstance(DateFormat.LONG, Locale.FRENCH);

  date = frFormat.parse("19 février 2009");
}
```

Jak widać, aby skonstruować formater używający lokalizacji innej niż domyślna należy obiekt określający lokalizację przekazać jako drugi argument metody `getDateInstance(…)`. Metoda `getInstance()` występuje jak pamiętamy jedynie w wariancie bezargumentowym, a więc zawsze tworzy obiekt formatera używający lokalizacji domyślnej.

Konstruktor dwuargumentowy umożliwia zdefiniowanie zarówno języka jak i kraju. Przykładowo, data wyrażona w języku francuskim w sposób właściwy dla Francji wygląda nieco inaczej niż data wyrażona w języku francuskim w sposób właściwy dla Szwajcarii (około 13% szwajcarów deklaruje francuski jako swój język ojczysty). Kod języka (ISO-639) powinien być zapisany małymi, a kod kraju/regionu (ISO-3166) wielkimi literami:

```
public void formatTest() {
  Date currentDate = new Date();

  DateFormat frFormat =
    DateFormat.getDateInstance(DateFormat.FULL, new Locale("fr", "FR"));

  DateFormat chFormat =
    DateFormat.getDateInstance(DateFormat.FULL, new Locale("fr", "CH"));

  System.out.println(frFormat.format(currentDate));
  System.out.println(chFormat.format(currentDate));
}
```

Uruchomienie powyższego kodu w chwili pisania niniejszego akapitu spowodowało wyświetlenie tekstu:

```
lundi 23 février 2009
lundi, 23. février 2009
```

Klasa `Locale` udostępnia dwie metody których znajomość jest wymagana na egzaminie: `getDisplayCountry(…)` oraz `getDisplayLanguage(…)`. Pierwsza zwraca pełną nazwę kraju a druga nazwę języka zdefiniowane dla danego obiektu `Locale`. Metody te występują w dwu wariantach: bezargumentowym i jednoargumentowym. Wariant bezargumentowy zwraca nazwy w języku domyślnym platformy a wariant jednoargumentowy w języku określonym przez ten argument, naturalnie argument typu `Locale`. Uruchomienie metody:

Pyt.26 **Jakiej metody klasy *Date* powinniśmy użyć do sformatowania daty przed wyświetleniem jej w formie napisu?**

```
public void localeTest() {
  Locale enLocale = new Locale("en");

  Locale plLocale = new Locale("pl", "PL");

  System.out.println(plLocale.getDisplayCountry());
  System.out.println(plLocale.getDisplayLanguage());

  System.out.println(plLocale.getDisplayCountry(enLocale));
  System.out.println(plLocale.getDisplayLanguage(enLocale));
}
```

spowoduje wyświetlenie tekstu (przy założeniu że domyślnym jest język polski):

```
Polska
polski
Poland
Polish
```

Dla obiektu lokalizacji dla którego nie określono kraju a jedynie język funkcja `getDisplayCountry(…)` zwraca tekst pusty.

FORMATOWANIE I PARSOWANIE LICZB I WARTOŚCI WALUTOWYCH

Tak jak sposób zapisu dat jest zależny od języka i kraju tak od lokalizacji zależny jest również sposób zapisu liczb i wartości walutowych. Do formatowania i parsowania dat używamy klasy `DateFormat`. Analogiczną funkcję dla liczb i

walut spełnia klasa `NumberFormat`. Podobnie jak `DateFormat` jest to klasa abstrakcyjna której instancje uzyskujemy z użyciem statycznych metod-fabryk. Jeśli potrzebujemy sformatować lub sparsować wartości liczbowe to odpowiedni obiekt uzyskamy wywołując metodę `getInstance(…)` lub `getNumberInstance(…)`, które są de facto tą samą metodą – mają identyczne działanie. Obiekt formatujący dla wartości walutowych uzyskamy wywołując metodę `getCurrencyInstance(…)`.

Odp.26 **Nie powinniśmy używać metod klasy *Date* do tego celu, mimo że jest to możliwe. Do formatowania i parsowania dat służy klasa *DateFormat*.**

Wszystkie wymienione metody występują w wariantach: bezargumentowym oraz jednoargumentowym. Wariant bezargumentowy zwraca obiekt posługujący się lokalizacją domyślną platformy. Wariant jednoargumentowy umożliwia określenie lokalizacji za pomocą instancji klasy `Locale`. Przykładowo, wywołanie metody:

```
public void formatTest() {
  NumberFormat curFormat = NumberFormat.getCurrencyInstance(Locale.ITALY);
  System.out.println(curFormat.format(13.598));

  curFormat = NumberFormat.getCurrencyInstance();
  System.out.println(curFormat.format(13.598));

  NumberFormat numFormat = NumberFormat.getInstance(Locale.UK);
  System.out.println(numFormat.format(13.598));

  numFormat = NumberFormat.getInstance();
  System.out.println(numFormat.format(13.598));
}
```

spowoduje wyświetlenie tekstu (przy założeniu, że domyślną lokalizacją jest Polska):

```
€ 13,60
13,6 zł
13.598
13,598
```

Do formatowania i parsowania, analogicznie jak w przypadku pracy z datami służą metody `parse(…)` i `format(…)`. Jak widać z powyższego przykładu metoda `format(…)` oprócz tego że dobiera odpowiedni symbol dziesiętny i walutowy

wykonuje również zaokrąglenia liczb. W przypadku wartości walutowych domyślnie wyświetlane są dwie cyfry po przecinku, w przypadku wartości liczbowych trzy, ale możemy te wartości zmieniać za pomocą metod `setMaximumFractionDigits(…)` i `setMinimumFractionDigits(…)`. Możemy także odczytać aktualne wartości za pomocą metod `getMaximumFractionDigits()` i `getMinimumFractionDigits()`. Wywołanie metody:

```
public void formatTest() {
  NumberFormat curFormat = NumberFormat.getCurrencyInstance(Locale.ITALY);

  curFormat.setMinimumFractionDigits(4);
  System.out.println(curFormat.format(13.598));

  curFormat.setMaximumFractionDigits(0);
  System.out.println(curFormat.format(13.598));
}
```

spowoduje wyświetlenie tekstu:

```
€ 13,5980
€ 14
```

Istnieją także analogiczne operacje dla ustawienia oraz odczytania minimalnej i maksymalnej liczby cyfr dla części całkowitej formatowanej wartości. Ostatnią metodą klasy `NumberFormat` której należy się przyjrzeć – użyteczną tylko w kontekście parsowania – jest `setParseIntegerOnly(boolean only)`. Wywołanie tejże z wartością `true` spowoduje odrzucenie w trakcie parsowania części ułamkowej liczby:

```
public void parseTest() throws ParseException {
  NumberFormat curFormat = NumberFormat.getCurrencyInstance(Locale.ITALY);

  curFormat.setParseIntegerOnly(true);
  Number num = curFormat.parse("€ 3,60");
}
```

Zmienna `num` z powyższego przykładu będzie miała po uruchomieniu kodu wartość `3`. Metoda `parse(…)`, podobnie jak analogiczna metoda z klasy `DateFormat` w razie niemożności sparsowania przy użyciu danego formatowania rzuca wyjątek `ParseException`.

WYSZUKIWANIE WZORCA W TEKŚCIE

Algorytm wyszukiwania wzorca w tekście dla platformy Java SE implementują klasy z pakietu `java.util.regex`, tj. klasy `Pattern` i `Matcher`. Klasa `Pattern` reprezentuje skompilowany wzorzec. Używając instancji wzorca, podając tekst do przeszukania, tworzymy instancję klasy `Matcher`, której używamy do wyszukiwania. Schemat użycia jest prosty; pokazuje go poniższy przykład:

```
public void searchTest() {
  Pattern pattern = Pattern.compile("\\d+");

  Matcher matcher = pattern.matcher("a 1 b 12 c 123");

  while(matcher.find())
    System.out.println(matcher.start() + ", " + matcher.group());
}
```

Instancję klasy `Pattern` tworzymy kompilując wzorzec zapisany jako tekst za pomocą statycznej metody `compile(…)`. Następnie, używając metody `matcher(…)` na takiej instancji wzorca, przekazując tekst który chcemy przeszukiwać jako argument, tworzymy instancję klasy `Matcher`. Do iteracji po przeszukiwanym tekście używamy metody `find()`. Metoda ta wyszukuje kolejne wystąpienie wzorca w tekście. Jeśli wystąpienie zostało odnalezione metoda ta zwraca wartość logiczną `true`, w przeciwnym wypadku zwraca `false`. Metoda `start()` zwraca pozycję (numeracja od zera) pierwszego znaku ostatnio odnalezionego wystąpienia zaś metoda `group()` treść dopasowanego wystąpienia. Uruchomienie powyższej metody spowoduje wyświetlenie tekstu:

```
2, 1
6, 12
11, 123
```

W powyższym przykładzie użyto wzorca `\d+`. Ciąg `\d` oznacza dowolną cyfrę (od angielskiego digit); znak + oznacza „co najmniej jedno wystąpienie”. Wzorzec ten używany jest więc do wyszukiwania w tekście ciągów cyfr, tj. liczb naturalnych. Znak \ jest w języku Java znakiem ucieczki (ang. escape character), używanym do zapisania znaków specjalnych, np. znaku końca linii (`\n`) czy znaku tabulacji (`\t`)

oraz cudzysłowów w treści tekstów. Aby pozbawić znaku \ specjalnego znaczenia, tj. aby powiedzieć że chodzi nam o zwykły znak \ a nie o znak ucieczki, należy napisać \\. Aby więc zapisać wzorzec \d+ trzeba było de facto napisać \\d+. Napisanie samego \d+ skończyłoby się błędem kompilacji jako że \d nie jest poprawnym znakiem specjalnym.

Ciąg \d jest meta-znakiem oznaczającym dowolną cyfrę. Tego typu meta znaków zdefiniowano więcej, między innymi meta-znak \w (od angielskiego word) który odpowiada dowolnej literze, cyfrze i znakowi podkreślenia oraz \s (od angielskiego space) który odpowiada znakowi spacji. Meta-znakiem jest także kropka, która odpowiada dowolnemu znakowi. Dla określenia konkretnego znaku używamy wprost danego znaku. Możemy też określić dowolny zbiór znaków wyliczając odpowiednie znaki w nawiasie kwadratowym. Jeśli interesujące nas znaki stanowią ciągły przedział to zamiast wyliczać każdy z tych znaków możemy posłużyć się notacją myślnikową. Przykładowo, wzorzec:

```
a[b-eg]f
```

oznacza, że pierwszym znakiem musi być mała litera a, następnie może wystąpić jedna z liter z przedziału od b do e (a więc b, c, d lub e) albo litera g. Na końcu spodziewamy się małej litery f. Jak już uprzednio powiedziano, można użyć w wyrażeniu regularnym znaku + aby określić, że poprzedzający znak może wystąpić w dopasowaniu więcej niż raz. Jeśli chcielibyśmy powiedzieć, że dany znak może wystąpić w tekście dowolną ilość razy, lub nie wystąpić w ogóle, to należy po nim umieścić znak *. Umieszczając za pewnym znakiem z kolei symbol ? określamy że znak ten jest opcjonalny, tj. może wystąpić co najwyżej raz. Jeśli chcemy określić dopuszczalną krotność występowania dla całego fragmentu wyrażenia regularnego, a nie tylko dla pojedynczego znaku (lub meta-znaku) to wystarczy ten fragment ująć w nawiasy i znak +, * lub ? umieścić za tym nawiasem. Przykładowo, jeśli byśmy chcieli wyszukać w tekście wszystkie liczby zapisane w notacji dziesiętnej, zarówno dodatnie jak i ujemne oraz całkowite jak i ułamkowe to moglibyśmy posłużyć się następującym wzorcem:

```
-?\\d+(\\.\\d+)?
```

Po pierwsze definiujemy że liczba może zaczynać się od znaku minusa, ale że jest to znak opcjonalny; zapisujemy więc `-?`. Po opcjonalnym znaku minusa następuje dowolna ilość cyfr, przy czym musi wystąpić co najmniej jedna; dodajemy więc `\\d+`. Jako znak oddzielający część ułamkową od części całkowitej liczby wybieramy znak kropki, ale jak wiemy kropka jest meta-znakiem oznaczającym dowolny znak, tak więc aby określić, że chodzi nam o zwykły znak kropki musimy posłużyć się zapisem `\\.`. Podobnie jak w przypadku meta-znaku `\d` nie możemy zapisać `\.` w Javie ot tak, jako że `\` jest znakiem specjalnym dla tego języka. Musimy napisać `\\.`; w przeciwnym wypadku kod by się nie skompilował. Po kropce znowu następuje nie pusty ciąg cyfr. Część ułamkowa jest opcjonalna, tak więc cały fragment wzorca definiujący część ułamkową ujmujemy w nawias za którym umieszczamy oznaczający opcjonalność znak `?`. Zerknijmy na jeszcze jeden przykład i spróbujmy odgadnąć, jaki będzie efekt wywołania tej funkcji:

```
public void searchTest() {
  Pattern pattern = Pattern.compile("[aA]..");

  Matcher matcher = pattern.matcher("Ala Ula Ela Ola");

  while(matcher.find())
    System.out.println(matcher.start() + ", " + matcher.group());
}
```

Po tym co widzieliśmy przed chwilą wzorzec `[aA]..` nie powinien nas zaskoczyć czy zakłopotać; odpowiada on wszystkim ciągom trój-znakowym zaczynającym się od wielkiej bądź małej litery a. Ile zatem takich ciągów znajduje się w przeszukiwanym przez powyższą funkcję tekście? Cztery; są to: „Ala”, „a U”, „a E” i „a O”. Tyle, że wywołanie metody spowoduje wyświetlenie tekstu:

```
0, Ala
6, a E
10, a O
```

Brakuje więc ciągu „a U”. Tym samym doszliśmy do momentu w którym trzeba nieco precyzyjniej odpowiedzieć na pytanie: jak właściwie działa wyszukiwanie wzorca zaimplementowane w klasie `Matcher`. Odpowiedź szczęśliwie jest prosta. Tekst przeszukiwany jest od lewej do prawej i każdy znak w przeszukiwanym tekście może być elementem tylko jednego dopasowania, tzn. wyszukiwanie kolejnego dopasowania rozpoczyna się od znaku następującego za dopasowaniem ostatnio odnalezionym. Symulując zachowanie funkcji `find()` na danych z

powyższego kodu zauważamy więc, że skoro jako pierwszy dopasowany był ciąg „Ala" to poszukiwanie kolejnego dopasowania rozpoczęto od następującego po wyrazie „Ala" znaku spacji, tak więc dopasowanie „a U" nie zostało odnalezione.

TOKENIZACJA TEKSTU

Tokenizacja to proces w wyniku którego monolityczny tekst zostaje podzielony na ciąg pojedynczych tokenów. Tokeny to ciągi znaków ograniczone ustalonymi separatorami takimi jak spacje czy przecinki, aczkolwiek separatorem może być dowolny ciąg który da się opisać w postaci wyrażenia regularnego. Najbardziej elementarny algorytm tokenizacji w Javie implementuje metoda `split(…)` z klasy `String`. Metoda ta dokonuje podziału tekstu reprezentowanego przez obiekt dla którego została wywołana używając separatora opisanego wyrażeniem regularnym przekazanym jako argument wywołania. Przykład poniżej:

```
public void tokenize() {
  String[] tokens = "dowolny tekst do tokenizacji".split("\\s");

  for(String token : tokens)
    System.out.println(token);
}
```

Wyrażenie regularne `\s` (które musimy zapisać jako `\\s`) oznacza biały znak (ang. whitespace character), tj. spację, tabulację, znak nowej linii itd. Uruchomienie powyższej metody spowoduje więc wyświetlenie tekstu:

```
dowolny
tekst
do
tokenizacji
```

Bardziej wyrafinowanych mechanizmów tokenizacji dostarcza klasa `Scanner` z pakietu `java.util`. Zasadnicza różnica w stosunku do metody `split(…)` polega na tym, że tokenizowany tekst jest przetwarzany stopniowo, w miarę potrzeby, i proces może być w dowolnej chwili zaniechany. Skanerem posługujemy się jak iteratorem. Klasa `Scanner` implementuje interfejs `java.util.Iterator<String>` więc de facto jest ona iteratorem; pewnym

szczególnym rodzajem iteratora, dzięki któremu możemy iterować po kolejnych tokenach przetwarzanego tekstu. Obiekt skanera tworzymy z użyciem jednego z kilku konstruktorów, z czego interesującymi dla nas są trzy pokazane poniżej:

```
public Scanner(String source)

public Scanner(File source) throws FileNotFoundException

public Scanner(InputStream source)
```

Identyczna funkcjonalnie metoda jak pokazana w pierwszym przykładzie, tyle że zaimplementowana przy użyciu skanera mogłaby więc wyglądać następująco:

```
public void tokenize() {
  Iterator<String> scanner = new Scanner("dowolny tekst do tokenizacji");

  while (scanner.hasNext())
    System.out.println(scanner.next());
}
```

Wnikliwy czytelnik z pewnością zauważył, że w powyższym przykładzie nie określono separatora; w takim wypadku skaner używa separatora domyślnego którym są białe znaki, stąd równoważność funkcjonalna z przykładem używającym metody `split(…)`. Aby ustawić separator – opisany jako wyrażenie regularne – należy użyć metody `useDelimiter(…)` co pokazano poniżej:

```
public void tokenize() {
  Scanner scanner = new Scanner("dowolny tekst do tokenizacji");

  scanner.useDelimiter("\\s");

  while (scanner.hasNext())
    System.out.println(scanner.next());
}
```

Oprócz metod `hasNext()` i `next()` pochodzących z interfejsu `Iterator<String>`, które zwracają wartości typu `String`, klasa `Scanner` implementuje metody `hasNextInt()` i `nextInt()`, `hasNextBoolean()` i `nextBoolean()` itd. dla pozostałych typów prostych. Przy użyciu tych metod możemy w łatwy sposób zaimplementować np. funkcję sumującą liczby naturalne występujące w danym tekście:

```
public int sum() {
  Scanner scanner = new Scanner("a 1 b 2 c 3 d e f");

  int sum = 0;

  while (scanner.hasNext()) {
    if (scanner.hasNextInt()) {
      sum += scanner.nextInt();
    } else {
      System.out.println("nie int: " + scanner.next());
    }
  }

  return sum;
}
```

KOLEKCJE I TYPY GENERYCZNE

Niniejszy rozdział pokrywa prezentowanym zakresem wiedzy następujące wymagania egzaminu 1Z0-851:

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

6.2 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 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 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 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`.

METODY EQUALS() I HASHCODE()

Zacznijmy od początku, a mianowicie od pytania – do czego właściwie służy metoda `equals(…)`; przecież mamy operator równości ==? Mówiąc najprościej, operator równości służy do sprawdzenia, czy dwa obiekty są dokładnie tym samym (tym samym obiektem), zaś metoda `equals(…)` odpowiada na pytanie, czy dwa obiekty są „takie same". Sformułowanie „takie same" ujęto w cudzysłów, bowiem co ono oznacza zależy tylko od naszego uznania, a mówiąc ściślej, właśnie od implementacji metody `equals(…)`. Metoda `equals(…)` jest zdefiniowana w klasie `java.lang.Object` w ten sposób, że jest tożsama z operatorem równości ==, ale w wielu przypadkach nie jest to implementacja właściwa. Dla przykładu, w klasie `java.lang.Integer` metodę tę nadpisano w ten sposób, że dwa obiekty x i y są uznawane za „takie same" (tzn. `x.equals(y) == true`), jeśli reprezentują tę samą wartość liczbową, tj. `x.intValue() == y.intValue()`.

Powiedzieliśmy sobie przed chwilą, że znaczenie sformułowania „takie same", a więc implementacja metody `equals(…)`, mogą być dowolne i zależą wyłącznie od naszego uznania, ale nie do końca jest to prawdą. Metodę `equals(…)` obowiązuje bowiem kontrakt, którego musimy przestrzegać. Oto on:

- Relacja wyznaczona metodą `equals(…)` musi być zwrotna, tj. dla każdej zmiennej referencyjnej `x` (różnej od `null`) wyrażenie `x.equals(x)` musi mieć wartość `true`.

- Relacja wyznaczona metodą `equals(…)` musi być symetryczna, tj. dla każdej pary zmiennych referencyjnych `x` i `y`, wyrażenie `x.equals(y)` ma wartość `true` wtedy i tylko wtedy gdy `y.equals(x)` ma wartość `true`. Chciałoby się powiedzieć, że `x.equals(y)` musi dawać dokładnie ten sam wynik co `y.equals(x)`, ale tak nie jest. Jeśli bowiem `x` wskazuje na pewien obiekt a `y` ma wartość `null`, to `x.equals(y)` ma wartość `false` (musi być `false`, co jest kolejnym punktem kontraktu) a wywołanie `y.equals(x)` spowoduje rzucenie wyjątku

`NullPointerException`.

- Relacja wyznaczona metodą `equals(…)` musi być przechodnia, tj. dla dowolnych zmiennych referencyjnych `x`, `y` i `z`, jeśli `x.equals(y)` ma wartość `true` oraz `y.equals(z)` ma wartość `true` to także `x.equals(z)` musi mieć wartość `true`.

- Przy założeniu, że porównywane obiekty się w międzyczasie nie zmieniają, każdorazowe wywołanie funkcji `x.equals(y)` dla tej samej pary zmiennych `x`, `y` musi dawać taki sam wynik, tj. albo zawsze `true` albo zawsze `false`.

- Każdy obiekt musi być różny od wartości `null`, tj. wywołanie `x.equals(null)` musi zawsze zwracać wartość `false`.

Ściśle związaną z metodą `equals(…)` jest operacja `hashCode()`. Operację `hashCode()` również obowiązuje pewien kontrakt i to taki, który definiuje jej zachowanie w zależności od zachowania metody `equals(…)`. Stąd w typowej sytuacji, ilekroć nadpisujemy jedną z tych metod musimy nadpisać też i drugą. Oto kontrakt dla metody `hashCode()`:

- Przy założeniu, że obiekt się w międzyczasie nie zmienia, każdorazowe wywołanie funkcji `hashCode()` dla tego obiektu musi zwracać – w ramach pojedynczego uruchomienia aplikacji – taką samą wartość.

- Jeśli dwa obiekty `x` i `y` są „takie same”, tj. `x.equals(y)` ma wartość `true`, to muszą one mieć takie same wartości skrótu (ang. hash code) – musi zachodzić równość `x.hashCode() == y.hashCode()`. Zauważmy, że implikacja działa tylko w jedną stronę, tj. jeśli obiekty „są różne” (tj. nie są „takie same”), to ich wartości skrótu wcale nie muszą być różne.

Poniżej pokazano przykład klasy nadpisującej metody `equals(…)` i `hashCode()`. Metody te zaimplementowano w oparciu o zmienną prywatną klasy:

```
public class MyClass {
  private int someValue;

  public int hashCode() {
    return someValue;
  }

  public boolean equals(Object obj) {
    if (obj == null || getClass() != obj.getClass())
      return false;

    if (someValue != ((MyClass) obj).someValue)
      return false;

    return true;
  }
}
```

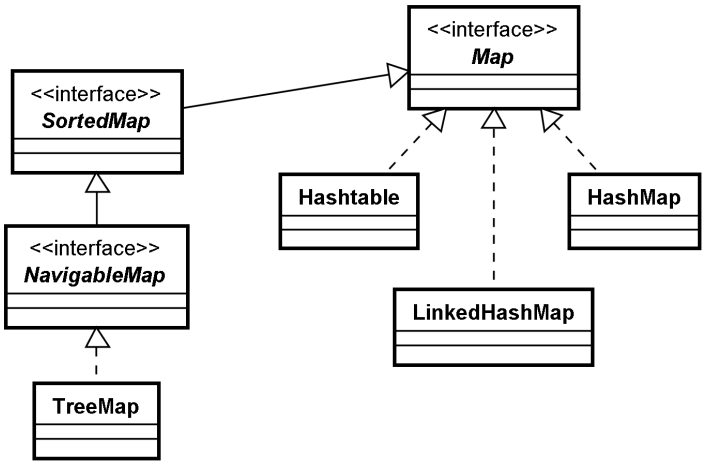
Dwie instancje klasy `MyClass` będą uznane za takie same, jeśli wartości ich zmiennej prywatnej `someValue` będą takie same. Zwróćmy szczególną uwagę na sygnatury metod `equals(…)` i `hashCode()`. Jeśli na egzaminie OCPJP pojawi się pytanie dotyczące tych metod to w pierwszej kolejności upewnijmy się, że pokazane metody to rzeczywiście nadpisania a nie przeciążenia.

STRUKTURY DANYCH

Ktoś kiedyś napisał, że programy to algorytmy plus struktury danych i nie ma w tym stwierdzeniu wiele przesady. Umiejętność posługiwania się strukturami danych – kolekcjami i mapami – to abecadło sprawnego programisty. Szczęśliwie dawno minęły już czasy, kiedy struktury takie jak listy czy zbiory i operacje na tychże trzeba było implementować samodzielnie; obecnie wystarczy nauczyć się używać implementacji już istniejących, dostarczanych w postaci biblioteki. Taką biblioteką dla języka Java jest pakiet `java.util` będący częścią platformy Java SE.

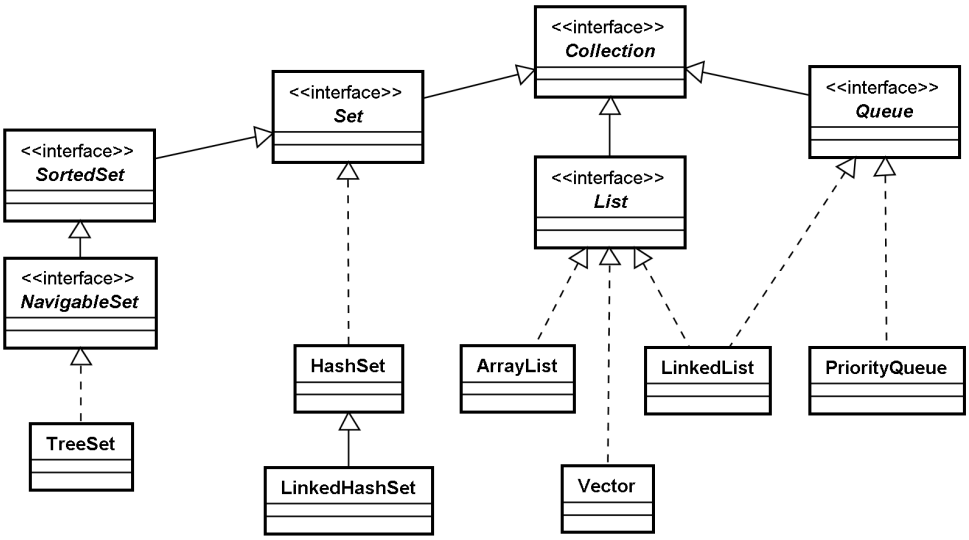
Struktury danych w języku Java podzielone są na dwie grupy: kolekcje, a więc wszystkie klasy implementujące interfejs `Collection`; oraz mapy, a więc klasy implementujące interfejs `Map`. Kolekcje reprezentują grupy obiektów będących elementami tych kolekcji. Niektóre kolekcje dopuszczają wielokrotne występowanie tego samego elementu a inne nie. Mapy reprezentują związki

obiektów z pewnymi kluczami, które identyfikują te obiekty. Dany klucz może być elementem mapy co najwyżej raz, tak więc w danej mapie może być powiązany z co najwyżej jednym obiektem. Mapy można postrzegać jako pewien wyspecjalizowany rodzaj kolekcji; kolekcji par klucz-wartość. Interesujące nas klasy i interfejsy map pokazuje poniższy diagram:



Kolekcje dzielą się na zbiory, czyli klasy implementujące interfejs `Set`; listy – klasy implementujące interfejs `List`; oraz kolejki – klasy implementujące interfejs `Queue`. Zbiory charakteryzują się tym, że nie mogą zawierać duplikatów, tj. nie mogą zawierać dwu równych (równość badana jest za pomocą metody `equals(…)`) elementów.

W sposób naturalny zbiory pasują wszędzie tam, gdzie istotny jest sam fakt należenia bądź nie należenia elementu do zbioru oraz tam gdzie chodzi o proste przechowywanie pewnej grupy elementów. Listy to kolekcje uporządkowane w których z kolei szczególnie istotna jest kolejność elementów. Umożliwiają one operowanie na swych elementach z użyciem indeksów, np. pobranie elementu znajdującego się na konkretnej pozycji listy. Kolejki zaprojektowane są do przechowywania elementów które powinny być przetwarzane kolejno, zgodnie z pewnym porządkiem (np. porządkiem wstawiania, tzw. kolejki FIFO). Hierarchię klas i interfejsów kolekcji ilustruje poniższy diagram:



MAPY

Mapy w języku Java to klasy implementujące interfejs `java.util.Map<K,V>`. Interfejs ten definiuje operacje wspólne dla wszystkich rodzajów map. Podstawowe trzy to operacja dodawania nowego skojarzenia, operacja pobierania wartości skojarzonej z danym kluczem oraz operacja usuwania istniejącego skojarzenia, odpowiednio:

```
V put(K key, V value)

V get(Object key)
```

oraz:

```
V remove(Object key)
```

Zarówno klucz jak i wartość skojarzenia są obiektami, tj. żadne z nich nie może być wartością prostą. Wartości proste mogą być jednak używane w skojarzeniach

mapy w formie opakowanej; jawnie, bądź za sprawą auto-boxingu. Gdybyśmy dla przykładu chcieli sporządzić ewidencję mieszkań w naszym bloku moglibyśmy użyć mapy z kluczem będącym numerem mieszkania – tj. liczbą opakowaną w obiekt klasy `Integer` – i wartością będącą instancją jakiejś samodzielnie zaimplementowanej klasy opisującej mieszkanie, tak jak pokazano w poniższym przykładzie:

```
public class TestClass {
  public static void main(String[] args) {
    Map<Integer, Flat> flats = new HashMap<Integer, Flat>();
      // auto-boxing wartości prostej 12 do typu Integer
    flats.put(12, new Flat());
      // obiekt zostanie odnaleziony i przypisany do zmiennej
    Flat flat = flats.get(12);
  }
}

class Flat {
  // dowolna implementacja klasy
}
```

Załóżmy teraz, że chcemy powyższy przykład nieco rozszerzyć, tak aby prowadzić ewidencję mieszkań dla całego osiedla, nie tylko dla własnego bloku. Wymagać to będzie zmiany klucza tak, aby oprócz numeru mieszkania zawierał także numer bloku. Użyjmy więc jako klucza instancji klasy opisującej adres, tak jak to pokazano poniżej:

```
public class TestClass {
  public static void main(String[] args) {
    Map<Address, Flat> flats = new HashMap<Address, Flat>();
      // blok nr. 8, mieszkanie 104
    flats.put(new Address(8, 104), new Flat());
      // obiekt nie zostaje odnaleziony, mimo że tam jest
    Flat flat = flats.get(new Address(8, 104));
  }
}

class Address {
  int building;
  int flat;

  public Address(int building, int flat) {
    this.building = building;
    this.flat = flat;
  }
}
```

Mogłoby się wydawać, że powyższy kod jest poprawny, ale jednak nie – zmienna `flat` z metody `main(…)` ma wartość `null`. Skojarzenie z podanym kluczem nie zostało odnalezione. Dlaczego? Otóż dlatego, że zgodnie z implementacją metod `equals(…)` i `hashCode()` odziedziczoną przez klasę `Address` z klasy `Object` obiekt `new Address(8, 104)` utworzony w chwili dodawania skojarzenia do mapy nie jest równy obiektowi `new Address(8, 104)` utworzonemu w chwili wyszukiwania skojarzenia w mapie. My naturalnie chcemy aby wszystkie adresy wskazujące na ten sam blok i te same mieszkanie w ramach bloku były uznawane za równe. Aby to osiągnąć musimy w klasie `Address` zaimplementować odpowiednio metody `equals(…)` i `hashCode()`.

Pyt.27 **Jaki warunek powinien spełniać typ, którego używamy jako klucz mapy?**

Interfejs `Map<K,V>` definuje także operacje które pozwalają sprawdzić czy mapa zawiera skojarzenie o danym kluczu lub z daną wartością, są to odpowiednio:

```
boolean containsKey(Object key)

boolean containsValue(Object value)
```

Przydatne mogą się ponadto okazać funkcje które pozwalają widzieć mapę jako kolekcję jej kluczy, wartości lub par klucz-wartość, są to odpowiednio:

```
Set<K> keySet()

Collection<V> values()
```

oraz:

```
Set<Map.Entry<K,V>> entrySet()
```

Co ważne, operacje te nie tworzą autonomicznych kolekcji. Wprost przeciwnie. Kolekcje te są ściśle powiązane z mapami na bazie których zostały utworzone. Są to kolekcje-widoki. Przykładowo, usunięcie elementu takiej kolekcji (dowolnej z nich) powoduje usunięcie odpowiadającego skojarzenia także z mapy. Ilustruje to poniższy przykład:

```
public void rmTest() {
  Map<String, Object> map = new HashMap<String, Object>();
  map.put("A", new Object());

  System.out.println(map.containsKey("A"));
  Set<String> set = map.keySet();

    // usunięcie klucza ze zbioru kluczy usuwa także skojarzenie z mapy
  set.remove("A");

  System.out.println(map.containsKey("A"));
}
```

Zaczynamy od utworzenia obiektu mapy. Następnie dodajemy do mapy skojarzenie z kluczem „A". Używamy operacji `containsKey(…)` aby potwierdzić, że mapa zawiera odpowiedni klucz. Za pomocą operacji `keySet()` tworzymy kolekcję zawierającą klucze tej mapy. Ze zbioru kluczy usuwamy klucz „A". Jeszcze raz sprawdzamy – za pomocą operacji `containsKey(…)` – czy mapa zawiera klucz „A". Nie zawiera, klucz został usunięty; wynikiem wywołania procedury jest tekst:

```
true
false
```

Z pośród interesujących nas klas aż trzy implementują interfejs `Map<K,V>` bezpośrednio; są to: `HashMap<K,V>`, `Hashtable<K,V>` oraz `LinkedHashMap<K,V>`. Klasa `HashMap` to podstawowa, wydajna implementacja, która jednak nie zapewnia żadnego określonego porządku sortowania skojarzeń. Klasa `Hashtable` tym różni się od klasy `HashMap`, że jej metody są synchronizowane (a więc bezpieczne w środowisku wielowątkowym, ale zarazem wolniejsze) a ponadto ani kluczem ani wartością skojarzenia nie może być `null` (klasa `HashMap` dopuszcza `null` zarówno jako klucz jak i jako wartość skojarzeń). Klasa `LinkedHashMap` łączy charakterystykę klasy `HashMap` z korzyściami wynikającymi z dodatkowego powiązania skojarzeń mapy w listę dwukierunkową, co zapewnia sortowanie skojarzeń – w porządku zgodnym z kolejnością dodawania (ang. insertion-order) albo ostatniego dostępu (ang. access-order) – oraz szybszą iterację po elementach mapy.

Odp.27 **Instancje tego typu powinny mieć odpowiednio zaimplementowane metody *equals(…)* i *hashCode()*.**

Interfejs `SortedMap<K,V>` rozszerza interfejs `Map<K,V>` o metody bazujące na posortowaniu skojarzeń według wartości klucza. Są to metody pozwalające na utworzenie mapy-córki będącej widokiem wybranego fragmentu (spójnego względem danego porządku sortowania) mapy-matki. Możemy więc zbudować mapę-widok zawierającą skojarzenia z kluczami o wartościach ściśle mniejszych lub większych bądź równych od podanej wartości. Możemy także określić obydwa końce przedziału wartości kluczy. Metody te to odpowiednio:

```
  // klucze ściśle mniejsze niż toKey
SortedMap<K,V> headMap(K toKey)

  // klucze większe bądź równe fromKey
SortedMap<K,V> tailMap(K fromKey)
```

oraz:

```
  // klucze większe lub równe fromKey
  // i jednocześnie ściśle mniejsze niż toKey
SortedMap<K,V> subMap(K fromKey, K toKey)
```

Rozszerzone wersje wymienionych powyżej metod, umożliwiające określenie czy mapa-córka ma zawierać wartości włącznie z tymi przekazanymi jako parametry wywołania czy nie (tj. czy parametry te określają przedział otwarty czy domknięty) definiuje interfejs `NavigableMap<K,V>` (który rozszerza interfejs `SortedMap<K,V>`). Są to metody:

```
  // włącznie z kluczem toKey jeśli inclusive ma wartość true
NavigableMap<K,V> headMap(K toKey, boolean inclusive)

  // włącznie z kluczem fromKey jeśli inclusive ma wartość true
NavigableMap<K,V> tailMap(K fromKey, boolean inclusive)
```

oraz:

```
  // włącznie z kluczem fromKey jeśli fromInclusive ma wartość true
  // włącznie z kluczem toKey jeśli toInclusive ma wartość true
NavigableMap<K,V> subMap(K fromKey,
                         boolean fromInclusive,
                         K toKey,
                         boolean toInclusive)
```

Podobnie jak usunięcie elementu z kolekcji-widoku (kolekcji kluczy, wartości bądź par klucz-wartość) powiązanej z mapą powoduje usunięcie odpowiadającego skojarzenia z mapy, tak usunięcie skojarzenia z mapy-córki (która jest mapą-widokiem) powoduje usunięcie skojarzenia z mapy-matki. A co by się stało, gdybyśmy do kolekcji kluczy (zwróconej przez operację `keySet()`) zechcieli dodać nową wartość? Został by rzucony wyjątek `UnsupportedOperationException`. Nie wiadomo przecież jaką wartość należy w takiej sytuacji przypisać temu kluczowi w powiązanej mapie a mapy nie wspierają operacji dodawania samych kluczy. Nie ma tego problemu w przypadku mapy-córki; możemy dodać do niej nowe skojarzenie, co spowoduje dodanie tego skojarzenia również do mapy-matki. Zerknijmy na przykład:

```
public void addTest() {
  NavigableMap<String, Object> map = new TreeMap<String, Object>();

  map.put("A", new Object());
  map.put("D", new Object());
  map.put("G", new Object());

  System.out.println(map.containsKey("B"));

  NavigableMap<String, Object> subMap = map.headMap("D", true);
  subMap.put("B", new Object());

  System.out.println(map.containsKey("B"));
}
```

W pierwszej linii kodu deklarujemy zmienną `map` typu `NavigableMap` i przypisujemy jej nowo utworzoną instancję klasy `TreeMap`. Klasa `TreeMap` jest jedyną z pośród interesujących nas klas implementującą ten interfejs. Następnie do tej mapy dodajemy trzy skojarzenia z kluczami „A”, „D” i „G”. Wywołanie metody `containsKey(…)` z wartością „B” w kolejnej linii naturalnie zwraca wartość `false`. Klasa `TreeMap` domyślnie sortuje klucze zgodnie z porządkiem naturalnym, który w przypadku kluczy typu `String` jest porządkiem leksykograficznym, tak więc mapa-córka `subMap` utworzona następnie przy użyciu operacji `headMap(…)` zawiera skojarzenia z kluczami „A” i „D”. Jak już sobie powiedzieliśmy dodanie skojarzenia do mapy-córki, co czynimy w kolejnej linii przy użyciu metody `put(…)`, powoduje dodanie tego skojarzenia także do mapy-matki a więc kolejne wywołanie metody `containsKey(…)` z wartością „B” zwraca wartość `true`.

Powiązanie mapy-matki i mapy-córki jest obustronne, tj. wszystkie operacje wykonywane na mapie-matce odzwierciedlone są bezpośrednio na skojarzeniach w mapie-córce. Ale co się stanie, gdy do mapy-matki dodamy skojarzenie z kluczem którego wartość nie należy do zakresu kluczy zawartych w mapie-widoku (mapie-córce)? Uruchomienie poniższego kodu:

```
public void addTest() {
  NavigableMap<String, Object> map = new TreeMap<String, Object>();

  map.put("A", new Object());
  map.put("D", new Object());
  map.put("G", new Object());

  NavigableMap<String, Object> subMap = map.headMap("D", true);

    // dodajemy do mapy-matki skojarzenie z kluczem B
  map.put("B", new Object());

    // jest ono widoczne także w mapie-córce
  System.out.println(subMap.containsKey("B"));

    // dodajemy do mapy-matki skojarzenie z kluczem F
  map.put("F", new Object());

    // F jest większe od D więc skojarzenie nie jest widoczne w mapie-córce
  System.out.println(subMap.containsKey("F"));
}
```

spowoduje wyświetlenie tekstu:

```
true
false
```

Mapę `subMap` zdefiniowano jako mapę-widok zawierającą te skojarzenia z mapy `map`, których klucz jest (w rozumieniu domyślnego porządku naturalnego) mniejszy bądź równy „D”. Skojarzenie z kluczem „B” dodane do mapy `map` jest więc odzwierciedlone także poprzez mapę `subMap`, ale skojarzenie z kluczem „F” już nie. A co by się stało, gdybyśmy skojarzenie z kluczem „F” zechcieli dodać do mapy `subMap`? Otrzymalibyśmy wymowny wyjątek:

```
java.lang.IllegalArgumentException: key out of range
      at java.util.TreeMap$NavigableSubMap.put(TreeMap.java:1386)
      at my.pckg.TestClass.addTest(TestClass.java:22)
      at my.pckg.TestClass.main(TestClass.java:10)
```

Chcieliśmy przecież, aby mapa `subMap` zawierała tylko skojarzenia o kluczu mniejszym bądź równym „D" a więc klucz „F" słusznie nie może być dodany, ponieważ jest poza tym zakresem.

Domyślnie elementy mapy implementującej interfejs `NavigableMap` są posortowane rosnąco. Aby otrzymać mapę-córkę będącą wiernym odbiciem oryginalnej mapy, ale sortującą swe skojarzenia w porządku malejącym możemy się posłużyć operacją:

```
NavigableMap<K,V> descendingMap()
```

Interfejs `NavigableMap<K,V>` definiuje jeszcze dość pokaźną liczbę operacji pozwalających na wyszukiwanie kluczy i skojarzeń. Możemy więc znaleźć klucz większy, większy bądź równy, mniejszy lub mniejszy bądź równy niż zadany. Służą do tego metody odpowiednio:

```
K higherKey(K key)

K ceilingKey(K key)

K lowerKey(K key)

K floorKey(K key)
```

Analogicznie, możemy także pobrać parę klucz-wartość o kluczu większym, większym bądź równym, mniejszym lub mniejszy bądź równym niż zadany. Dodatkowo, mapą możemy posłużyć się jak kolejką dwustronną tj. pobrać z niej pierwszą (z najmniejszym kluczem) lub ostatnią (z kluczem największym) względem danego porządku sortowania parę, jednocześnie usuwając ją z mapy. Służą do tego metody odpowiednio:

```
Map.Entry<K,V> higherEntry(K key)

Map.Entry<K,V> ceilingEntry(K key)

Map.Entry<K,V> lowerEntry(K key)

Map.Entry<K,V> floorEntry(K key)
```

```
  // pobiera i jednocześnie usuwa z mapy
Map.Entry<K,V> pollFirstEntry()

  // pobiera i jednocześnie usuwa z mapy
Map.Entry<K,V> pollLastEntry()
```

KOLEKCJE

Kolekcje w języku Java to klasy implementujące interfejs `java.util.Collection<E>`. Podstawową funkcją kolekcji jest przechowywanie grupy elementów, toteż każda kolekcja udostępnia operację dodawania elementów:

```
boolean add(E e)
```

Operacja ta jednak sensu stricte nie tyle dodaje do kolekcji nowy element, co pozwala upewnić się, że taki element do kolekcji – po wywołaniu tej metody – będzie należał. Kolekcja jest bytem abstrakcyjnym i nieokreślonym precyzyjnie, toteż metody – te zdefiniowane na poziomie interfejsu `Collection<E>` – są określone na wysokim stopniu ogólności. Faktyczne działanie tych operacji jest różne dla różnych rodzajów kolekcji, dostosowane do ich specyfiki. Metoda `add(…)` zdefiniowana w interfejsie `Set<E>` dodaje element do zbioru tylko wtedy, kiedy taki element (tj. element równy według metody `equals(…)`) jeszcze do zbioru nie należy. Zbiory nie dopuszczają przecież możliwości przechowywania duplikatów. Zwracana wartość logiczna informuje, czy element został faktycznie dodany (wartość `true`) czy nie (wartość `false`). W przypadku list, a więc klas implementujących interfejs `List<E>` metoda `add(…)` dodaje nowy element zawsze, dołączając go do końca listy. Interfejs `List<E>` dodatkowo definiuje operację:

```
void add(int index, E element)
```

która umożliwia dodanie elementu na dowolnie wybranej pozycji, wskazanej przez pierwszy parametr wywołania. Kolejki, a więc klasy implementujące interfejs `Queue<E>` przechowują elementy posortowane zgodnie z pewnym porządkiem a więc operacja `add(…)` dodaje nowy element poprzez wsortowanie go na odpowiedniej pozycji. Aby wsortowanie było możliwe nowo dodawany element

musi być porównywalny z elementami obecnymi w kolejce. W typowej sytuacji oznacza to, że elementy kolejki muszą być tego samego typu; w przeciwnym wypadku uruchomienie operacji `add(…)` zakończy się wyjątkiem `ClassCastException`. Wyjątkiem takim zakończy się dla przykładu wywołanie poniższego kodu:

```
public static void main(String[] args) {
  Queue<Number> queue = new PriorityQueue<Number>();

  queue.add(new Integer(12));

    // błąd! - typ Long nie jest porównywalny z typem Integer
  queue.add(new Long(5));
}
```

W łatwy sposób możemy przekonać się czy kolekcja jest pusta, jaki jest rozmiar kolekcji (tj. ile zawiera elementów) oraz czy zawiera konkretny, dany element. Służą do tego odpowiednio operacje:

```
boolean isEmpty()

int size()

boolean contains(Object o)
```

Listy poza tym, że implementują oczywiście operację `contains(…)`, która pozwala stwierdzić czy dany element do tej listy należy, pozwalają ustalić indeks wystąpienia elementu. Operacja:

```
int indexOf(Object o)
```

zwraca indeks pierwszego wystąpienia elementu na liście lub wartość `-1` jeśli element ten na liście nie występuję, a operacja:

```
int lastIndexOf(Object o)
```

analogicznie, indeks wystąpienia ostatniego. Metoda:

```
E get(int index)
```

pozwala z kolei pobrać element znajdujący się na zadanej pozycji listy. Jeśli indeks jest z poza dozwolonego zakresu, tj. jest liczbą ujemną lub nie mniejszą niż liczba elementów kolekcji (pierwszy element ma indeks 0) to rzucony zostanie wyjątek `IndexOutOfBoundsException`.

Wszystkie kolekcje umożliwiają ponadto iterację po swoich elementach. Interfejs `Collection<E>` definiuje metodę:

```
Iterator<E> iterator()
```

która zwraca obiekt iteratora po elementach kolekcji. Porządek iteracji jest w ogólności nieokreślony, tj. jeśli kolekcja sama w sobie nie zapewnia uporządkowania swych elementów to kolejność elementów zwracanych przez iterator jest nie tylko nieznana, ale może być także zmienna (różne iteratory dla tej samej kolekcji mogą zwracać elementy w różnej kolejności). Jeśli kolekcja jest uporządkowana to iterator zwraca elementy w kolejności zgodnej z tym porządkiem. Kolejne elementy z iteratora pobieramy za pomocą metody:

```
E next()
```

ale zanim to zrobimy, za pomocą metody:

```
boolean hasNext()
```

upewniamy się, że iterator nie jest pusty. Typowy przykład użycia iteratora – w tym przypadku iteratora dla listy, zachowującego uporządkowanie elementów z tejże – pokazuje poniższy przykład:

```
public static void main(String[] args) {
  List<String> list = Arrays.asList("Ola", "Ala", "Ula");

  Iterator<String> iterator = list.iterator();

  while(iterator.hasNext())
    System.out.println(iterator.next());
}
```

W pokazanym powyżej przykładzie konstruujemy listę poprzez konwersję tablicy (dynamiczna lista argumentów wywołania funkcji jest tożsama z tablicą tych

argumentów), przy użyciu statycznej metody `asList(…)` z klasy `Arrays`. Sygnatura tej metody jest następująca:

```
public static <T> List<T> asList(T... a)
```

Funkcjonalność odwrotną, tj. konwersję listy do tablicy implementują metody `toArray(…)` zdefiniowane w interfejsie `List<E>`:

```
Object[] toArray()

<T> T[] toArray(T[] a)
```

Druga wersja tej metody umożliwia uzyskanie tablicy silnie typowanej. Co więcej, jeśli tablica przekazana jako argument wywołania jest dostatecznie duża by pomieścić elementy konwertowanej listy to zwracana jest ta właśnie tablica, tyle że wypełniona elementami z listy. W przeciwnym wypadku alokowany jest nowy obiekt tablicy.

Co prawda język Java nie zawiera mechanizmów, które pozwoliłyby na określenie w interfejsie konstruktorów które muszą być implementowane przez klasy chcące implementować ten interfejs, ale elementem kontraktu interfejsu `Collection<E>` jest wymaganie, aby wszystkie kolekcje implementowały oprócz konstruktora bezargumentowego także konstruktor jednoargumentowy akceptujący obiekt kolekcji, tak aby możliwa była konwersja pomiędzy dowolnymi dwoma typami kolekcji. Konstruktor ten tworzy nowy obiekt kolekcji zawierający wszystkie elementy z kolekcji przekazanej.

Operacja usuwania elementów kolekcji również została zdefiniowana w interfejsie `Collection<E>`, tyle że jako operacja opcjonalna. Z tej opcjonalności korzystają kolejki, które nie wspierają funkcjonalności usunięcia konkretnego elementu. Operacja usuwania w wersji ogólnej ma postać:

```
boolean remove(Object o)
```

zaś kolejki implementują wersję:

```
E remove()
```

Wersja ogólna implementowana przez listy i zbiory pozwala na usunięcie konkretnego elementu, zaś wersja bezargumentowa, implementowana przez kolejki zawsze usuwa element pierwszy. Listy implementują ponadto operację:

```
E remove(int index)
```

pozwalającą usunąć element znajdującej się na konkretnej – wskazanej indeksem – pozycji.

Kolejki są kolekcjami a więc implementują operacje zdefiniowane przez interfejs `Collection<E>` (bez operacji `remove(Object o)` która jest opcjonalna), jednak typowy zestaw funkcji wykorzystywanych do pracy z nimi to te zdefiniowane w interfejsie `Queue<E>` a więc:

```
boolean offer(E e)

E peek()
```

oraz:

```
E poll()
```

Metoda `offer(…)` to odpowiednik operacji `add(…)`. Metoda `peek()` służy do podejrzenia pierwszego elementu kolejki, tj. zwraca ten element ale nie usuwa go z kolejki. Metoda `pool()` także zwraca pierwszy element kolejki ale jest to pobranie elementu a nie podejrzenie go, tak więc element zostanie z kolejki usunięty (odpowiednik metody `remove()`). Z pośród interesujących nas klas interfejs `Queue<E>` implementują klasy: `PriorityQueue<E>` oraz `LinkedList<E>`. Ta druga implementuje dodatkowo interfejs `List<E>`. Klasa `LinkedList<E>` jest kolejką typu FIFO (ang. akr. First In First Out); operacja `offer(…)` dodaje elementy do końca listy zaś operacje `peek()` i `poll()` pobierają element z początku. Klasa `PriorityQueue<E>` to kolejka priorytetowa, która domyślnie stosuje naturalny porządek elementów, ale możliwe jest także stworzenie instancji używającej dowolnego innego porządku. Służy do tego konstruktor:

```
public PriorityQueue(int initialCapacity, Comparator<? super E> comparator)
```

Co ciekawe, wyższy priorytet mają elementy mniejsze, co jakby się nad tym chwilę zastanowić jest jednak rozsądne. W końcu liczba `1` jest mniejszą niż `2` a zgodne z intuicją jest, że `1` oznacza priorytet wyższy. Zerknijmy na poniższy przykład:

```
public static void main(String[] args) {
  Queue<String> queue = new PriorityQueue<String>();

  queue.offer("1");
  queue.offer("b");
  queue.offer("A");
  queue.offer("2");
  queue.offer("a");
  queue.offer(" ");

  while(!queue.isEmpty())
    System.out.println("-" + queue.poll() + "-");
}
```

Obiekt kolejki priorytetowej tworzymy z użyciem konstruktora bezargumentowego tak więc użyte będzie sortowanie zgodne z porządkiem naturalnym – w przypadku stringów porządkiem leksykograficznym. Do kolejki dodajemy kilka cyfr, liter małych oraz wielkich i spację, aby przy okazji przekonać się jaki jest porządek sortowania tych znaków. Uruchomienie metody spowoduje wyświetlenie tekstu:

```
- -
-1-
-2-
-A-
-a-
-b-
```

Widzimy więc, że spacje są w porządku leksykograficznym mniejsze niż cyfry (tj. mają najwyższy priorytet), po cyfrach następują wielkie litery a największe są litery małe.

Oprócz wspomnianej już klasy `LinkedList` interfejs `List` implementują klasy `ArrayList` i `Vector`. Klasa `LinkedList` jest klasyczną listą dwukierunkową elementów powiązanych ze sobą za pomocą referencji, dlatego też sprawdza się najlepiej w zastosowaniach które wymagają wydajnych operacji dodawania i usuwania elementów. Klasy `ArrayList` i `Vector` zaimplementowane są w oparciu o tablicę; sprawia to że bardzo wydajna jest operacja pobierania elementów z użyciem indeksu (w czasie stałym) oraz iteracja po elementach listy,

ale za to kosztowne są wszelkie modyfikacje, tj. dodawanie i usuwanie elementów. Zasadnicza różnica między tymi dwiema klasami jest taka, że metody klasy `ArrayList` nie są synchronizowane a więc w ogólności są szybsze niż metody klasy `Vector`; klasa `ArrayList` powinna być więc preferowana zawsze gdy synchronizacja nie jest wymagana, tj. gdy lista nie będzie współdzielona przez wiele wątków. Metody klasy `LinkedList` także nie są synchronizowane.

Podstawową implementację zbioru zapewnia klasa `HashSet`. Klasa `LinkedHashSet` rozszerza tę implementację poprzez wewnętrzne zorganizowanie elementów w listę, co pozwala zapewnić stały – zgodny z porządkiem dodawania elementów do zbioru – porządek iteracji. Klasa `TreeSet` implementuje interfejs `NavigableSet` i jest czystą analogią do mapy `TreeMap` implementującej interfejs `NavigableMap`. Podobnie jak `TreeMap` implementuje metody do tworzenia map-widoków tak klasa `TreeSet` implementuje metody do tworzenia zbiorów-widoków:

```
  // elementy ściśle mniejsze niż toElement
SortedSet<E> headSet(E toElement)

NavigableSet<E> headSet(E toElement, boolean inclusive)
  // elementy większe bądź równe fromElement
SortedSet<E> tailSet(E fromElement)

NavigableSet<E> tailSet(E fromElement, boolean inclusive)
```

oraz:

```
  // elementy większe bądź równe fromElement
  // i ściśle mniejsze niż toElement
SortedSet<E> subSet(E fromElement, E toElement)

NavigableSet<E> subSet(E fromElement,
                       boolean fromInclusive,
                       E toElement,
                       boolean toInclusive)
```

Dokładnie tak jak w przypadku map zbiory-widoki są powiązane ze zbiorami-matkami w ten sposób, że dodanie bądź usunięcie elementów z jednego z nich powoduje analogiczny efekt dla drugiego. Klasa `TreeSet` implementuje też analogiczne do metod z klasy `TreeMap` metody do wyszukiwania elementów mniejszych bądź większych niż element przekazany:

```
E higher(E e)

E ceiling(E e)

E lower(E e)

E floor(E e)
```

oraz do pobierania – z jednoczesnym usunięciem – elementu największego i najmniejszego:

```
  // pobiera ze zbioru element najmniejszy
E pollFirst()

  // pobiera ze zbioru element największy
E pollLast()
```

Metoda `descendingSet()` tworzy zbiór zawierający te same elementy co zbiór wyjściowy, ale z elementami posortowanymi w porządku odwrotnym. Zwracany zbiór jest zbiorem widokiem, tj. modyfikacje wykonane na jednym z tych zbiorów są odwzorowane odpowiednio w drugim.

SORTOWANIE LIST I TABLIC ORAZ WYSZUKIWANIE BINARNE

Przyjrzymy się teraz funkcjonalności sortowania list i tablic oraz wyszukiwania binarnego w tychże, a więc klasom `java.util.Arrays` i `java.util.Collections` (nie mylmy z interfejsem `Collection`) oraz interfejsom `java.util.Comparator<T>` i `java.lang.Comparable<T>`.

Klasy `Arrays` oraz `Collections` to klasy narzędziowe implementujące szereg operacji statycznych ułatwiających pracę z kolekcjami i tablicami. Różnych ciekawych metod jest tam kilka, jednak z perspektywy egzaminu na OCPJP interesują nas głównie operacje `sort(…)` i `binarySearch(…)` zaimplementowane w wielu wersjach w obydwu klasach.

Operacja `sort(…)` z klasy `Arrays` służy do sortowania tablic a analogiczna operacja zaimplementowana w klasie `Collections` do sortowania list. Podstawowa wersja operacji `sort(…)` z klasy `Arrays` akceptuje jeden argument

– tablicę wartości prostych lub tablicę obiektów. Mamy także wersje sortujące tylko pewien fragment tablicy; wtedy jako drugi argument wywołania podajemy indeks tablicy od którego (włącznie) należy zacząć sortowanie a jako trzeci argument indeks końca (nie włącznie) sortowanego fragmentu. Mamy więc do dyspozycji następujące funkcje:

```
  // analogiczne operacje istnieją
  // dla tablic long[], float[], Object[] itd.
public static void sort(int[] tab)

  // analogiczne operacje istnieją
  // dla tablic long[], float[], Object[] itd.
public static void sort(int[] tab, int fromIndex, int toIndex)
```

Operacje te sortują elementy tablicy w porządku rosnącym. Oczywiste jest jak wygląda ten porządek w przypadku liczb – znaki (tj. tablica `char[]`) sortowane są alfabetycznie – co jednak z sortowaniem tablicy obiektów? Żeby posortować elementy musimy wiedzieć, jak porównać je między sobą, tzn. musimy wiedzieć jak stwierdzić który z dwu elementów jest większy. Innymi słowy – obiekty w sortowanej tablicy muszą być porównywalne. W języku Java obiekty są porównywalne jeśli implementują interfejs `Comparable<T>`. Interfejs ten definiuje jedną metodę:

```
int compareTo(T obj)
```

Metoda ta zwraca liczbę ujemną jeśli obiekt dla którego ją wywołano jest mniejszy od obiektu przekazanego jako argument, liczbę dodatnią jeśli jest większy a liczbę 0 jeśli obiekty są równe.

Pyt.28 **Jaki interfejs implementuje klasa *TreeSet*? *Collection*, *Collections* czy *Map*?**

Interfejs `Comparable<T>` implementowany jest przez wiele klas standardu Java SE, w tym przez wszystkie klasy opakowujące typów prostych (np. `Integer`, `Double`, `Character`) a także klasy `Calendar`, `Date`, `Time` czy `String`.

A co jeśli chcemy posortować tablicę zgodnie z innym porządkiem niż rosnący? A może tablicę stringów chcemy posortować inaczej niż leksykograficznie (standardowy porządek dla stringów to porządek leksykograficzny)? Może ważniejsza jest dla nas w danym zastosowaniu długość stringa? A może

chcielibyśmy posortować ignorując wielkość liter? Da się? Da! Wystarczy zdefiniować interesujący nas porządek implementując interfejs `Comparator<T>`. Klasa `Arrays` oprócz dwu wersji operacji `sort(…)` pokazanych powyżej implementuje jeszcze dwie wersje różniące się tym, że jako kolejny argument wywołania dodajemy obiekt implementujący interfejs `Comparator<T>`, który to określa porządek sortowanych elementów. Poniżej przykład:

```
public class TestClass {
  public static void main(String[] args) {
    String[] strings = { "Urszula", "Ola", "Agata", "Jola" };

    Arrays.sort(strings, new LengthComparator());

    for(String s : strings)
      System.out.print(s + " ");
  }
}

  // określa sortowanie stringów od najkrótszego do najdłuższego
class LengthComparator implements Comparator<String> {
  public int compare(String strA, String strB) {
    return strA.length() - strB.length();
  }
}
```

Interfejs `Comparator<T>` wymaga implementacji metody `compare(…)`, która zwraca liczbę ujemną, jeśli pierwszy argument wywołania jest mniejszy niż drugi; liczbę 0, jeśli są równe; liczbę dodatnią jeśli pierwszy argument jest większy. Efektem uruchomienia powyższego programu będzie wyświetlenie tekstu:

```
Ola Jola Agata Urszula
```

Klasa `Collections` implementuje dwie wersje metody `sort(…)`. Pierwsza służy do sortowania listy elementów które są porównywalne same z siebie, tj. implementują interfejs `Comparable<T>` zaś druga do sortowania listy dowolnych obiektów, ale za to musimy określić porządek sortowania przekazując jako drugi

Odp.28 **Interfejs *Collection*. *Collections* to klasa narzędziowa udostępniająca metody do przetwarzania kolekcji. *Map* to interfejs który implementują klasy map a nie klasy kolekcji. *TreeSet* to implementacja zbioru a więc kolekcji.**

argument wywołania instancję komparatora, tj. instancję klasy implementującej interfejs `Comparator<T>`. Sygnatury tych metod wyglądają nieco odstraszająco i są następujące:

```
public static <T extends Comparable<? super T>> void sort(List<T> list)

public static <T> void sort(List<T> list, Comparator<? super T> comp)
```

Wyszukiwanie w kontekście list i tablic oznacza szukanie odpowiedzi na pytanie, czy dana lista lub tablica zawiera zadany element i jeśli tak to na której pozycji. Wyszukiwanie binarne (implementowane przez metody `binarySearch(…)` z klasy `Collections` oraz `Arrays`) to algorytm, który działa tylko dla kolekcji (list i tablic) posortowanych. Wyszukiwanie z zastosowaniem tego algorytmu działa o rząd wielkości szybciej niż wyszukiwanie liniowe (tj. przeglądanie listy element po elemencie) ale musimy pamiętać, że można go użyć tylko i wyłącznie do wyszukiwania w listach lub tablicach posortowanych. Co więcej, użycie algorytmu wyszukiwania binarnego dla listy czy tablicy nieposortowanej nie spowoduje błędu kompilacji czy wykonania. Metoda `binarySearch(…)` wykona się, tyle że otrzymamy wynik nie zgodny ze stanem faktycznym.

W klasie `Collections` zaimplementowano dwie wersje metody `binarySearch(…)`. Analogicznie jak w przypadku sortowania, pierwsza z nich służy do wyszukiwania z listy elementów, które są porównywalne a więc implementują interfejs `Comparable<T>`. Druga wersja służy do wyszukiwania z listy elementów, które same z siebie nie są porównywalne a więc musimy przekazać także komparator – obiekt, który będzie używany do porównywania – a więc obiekt implementujący interfejs `Comparator<T>`. Sygnatury tych metod są następujące:

```
public static <T> int binarySearch(
                 List<? extends Comparable<? super T>> list, T key)

public static <T> int binarySearch(
                 List<? extends T> list, T key, Comparator<? super T> comp)
```

Drugiej wersji metody – tej w której przekazujemy także komparator – należy użyć także w przypadku, gdy lista posortowana jest zgodnie z porządkiem innym niż naturalny (naturalny to ten określony przez metodę `compareTo(…)`

zdefiniowaną przez interfejs `Comparable<T>`). W szczególności, jeśli listę sortowaliśmy z użyciem konkretnego komparatora to do wyszukiwania binarnego musimy użyć tego samego komparatora.

W klasie `Collections` zaimplementowano także metody `reverseOrder(…)`, które konstruują obiekt komparatora definiującego sortowanie odwrotne. Bezargumentowa wersja metody `reverseOrder()` zwraca komparator definiujący porządek odwrotny do porządku naturalnego a wersja jednoargumentowa komparator odwrotny do komparatora przekazanego jako parametr wywołania. Sygnatury tych metod są następujące:

```
public static <T> Comparator<T> reverseOrder()

public static <T> Comparator<T> reverseOrder(Comparator<T> cmp)
```

Możemy także odwrócić kolejność elementów listy, niezależnie od tego czy jest ona posortowana czy nie używając operacji:

```
public static void reverse(List<?> list)
```

W klasie `Arrays` zaimplementowano także metody wyszukiwania – po jednej metodzie dwuargumentowej dla tablicy każdego typu prostego i dla tablicy obiektów. Pierwszym argumentem jest tablica a drugim szukany element. Zaimplementowano także metodę, która akceptuje komparator. Sygnatury tych metod są następujące:

```
  // analogiczne operacje istnieją
  // dla tablic long[], float[], Object[] itd.
public static int binarySearch(int[] a, int key)

public static <T> int binarySearch(T[] a,
                  T key,
                  Comparator<? super T> comp)
```

TYPY GENERYCZNE

Język Java jest językiem ściśle typowanym; do referencji o danym typie możemy przypisać jedynie obiekty tego typu bądź typów pochodnych a jako

parametry wywołania metody możemy przekazać tylko obiekty zgodne co do typu z argumentami tej metody. Jest jednak od zasady ścisłego typowania jeden wyjątek – kolekcje. Aż do Javy w wersji 1.4 włącznie nie było możliwości określenia typu elementów kolekcji czy typów kluczy bądź wartości map. Operacja sumowania listy liczb w Javie 1.4 i wcześniejszych mogłaby wyglądać jakoś tak:

```
  // lista może zawierać dowolne obiekty, nie tylko liczby
public static double sum(List listOfNums) {
  double sum = 0;

  for(Object obj : listOfNums) {
    Number num = (Number)obj; // to rzutowanie może zakończyć się wyjątkiem

    sum += num.doubleValue();
  }

  return sum;
}
```

Jako parametr wywołania powyższej metody możemy przekazać listę obiektów typu `Number` i wówczas sumowanie zakończy się pomyślnie, ale równie dobrze możemy użyć listy obiektów dowolnych innych typów; kompilator nie będzie protestował ale wystąpi błąd wykonania – rzutowanie niekompatybilnego elementu kolekcji na typ `Number` zakończy się wyjątkiem `ClassCastException`. Szczęśliwie, mamy obecnie do dyspozycji – począwszy od Javy 5 – mechanizm typów generycznych; dzięki temu mechanizmowi możemy określić typ elementów kolekcji. Kolekcje przestają być li tylko grupami obiektów – są one teraz grupami obiektów ściśle określonych typów. Ścisłe typowanie kolekcji zawitało pod nasze strzechy i dzięki temu możemy teraz zaproponować bezpieczną implementację metody sumowania:

```
  // mamy pewność, że lista będzie zawierała tylko obiekty typu Number
public static double sum(List<Number> listOfNums) {
  double sum = 0;

  for(Number num: listOfNums)
    sum += num.doubleValue();

  return sum;
}
```

Argument wywołania metody jest typu `List<Number>` dzięki czemu mamy pewność, że nikt nie wywoła jej z listą obiektów typu innego niż `Number`. Nie musimy też wykonywać rzutowania. Przypomnijmy sobie jak wygląda operacja zwracająca element listy `List<E>` znajdujący się na wskazanej jej pozycji – jest to:

```
E get(int index)
```

Jak widzimy typem zwracanym przez metodę `get(…)` nie jest `Object`, tylko `E`. `E` to tak zwany typ parametryczny. Jest to ten sam typ którym sparametryzowaliśmy listę deklarując referencję czy tworząc jej instancję. Jeśli więc zadeklarujemy referencję typu `List<String>` lub stworzymy obiekt typu `LinkedList<String>` to typem zwracanym przez metodę `get(…)` tej listy będzie `String`.

W powyższych przykładach klasy parametryzowane były jednym typem, ale w ogólności liczba typów parametrycznych może być dowolnie duża. W szczególności, mapy parametryzowane są przecież dwoma typami. Pierwszy typ parametryczny to typ klucza zaś drugi to typ wartości skojarzeń reprezentowanych przez tę mapę.

Język Java udostępnia obecnie mechanizm typów generycznych (wszystkie klasy kolekcji pakietu Java SE są obecnie zaimplementowane jako klasy generyczne) ale jednocześnie umożliwia używanie klas i interfejsów parametryzowanych typem bez określania tego typu. Przykład podany na początku tego podrozdziału używa interfejsu `List` bez parametryzacji typem i jak najbardziej kod ten kompiluje się i działa także na platformie Java 5 i 6. Powodem dla którego pozostawiono taką możliwość była potrzeba zachowania wstecznej zgodności języka.

Wyobraźmy sobie, że chcemy użyć zewnętrznej biblioteki zaimplementowanej jeszcze w czasach z przed Javy 5 i wywołać jedną z jej operacji której argumentem jest kolekcja. Mamy więc sytuację, w której kolekcja sparametryzowana typem, bo pochodząca z nowo implementowanego kodu, musi być przekazana jako parametr wywołania starej metody, której argumentem jest kolekcja nietypowana. Zerknijmy na taki przykład:

```
  // nowy kod
public class NewApp {
  public static void main(String[] args) {
    Map<Integer, String> newMap = new HashMap<Integer, String>();

    OldClass oldClass = new OldClass();

    Map<Integer, String> processedMap = oldClass.oldOp(newMap);
  }
}

  // stary kod
class OldClass {
  public Map oldOp(Map preGenericMap) {
    // implementacja metody

    return preGenericMap;
  }
}
```

W metodzie `main(…)` deklarujemy referencję oraz tworzymy obiekt mapy sparametryzowanej typami `Integer` i `String` (to pierwsze to typ klucza a drugie typ wartości). Następnie tworzymy instancję klasy `OldClass` która obrazuje u nas starą klasę biblioteczną i wywołujemy jej metodę `oldOp(…)` przekazując w miejsce niesparametryzowanego argumentu typu `Map` mapę `Map<Integer, String>`. Kod kompiluje się i uruchamia w pełni poprawnie. Zastanówmy się natomiast nad jedną związaną z tym kwestią – skoro w metodzie `oldOp(…)` z klasy `OldClass` mapa nie jest sparametryzowana typami, to czy jest jakiś mechanizm który powstrzymuje nas przed złamaniem w tej metodzie kontraktu naszej mapy polegającego na tym, że jej klucze są zawsze typu `Integer` a wartości typu `String`? Przecież kod:

```
class OldClass {
  public Map oldOp(Map preGenericMap) {
    preGenericMap.put(1, new Date()); // wartość typu Date a nie String

    return preGenericMap;
  }
}
```

jest w pełni poprawny; tak zaimplementowana klasa `OldClass` kompiluje się i co więcej, wywołanie jej z parametrem typu `Map<Integer, String>` także jest poprawne. Taki kod skompiluje się i wykona poprawnie, problemy pojawią się

natomiast gdy zechcemy feralne skojarzenie z mapy pobrać, np. tak jak w poniższym przykładzie:

```
public class NewApp {
  public static void main(String[] args) {
    Map<Integer, String> newMap = new HashMap<Integer, String>();

      // klasa OldClass zaimplementowana jak w poprzednim przykładzie
    OldClass oldClass = new OldClass();

    Map<Integer, String> processedMap = oldClass.oldOp(newMap);

      // tu dostaniemy wyjątek ClassCastException
    String value = processedMap.get(1);
  }
}
```

Spodziewamy się, że mapa z powyższego przykładu ma wartości o typie `String` – i co więcej, kompilator spodziewa się tego samego, więc nie zgłosi błędu – a tymczasem metoda `get(…)` zwraca obiekt typu `Date` dodany w metodzie `oldOp(…)` i program kończy się wyjątkiem:

```
java.lang.ClassCastException: java.util.Date cannot be cast to
java.lang.String
      at my.pckg.NewApp.main(NewApp.java:17)
```

Przekazując kolekcje (czy mapy) typowane jako parametry wywołania metod używających kolekcji w starym stylu musimy zachować szczególną ostrożność – może to doprowadzić do złamania kontraktu dotyczącego typu elementów takiej kolekcji. Kompilator co prawda nie może wiedzieć, czy opisywana sytuacja faktycznie będzie miała miejsce więc nie zgłasza błędu, ale widząc że używamy kolekcji w sposób niebezpieczny ostrzega nas na każdym kroku. Kompilując dyskutowany powyżej kod otrzymamy dwa ostrzeżenia:

```
public class NewApp {
  public static void main(String[] args) {
    Map<Integer, String> newMap = new HashMap<Integer, String>();

    OldClass oldClass = new OldClass();

      // ostrzeżenie o niejawnym rzutowaniu Map na Map<Integer, String>
    Map<Integer, String> processedMap = oldClass.oldOp(newMap);
  }
}
```

```
class OldClass {
  public Map oldOp(Map preGenericMap) {

      // ostrzeżenie o niebezpieczeństwie związanym z użyciem metody put(…)
    preGenericMap.put(1, new Date());

    return preGenericMap;
  }
}
```

Przypisanie instancji typu `Map` zwracanej przez metodę `oldOp(…)` do zmiennej `processedMap` typu `Map<Integer, String>` nie wymaga jawnego rzutowania, ale dostajemy ostrzeżenie, że rzutowanie takie jest wykonywane implicite i że jest ono potencjalnie niebezpieczne. Ostrzegani jesteśmy również gdy używamy metody `put(…)` dla referencji typu `Map`, a więc niesparametryzowanej typami, co może spowodować prawdziwą katastrofę – dodanie do mapy skojarzenia o niedopuszczalnych typach. Co ważne, kompilator został odpowiednio poinstruowany i wie, że operacja `put(…)` dodaje elementy więc jest niebezpieczna a że z kolei metody takie jak `get(…)`, `containsKey(…)` czy nawet `remove(…)` nie mogą nic zepsuć i w związku z tym generuje ostrzeżenia tylko gdy używamy metody `put(…)`. My również musimy to wszystko wiedzieć, bo możemy być o to zapytani na egzaminie. Musimy wiedzieć kiedy próba kompilacji zakończy się błędem, kiedy kompilacja się powiedzie, ale będą wygenerowane ostrzeżenia oraz kiedy kod skompiluje się bez zastrzeżeń. Musimy umieć rozróżnić te trzy przypadki oraz dodatkowo, musimy umieć rozróżniać błędy wykonania od błędów kompilacji.

TYPY GENERYCZNE A POLIMORFIZM

Programowanie z użyciem interfejsów jest jednym z podstawowych wyznaczników dobrego stylu programowania obiektowego i właśnie tak też przeważnie używamy kolekcji – poprzez ich w miarę ogólne interfejsy. Jeśli więc potrzebujemy listy to na typ referencji wybieramy interfejs `List<E>` a nie np. `LinkedList<E>`; niemalże odruchowo piszemy:

```
List<Number> list = new LinkedList<Number>();
```

Klasa `LinkedList` implementuje interfejs `List` więc wszystko jest w jak największym porządku – dzięki polimorfizmowi języka Java. Czy jednak analogicznie możemy postąpić w stosunku do typu parametryzującego? Czy poprawna jest instrukcja:

```
  // kod ten nie skompiluje się
List<Number> list = new LinkedList<Integer>();
```

Klasa `Integer` jest podklasą klasy `Number` więc mogłoby się wydawać, że powyższy kod jest poprawny, ale nie jest, tzn. nie skompiluje się. Zanim jednak dojdziemy do ostatecznej konkluzji zastanówmy się, co by to oznaczało, gdyby poprawny był. Oto więc mamy referencję `list` typu `List<Number>`, zatem kompilator nie powstrzyma nas od dodania do tej listy elementu typu np. `Long` (klasa `Long` dziedziczy z klasy `Number`). Kod skompilowałby się, ale jego uruchomienie niechybnie skończyłoby się wyjątkiem, jako że obiektu typu `Long` nie da się przecież dodać do kolekcji typu `LinkedList<Integer>`.

Sformułujmy teraz ostateczny wniosek – typy parametryzujące typ referencji muszą być dokładnie takie same jak typy parametryzujące typ obiektu który do tej referencji chcemy przypisać! Zauważmy jeszcze, że w przypadku tablic sytuacja jest zgoła odwrotna. Zerknijmy na poniższy przykład:

```
public void testArrays() {
  Number[] array = new Integer[4]; // to przypisanie jest poprawne

  array[0] = new Long(8); // wyjątek java.lang.ArrayStoreException
}
```

Sytuacja jest odwrotna, ponieważ nie mamy tu analogicznego mechanizmu wymuszania zgodności typu referencji z typem obiektu tablicy w czasie kompilacji, za to mamy kontrolę zgodności typów elementów dodawanych do tablicy z typem obiektu tablicy w czasie wykonania. Próba dodania do tablicy elementu nie zgodnego z typem tej tablicy powoduje rzucenie wyjątku `ArrayStoreException`.

Obiektu typu `LinkedList<Integer>` nie można przypisać do referencji o typie `List<Number>` a więc nie można także przekazać tego obiektu do metody akceptującej argument typu `List<Number>`. A zatem czy da się napisać metodę

która przetwarza w jakiś sposób wszystkie listy liczb, niezależnie od tego czy są to liczby typu `Long`, `Integer` czy może `Float`? O tym za chwilę, a póki co zobaczmy jak sprawa się ma z tablicami. Taka metoda dla tablic – powiedzmy metoda sumująca wszystkie liczby z tablicy – mogłaby wyglądać następująco:

```
public static Number sumArray(Number[] numsArray) {
  double sum = 0;

  for(Number num : numsArray)
    sum += num.doubleValue();

  return sum;
}
```

Metoda `sumArray(…)` deklaruje argument typu `Number[]`. Jak wiemy do referencji typu `Number[]` można przypisać obiekty typu `Long[]` czy `Integer[]` oraz tablice wszystkich innych typów które są podklasami klasy `Number`. Analogicznie, możemy wywołać metodę `sumArray(…)` z parametrem będącym tablicą dowolnego z tych typów. Poprawne więc będzie wywołanie postaci:

```
sumArray(new Long[] {1L, 2L, 3L});
```

czy

```
sumArray(new Float[] {1F, 2F, 3F});
```

Szczęśliwie da się analogiczny mechanizm zaimplementować dla kolekcji, tyle że musimy posłużyć się nieco innym określeniem typu parametrycznego. Jeśli zadeklarowaliśmy, że argumentem wywołania funkcji jest lista obiektów typu `Number` to funkcję tę możemy wywołać tylko i wyłącznie przekazując jako parametr taką właśnie listę, jednak możemy zadeklarować, że argumentem jest lista obiektów typu `Number` lub dowolnego jego podtypu. Analogiczna do metody `sumArray(…)` metoda sumująca dla list mogłaby wyglądać następująco:

Pyt.29 **Jeśli *AA* jest podklasą klasy *A*, to czy poprawnym będzie przypisanie obiektu typu *LinkedList<AA>* do referencji typu *List<A>*?**

```
public Number sumList(List<? extends Number> numsList) {
  double sum = 0;

  for(Number num : numsList)
    sum += num.doubleValue();

  return sum;
}
```

Różnica w stosunku do deklaracji jakie widzieliśmy uprzednio to sposób określenia typu parametrycznego. Deklaracja postaci `List<Number>` oznacza listę obiektów typu `Number` i żadną inną, natomiast `List<? extends Number>` to lista obiektów typu `Number` lub dowolnego innego podtypu. Konstrukcja ta może być używana także w stosunku do interfejsów, np. aby zadeklarować listę obiektów wszelkich typów implementujących bądź rozszerzających interfejs `Runnable` napiszemy `List<? extends Runnable>` (używamy słówka kluczowego `extends` a nie `implements` nawet w stosunku do interfejsów). Ważne jest abyśmy pamiętali przy tym, że typ kolekcji i typy obiektów faktycznie należących do tej kolekcji to dwie różne rzeczy. Przykładowo, jeśli zadeklarujemy zmienną referencyjną typu `List<Number>` to możemy do niej przypisać obiekt listy typu `LinkedList<Number>` ale typu `LinkedList<Integer>` czy `LinkedList<Float>` już nie; i to jest jedna rzecz. Osobną rzeczą jest natomiast to, że do kolekcji sparametryzowanej pewnym typem możemy dodawać także obiekty dowolnych podtypów tego typu. Jeśli więc mamy listę:

Odp.29 **Nie! Typ parametryzujący typ referencji musi być tym samym typem co typ parametryzujący typ obiektu, który do tej referencji chcemy przypisać. Nie ma znaczenia czy *AA* dziedziczy z *A* czy nie. Aby przypisanie było poprawne obydwa typy muszą być parametryzowane dokładnie tym samym typem.**

```
List<Number> myList = new LinkedList<Number>();
```

to możemy dodawać do niej obiekty zarówno typu `Integer`, `Long` jak i `Float` oraz dowolnych innych podklas klasy `Number`. Poprawne są więc instrukcje:

```
myList.add(new Long(1));
myList.add(new Double(1));
myList.add(new BigInteger("12312312312312123123123"));
```

Używanie w taki sposób typów nieokreślonych (ang. wildcard types) do parametryzacji typu kolekcji ma jednak jedno istotne ograniczenie – otóż do kolekcji takiej nie można dodawać elementów. Dlaczego? Załóżmy, że mamy metodę która deklaruje argument typu `List<? extends Number>`. Możemy ją wywołać z parametrem typu `LinkedList<Integer>` czy `LinkedList<Long>` a także z dowolną inną listą sparametryzowaną podklasą klasy `Number`. W jaki sposób kompilator miałby weryfikować czy dodawany obiekt jest co do typu zgodny z faktycznym typem elementów przekazanej kolekcji? Załóżmy, że w metodzie takiej dodajemy do kolekcji widocznej poprzez referencję typu `List<? extends Number>` instancję klasy `Long`. Wówczas wywołanie tej metody z parametrem typu `LinkedList<Integer>` zakończyłoby się wyjątkiem – nie można przecież dodać obiektu klasy `Long` do listy `LinkedList<Integer>`.

Powyżej używaliśmy typów nieokreślonych aby umożliwić polimorficzne traktowanie typu parametryzującego kolekcje, dzięki czemu mogliśmy zaimplementować generyczną metodę sumującą liczby z listy niezależnie od tego czy była to lista liczb typu `Float`, `Long`, `Integer` czy dowolnego innego typu. Jak wiemy mechanizm ten ma jednak poważne ograniczenie – do takiej kolekcji nie możemy dodawać nowych elementów. Co jednak począć w przypadku gdy chcemy zaimplementować generyczną metodę która dodaje elementy do kolekcji? Jest to możliwe i to przy użyciu tego samego mechanizmu typów nieokreślonych (ang. wildcard types), tyle, że użytego nieco inaczej; zamiast definiować listę podtypów możemy zdefiniować listę nadtypów. Aby zadeklarować listę obiektów typu `Number` albo dowolnych jego podtypów pisaliśmy `List<? extends Number>`, aby zadeklarować listę obiektów typu `Number` albo dowolnego jego nadtypu napiszemy `List<? super Number>`. Przykład – metoda kopiująca liczby z jednej listy do drugiej z jednoczesnym podwojeniem wartości – poniżej:

```
public void copy(List<? super Double> out, List<? extends Number> in) {
  for(Number num : in)
    out.add(num.doubleValue() * 2);
}
```

Możemy taką metodę wywołać przekazując jako pierwszy parametr listę typu `List<Double>`, `List<Number>` lub `List<Object>` a jako drugi listę `List<Number>`, `List<Integer>`, `List<Float>` lub dowolną inną listę parametryzowaną podtypem typu `Number`.

Istnieje jeszcze jeden wariant składniowy dla parametryzacji typem nieokreślonym. Mianowicie, zamiast pisać `List<? extends Object>` możemy napisać `List<?>`. Jest to dokładnie to samo.

DEKLARACJE GENERYCZNE

Mechanizm parametryzacji typem nie jest zastrzeżony dla kolekcji; kolekcje są w końcu klasami takimi jak wszystkie inne klasy i tak samo jak kolekcje tak samo inne klasy mogą być parametryzowane. Taką klasę możemy bez dużego trudu napisać także sami. Oto przykład:

```
abstract class ListProcessor<T> {
  public List<T> process(List<T> listOfT) {
    List<T> newList = new LinkedList<T>();

    for(T t: listOfT)
      newList.add(doProcess(t));

    return newList;
  }

  protected abstract T doProcess(T t);
}
```

Klasa `ListProcessor` jest parametryzowana typem `T`. Metoda `process(…)` tej klasy akceptuje parametr i zwraca rezultat typu `List<T>`, gdzie `T` jest tym samym typem którym sparametryzujemy klasę `ListProcessor`. W metodzie tej przetwarzamy otrzymaną listę, wykonując na każdym z jej elementów operację `doProcess(T t)` i zwracamy listę zawierającą tak przetworzone elementy. Użyjmy teraz tej klasy do zaimplementowania metody która podwoi wartość wszystkich elementów listy zawierającej elementy typu `Integer`:

```
public void double() {
  List<Integer> list = new LinkedList<Integer>();

  list.add(1);
  list.add(2);

  ListProcessor<Integer> doubleProcessor = new ListProcessor<Integer>() {
    protected Integer doProcess(Integer t) {
      return t * 2;  // klasa anonimowa rozszerzająca typ ListProcessor<T>
    }                // parametryzowany typem Integer
  };

  List<Integer> newList = doubleProcessor.process(list);
}
```

Typ parametryczny w powyższych przykładach oznaczany jest identyfikatorem T (od angielskiego słowa „type"). W poprzednich podrozdziałach widzieliśmy, że w stosunku do typów parametrycznych kolekcji używana była litera E (od angielskiego słowa „element"). Z kolei typ kluczy map oznacza się literą K (od słowa „key") a typ wartości literą V (od słowa „value"). Tego typu mniej lub bardziej oficjalnych konwencji zapewne jest więcej, jednak w ogólności nie ma co do identyfikatorów typów parametrycznych żadnych ograniczeń, poza tym jednym, że musi to być poprawny identyfikator języka Java.

Klasę `ListProcessor` z powyższego przykładu możemy sparametryzować dowolnym typem, ale może się zdarzyć, że typ parametryzujący zechcemy ograniczyć. Przykładowo, zaimplementujmy klasę z metodą znajdującą maksimum z listy liczb. Klasa ta może być parametryzowana tylko typem liczbowym – typem liczb które będziemy sumować – a więc tylko podtypem typu `Number`. Mogłaby ona wyglądać jakoś tak:

```
  // możliwa jest parametryzacja tylko typem Number lub jego podtypem
class GetMaxProcessor<T extends Number> {

  public T process(List<T> numsList) {
    T max = numsList.get(0); // załóżmy że lista jest niepusta

    for(T num: numsList)
      if(num.doubleValue() > max.doubleValue())
        max = num;

    return max; // wartość zwracana jest tego samego typu co liczby z listy
  }
}
```

Użyjmy teraz tej klasy do znalezienia maksimum na liście liczb typu `Float`. Zauważmy, że funkcja `process(…)` z klasy `GetMaxProcessor` sparametryzowanej typem `Float` zwraca rezultat typu `Float`:

```
public class TestClass {
  public static void main(String[] args) {
    List<Float> floatsList = Arrays.asList(1F, 12F, 3F, 14F, 5F);
    GetMaxProcessor<Float> maxProcessor = new GetMaxProcessor<Float>();

      // funkcja process(...) zwraca rezultat typu Float
    Float max = maxProcessor.process(floatsList);
  }
}
```

Generyczna może być nie tylko deklaracja klasy, ale także samej metody, tj. w obrębie nie generycznej klasy możemy zdefiniować generyczne operacje. Możemy w ten sposób zaimplementować funkcjonalność pokazanej powyżej klasy `GetMaxProcessor`:

```
class GetMaxProcessor { // klasa nie jest już generyczna

    // za to generyczna jest metoda
  public <T extends Number> T process(List<T> numsList) {
    T max = numsList.get(0); // załóżmy że lista jest niepusta

    for(T num: numsList)
      if(num.doubleValue() > max.doubleValue())
        max = num;

    return max;
  }
}
```

Zauważmy, że deklaracja typu parametrycznego `T` zniknęła z deklaracji klasy; musiała się w związku z tym pojawić gdzie indziej… i pojawiła się w deklaracji metody, tuż przed określeniem zwracanego typu. W powyższym przykładzie potrzebowaliśmy tej deklaracji choćby po to, aby określić, że typ `T` może być tylko typem `Number` lub podtypem typu `Number`, ale to nie jedyny powód. Deklaracja ta jest konieczna chociażby po to, żeby poinformować kompilator, że identyfikator `T` to parametr typu, i że nie musi poszukiwać klasy o nazwie `T`. Gdybyśmy spróbowali skompilować metodę:

```
public void someOp(T t) {
  // implementacja metody
}
```

kompilator myślałby, że `T` to nazwa klasy i otrzymalibyśmy komunikat błędu mówiący o tym, że taki typ nie istnieje. Musimy określić, że `T` to parametr typu, a nie konkretny typ. Poprawna deklaracja tej metody to:

```
public <T> void someOp(T t) {
  // implementacja metody
}
```

WĄTKI

Niniejszy rozdział pokrywa prezentowanym zakresem wiedzy następujące wymagania egzaminu 1Z0-851:

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

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

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

TWORZENIE I URURCHAMIANIE WĄTKÓW

Wątek to nic innego jak pewien kod, który wykonywany jest przez Wirtualną Maszynę Javy niezależnie od kodu innych wątków, przy czym owo niezależnie oznacza współbieżnie. Aby wykonać pewien kod w osobnym wątku wykonania należy przede wszystkim określić jaki kod chcemy wykonać; należy więc nowy wątek zdefiniować. Definicja wątku w najprostszej postaci to implementacja podklasy klasy `java.lang.Thread`. Alternatywnie, wątek można zdefiniować implementując interfejs `java.lang.Runnable`.

Implementacja programu w języku Java jest równoznaczna z implementacją metody `main(…)`. Uruchomienie programu napisanego w Javie to de facto uruchomienie metody `main(…)`. Zakończenie się metody `main(…)` jest równoznaczne z zakończeniem się programu. Metoda `main(…)` to nic innego jak implementacja głównego wątku wykonania naszej aplikacji – wątek ten uruchamiamy uruchamiając aplikację. Każdy inny wątek implementujemy tworząc podklasę klasy `Thread` albo implementując interfejs `Runnable`, co w obu przypadkach sprowadza się do implementacji metody `run()`. Definicja wątku to jednak nic więcej jak tylko pewna klasa zawierająca fragment kodu. Aby utworzyć nowy wątek i uruchomić w jego ramach kod zdefiniowany we wspomnianej metodzie `run()` należy wywołać metodę `start()` klasy `Thread` – dopiero wówczas tworzony i równocześnie uruchamiany jest wątek. Zerknijmy na poniższy przykład:

```
public class TestClass {
  public static void main(String[] args) {
    System.out.println("Kod wykonany w wątku głównym");

    Thread newThread = new MyThread();
      // dopiero wywołując metodę start() tworzymy nowy wątek
    newThread.start();
  }
}

class MyThread extends Thread {  // ta klasa to tylko definicja wątku
  public void run() {
    System.out.println("Kod wykonany w nowym wątku");
  }
}
```

Metoda `start()` z klasy `Thread` pokazana w powyższym przykładzie tworzy nowy wątek i uruchamia w jego ramach kod zdefiniowany w metodzie `run()` z klasy `MyThread`. Definiowanie wątków poprzez implementację podklas klasy `Thread` jest najprostszą możliwą metodą, jednak nie zawsze najwygodniejszą. Alternatywnie, wątek możemy zdefiniować implementując interfejs `java.lang.Runnable` i na podstawie takiej klasy tworząc instancje klasy `Thread`. Klasa `Thread` udostępnia konstruktor jednoargumentowy akceptujący instancję klasy implementującej `Runnable`. Metoda `run()` w klasie `Thread` zaimplementowana jest w ten sposób, że uruchamia metodę `run()` z przekazanej jako argument konstruktora klasy implementującej interfejs `Runnable`. Wszystko powinno stać się jasne po przestudiowaniu kolejnego przykładu:

```
public class TestClass {
  public static void main(String[] args) {
    System.out.println("Kod wykonany w wątku głównym");

      // wątek tworzymy na bazie instancji klasy implementującej Runnable
    Thread newThread = new Thread(new MyRunnable());

    newThread.start();
  }
}

class MyRunnable implements Runnable {
  public void run() {
    System.out.println("Kod wykonany w nowym wątku");
  }
}
```

Oprócz metody `start()` klasa `Thread` udostępnia jeszcze szereg innych, mniej lub bardziej kluczowych metod. Jedną z nich jest metoda `getName()`, która zwraca nazwę wątku. Wątek główny domyślnie nazywa się „main”. Wątki tworzone przez programistę mają nazwy wygenerowane automatycznie przez Wirtualną Maszynę Javy, ale nazwa ta może być zmieniona. W tym celu można użyć metody `setName(…)` albo konstruktora, który jako argument wywołania (drugi jeśli przekazujemy instancję `Runnable` lub pierwszy i jedyny, jeśli nie) akceptuje nazwę nowo tworzonego wątku. Nazwy te nie muszą być unikalne – różne wątki mogą mieć tę samą nazwę. Unikalny natomiast jest identyfikator wątku, który możemy odczytać z użyciem metody `getId()` i którego nie możemy zmienić. Przydatna jest także statyczna metoda `currentThread()`,

która zwraca instancję klasy `Thread` reprezentującą aktualnie wykonywany wątek.

Ważną właściwością wątku jest jego stan. Aktualny stan wątku możemy sprawdzić posługując się metodą `getState()`, która zwraca jedną z wartości wyliczeniowych typu `java.lang.Thread.State`. Nowo utworzonej instancji wątku, który jeszcze nie został uruchomiony odpowiada stan `NEW`. Gdy wątek jest startowany, jego stan automatycznie zmienia się na `RUNNABLE`, by w końcu – potencjalnie przechodząc przez różne stany przejściowe – zakończyć się w stanie `TERMINATED`. Bardzo ważne jest by zapamiętać – zwłaszcza z perspektywy egzaminu na OCPJP – że wystartować można tylko i wyłącznie wątek, który znajduje się w stanie `NEW`. Oznacza to, że każdy wątek może być uruchomiony tylko raz! Każde następne – tj. nie pierwsze – wywołanie metody `start()` na instancji reprezentującej wątek spowoduje rzucenie wyjątku `IllegalThreadStateException`. Zerknijmy na poniższy przykład:

```
public class TestClass {
  public static void main(String[] args) {
    Thread newThread = new Thread(new MyRunnable(), "newThread");

    System.out.println(
      "Wątek " + newThread.getName() +
      " w stanie " + newThread.getState());

    newThread.start();
  }
}

class MyRunnable implements Runnable {
  public void run() {
    Thread thread = Thread.currentThread();

    System.out.println(
      "Wątek " + thread.getName() + " w stanie " + thread.getState());
  }
}
```

Zauważmy, że w metodzie `run()` implementowanej w klasie `MyRunnable` nie mamy bezpośredniego dostępu do instancji klasy `Thread`, bo też póki co kod ten nie jest powiązany z żadną instancją reprezentującą wątek; musimy się więc posłużyć metodą statyczną `currentThread()`. Uruchomienie tego programu spowoduje wyświetlenie tekstu:

```
Wątek newThread w stanie NEW
Wątek newThread w stanie RUNNABLE
```

Wiemy już, że wątek możemy uruchomić tylko i wyłącznie raz (także wątek który się zakończył nie może być uruchomiony ponownie), ale czy możemy utworzyć i uruchomić wiele wątków wykonujących jednocześnie (współbieżnie) ten sam kod? Oczywiście możemy, przykład poniżej:

```
public class TestClass {
  public static void main(String[] args) {
    Runnable runnable = new MyRunnable();

      // wszystkie wątki będą wykonywały dokładnie ten sam kod
    Thread threadA = new Thread(runnable);
    Thread threadB = new Thread(runnable);
    Thread threadC = new Thread(runnable);
    Thread threadD = new Thread(runnable);

    threadA.start();
    threadB.start();
    threadC.start();
    threadD.start();
  }
}

class MyRunnable implements Runnable {
  private int val = 0;

  public void run() {
    for (int i = 0; i < 100; i++)
      System.out.println(
        "id == " + Thread.currentThread().getId() +
        ", val == " + getVal());
  }

  private synchronized int getVal() {
    return ++val;
  }
}
```

Powyższy program tworzy i uruchamia cztery nowe wątki z których każdy, współbieżnie z pozostałymi, wykonuje metodę `run()` dokładnie tej samej instancji klasy `MyRunnable`, a więc wyświetla i inkrementuje wartość tej samej zmiennej `val`. Poniżej fragment tego, co wypisał ten program po uruchomieniu w trakcie jednego z testów:

```
id == 10, val == 224
id == 9, val == 150
id == 10, val == 226
id == 11, val == 225
id == 10, val == 228
```

Jak widać wątki przeplatają się między sobą (choć często występują też długie fragmenty, gdy nieprzerwanie wykonywał się jeden z wątków). Co ciekawe, w drugiej linii pojawia się wartość 150, mimo że w linii pierwszej pojawiła się wartość 224. Dowodzi to, że wątek o identyfikatorze 9 wykonał operację `getVal()` w momencie gdy wartość zmiennej `val` wynosiła 149 po czym przestał się wykonywać przed wypisaniem komunikatu „id == 9, val == 150" i wznowił swe wykonanie, tj. wypisał ten komunikat, dopiero po tym jak inne wątki zwiększyły wartość tej zmiennej grubo ponad 200.

SYNCHRONIZACJA WĄTKÓW

Wątki komunikują się ze sobą w głównej mierze poprzez współdzielone obiekty. Operacje wykonywane na obiekcie współdzielonym muszą być odpowiednio zsynchronizowane, tak aby w skutek współbieżnego ich wykonania nie doszło do wystąpienia sytuacji niepożądanych. Język Java udostępnia mechanizm synchronizacji wątków oparty o monitory skojarzone z obiektami. Każdy obiekt, niezależnie od typu, posiada skojarzony z nim monitor który może być użyty do synchronizacji. Synchronizacja polega na wskazaniu pewnych fragmentów kodu oraz obiektu, którego monitor będzie użyty do synchronizacji wątków próbujących ten kod wykonać. Synchronizacja dotyczy tylko i wyłącznie operacji – a więc instrukcji, nigdy zmiennych czy całych klas – mimo że przeważnie chodzi de facto o ochronę spójności stanu. Fragment kodu synchronizujemy umieszczając go w obrębie bloku `synchronized`, tak jak to pokazano poniżej:

```
synchronized (mutex) {
  // kod który synchronizujemy na monitorze obiektu mutex
}
```

Identyfikator `mutex` to referencja do obiektu którego monitor zostanie użyty to synchronizacji tego kodu. Alternatywie, możemy synchronizować całą metodę

dodając do jej deklaracji modyfikator `synchronized`. Synchronizacja metody to nic innego jak synchronizacja całego kodu zawartego w tej metodzie. Synchronizując metodę nie mamy możliwości określenia obiektu którego monitor zostanie użyty – metody instancyjne synchronizowane są zawsze na obiekcie `this`, zaś metody klasowe (statyczne) na obiekcie klasy w której zostały zaimplementowane. Deklaracja:

```
synchronized void someOp() {
  // synchronizowany kod
}
```

jest więc równoważna deklaracji:

```
void someOp() {
  synchronized (this) {
    // synchronizowany kod
  }
}
```

zaś deklaracja:

```
static synchronized void someOp() {
  // synchronizowany kod
}
```

jest równoważna deklaracji:

```
static void someOp() {
  synchronized (MyClass.class) { // gdy kod pochodzi z klasy MyClass
    // synchronizowany kod
  }
}
```

Synchronizowany kod ma tę własność, że może go wykonywać tylko i wyłącznie ten wątek, który aktualnie posiada monitor obiektu użytego do synchronizacji – wątek musi pozyskać monitor zanim rozpocznie wykonywanie takiego kodu. Jeśli monitor ten byłby w danej chwili niedostępny, tj. byłby w posiadaniu innego wątku, to wątek bieżący zostanie uśpiony aż do czasu kiedy monitor zostanie zwolniony. Wątek który pozyskał monitor synchronizowanego bloku odda go dopiero gdy zakończy wykonanie tego kodu lub ewentualnie gdy wykona metodę `wait(…)`.

Metoda `wait(…)` to metoda finalna zaimplementowana w klasie `Object`. Wywołanie tej metody spowoduje, że bieżący wątek zawiesi swe wykonanie w oczekiwaniu aż inny wątek wywoła na tym samym obiekcie metodę `notify()` lub `notifyAll()`. Metoda `wait(…)` zaimplementowana jest w trzech wariantach. Wariant bezargumentowy powoduje, że wątek zawiesza swe wykonanie bezterminowo, wariant jednoargumentowy pozwala na określenie limitu czasowego na oczekiwanie w milisekundach a dwuargumentowy dodatkowo akceptuje liczbę nanosekund.

Metody `notify()` i `notifyAll()` to także metody finalne z klasy `Object`. Wywołanie metody `notify()` powoduje, że zostanie wznowione wykonanie jednego z wątków, które uprzednio wywołały metodę `wait(…)` dla tego obiektu. Wybór, który z czekających wątków zostanie wznowiony zależny jest od implementacji Maszyny Wirtualnej i nie możemy zakładać żadnego determinizmu w tym względzie. Aby obudzić wszystkie wątki oczekujące na danym obiekcie należy wywołać metodę `notifyAll()`.

Pyt.30 **Czy wywołanie metody *wait(…)* z jednym argumentem równym *10* gwarantuje, że wątek wznowi swe działanie po dokładnie *10* milisekundach?**

Wszystkie trzy wymienione powyżej metody mogą być wywołane tylko i wyłącznie dla obiektów których monitor znajduje się w posiadaniu wywołującego wątku, tj. tylko i wyłącznie wewnątrz metody synchronizowanej lub bloku synchronizacyjnego, które – co więcej – synchronizowane są na tym samym obiekcie dla którego wywołujemy metodę `wait(…)`, `notify()` czy `notifyAll()`. Wywołanie którejś z tych metod dla obiektu dla którego bieżący wątek nie posiada monitora zakończy się wyjątkiem `IllegalMonitorStateException`. Poniższy przykład pokazuje typowy sposób wykorzystania opisanych mechanizmów:

```
class DataBuffer {
  private Integer numericData;

  public synchronized void setData(int value) {
    numericData = value;

    notifyAll(); // wzbudź wątki oczekujące na wypełnienie bufora
  }
```

```
  public synchronized int getData() {
    while (numericData == null) {
      try {
        wait(); // poczekaj, aż inny wątek wypełni bufor
      } catch (InterruptedException exc) {
        // ignorujemy ten wyjątek - jego wystąpienie nam nie szkodzi
      }
    }

    int value = numericData;
    numericData = null; // po pobraniu danych wyczyść bufor

    return value;
  }
}
```

Klasa `DataBuffer` może służyć jako mechanizm dla bezpiecznego, synchronicznego przekazywania wartości liczbowej pomiędzy wątkami. Jeden z wątków wywołuje metodę `setData(…)` aby zapisać wartość w buforze (na zmiennej `numericData`), podczas gdy inny wątek wywołuje metodę `getData()` aby tę wartość odczytać. Obydwie metody synchronizowane są na tym samym obiekcie – obiekcie `this` – dzięki czemu mamy pewność, że nie będą one nigdy wykonywane równocześnie (przez różne wątki).

Odp.30 **Nie! Wątek może być wznowiony wcześniej, jeśli będzie rzucony wyjątek *InterruptedException*. Ponadto, nawet jeśli wątek zostanie obudzony po *10* milisekundach, to nie ma gwarancji, że natychmiast wznowione będzie jego wykonanie. Jeśli wątek ten będzie konkurował o czas procesora z innymi wątkami, to być może będzie on musiał odczekać kolejne kilka milisekund.**

Wątek wywołujący metodę `getData()` w pierwszej kolejności musi upewnić się, że dane zostały już do bufora zapisane, tj. że zmienna `numericData` ma wartość różną niż `null`. Jeśli do zmiennej tej nie została jeszcze przypisana wartość wątek ten wykona metodę `wait()` przechodząc tym samym w stan uśpienia i zwalniając monitor obiektu synchronizującego. Zauważmy, że sprawdzenie warunku `numericData == null` wykonujemy w pętli `while` a nie za pomocą instrukcji warunkowej `if`. Jest to konieczne, ponieważ stan wątku oczekującego na notyfikację (tj. wywołanie przez inny wątek metody `notify()` lub `notifyAll()`) może być zakłócony,

przez co wznowi on swe wykonanie przedwcześnie, rzucając wyjątek `InterruptedException`. Wątek musi więc ponownie sprawdzić warunek `numericData == null` i uśpić się wywołując metodę `wait()` jeśliby się okazało, że warunek ten nadal nie jest prawdziwy.

W czasie gdy wątek który wywołał metodę `wait()` oczekuje na notyfikację inny wątek ma szansę wykonać metodę `setData(…)`. Wywołując ją w pierwszej kolejności przypisze wartość do zmiennej `numericData` po czym wywoła operację `notifyAll()` co spowoduje wznowienie wykonania wątku, który czeka uśpiony po wywołaniu metody `wait()`.

Wątek wznowiony w wyniku wywołania operacji `notifyAll()` zacznie się wykonywać, jednak będzie on natychmiast wstrzymany ponownie, tyle że tym razem w oczekiwaniu na zwolnienie monitora przez wątek kończący wykonanie metody `setData(…)`. Zakończenie się metody `setData(…)` jest równoznaczne ze zwolnieniem monitora. Dopiero teraz może być zakończone wywołanie metody `getData()`.

Zauważmy, że metoda `notifyAll()` wywoływana jest – co bardzo ważne – dla tej samej instancji (instancji klasy `DataBuffer`), dla której uprzednio wywołano metodę `wait()`. Instancji klasy `DataBuffer` możemy utworzyć dowolnie wiele; wówczas operacje wykonywane na każdej z nich będą synchronizowane zupełnie niezależnie od siebie – będą przecież synchronizowane na innych obiektach, innych instancjach klasy `DataBuffer`.

STANY WĄTKÓW

Stan wątku to jedna z wartości typu wyliczeniowego `Thread.State`. Nowo utworzonej instancji klasy `Thread` odpowiada stan `NEW`. Stan ów oznacza, że wątek nie został jeszcze de facto utworzony – instancja klasy `Thread` to póki co jedynie instancja definicji wątku. Wątek tworzony jest dopiero gdy wywołamy dla tej instancji metodę `start()`. Zmienia on wówczas swój stan na `RUNNABLE`, który to stan oznacza, że wątek jest gotowy by zacząć się wykonywać, ale bynajmniej nie jest to jednoznaczne z tym, że jest on wykonywany. Maszyna

Wirtualna dzieli dostępne zasoby systemowe między aktywne wątki wykonując je naprzemiennie w bardzo krótkich przedziałach czasu, powodując że wątek często przechodzi ze stanu „wykonywany" do stanu „gotowy do wykonania". Trzeba odróżniać od siebie te dwa stany faktyczne, niemniej jednak im obydwu odpowiada systemowy stan `RUNNABLE`. W praktyce nie jesteśmy w mocy aby stwierdzić, kiedy wątek jest wykonywany, a kiedy – pozostając w stanie gotowości do wykonania – czeka na przydzielenie kwantu czasu procesora. Gdy wątek się kończy – a kończy się gdy wykona swą metodę `run()` – przechodzi do stanu `TERMINATED`.

Aktywny wątek – będący w stanie `RUNNABLE` – może zakończyć się przechodząc do stanu `TERMINATED`, ale może także wstrzymać swe wykonanie, oczekując na zajście pewnego zdarzenia lub upływ określonego interwału czasowego. Wątek który wstrzymuje swe wykonanie w oczekiwaniu na możliwość wykonania synchronizowanego fragmentu kodu przechodzi do stanu `BLOCKED`. Zerknijmy ponownie na klasę `DataBuffer`:

```
class DataBuffer {
  private Integer numericData;

  public synchronized void setData(int value) {
    numericData = value;

    notifyAll();
  }

  public synchronized int getData() {
    while (numericData == null) {
      try {
        wait();
      } catch (InterruptedException exc) { }
    }

    int value = numericData;
    numericData = null;

    return value;
  }
}
```

Załóżmy że jeden z wykonujących się wątków wywołuje dla pewnej instancji klasy `DataBuffer` metodę `setData(…)` i tym samym pobiera monitor tego obiektu.

Załóżmy że tuż po wykonaniu przypisania `numericData = value` wątek ten zostaje wywłaszczony i wznawiany jest inny wątek, który próbuje wywołać metodę `getData()`. Metoda ta nie może być jednak w tej chwili wykonana, ponieważ monitor obiektu synchronizującego jest obecnie pobrany przez wątek wykonujący metodę `setData(…)`. Wątek próbujący wywołać metodę `getData()` przejdzie zatem do stanu `BLOCKED` a jego wykonanie zostanie ponownie wstrzymane.

Wątek może wstrzymać swe wykonanie także w wyniku szeregu innych zdarzeń. W szczególności, wątek może wstrzymać się w oczekiwaniu na zakończenie innego wątku – taki efekt będzie miało wywołanie metody `join(…)`. Jest to finalna, instancyjna metoda klasy `Thread`. Oprócz wariantu bezargumentowego zaimplementowano także wariant jednoargumentowy, pozwalający określić limit czasowy oczekiwania w milisekundach oraz wariant dwuargumentowy, pozwalający dodatkowo określić liczbę nanosekund. Metoda `join(…)` wstrzymuje wykonanie bieżącego wątku w oczekiwaniu na zakończenie się wątku dla którego została wywołana. Przykład poniżej:

```
public class TestClass {
  static String sharedVar;

  public static void main(String[] args) {
    Thread setThread = new Thread() {
      public void run() {
        sharedVar = "jakaś wartość";
      }
    };

      // uruchamiamy wątek przypisujący wartość do zmiennej sharedVar
    setThread.start();

    try {
          // czekamy aż wątek przypisujący wartość się zakończy
      setThread.join();
    } catch (InterruptedException exc) { }

      // wyświetlamy wartość ustawioną przez wątek na który czekaliśmy
    System.out.println(sharedVar);
  }
}
```

W powyższym kodzie użyto metody `join()` do synchronizacji wątku głównego (wątku „main") – który kończy się wyświetleniem wartości zmiennej `sharedVar` – z wątkiem przypisującym wartość do tej zmiennej. Podobnie jak wywołanie metody `wait(…)` wywołanie metody `join(…)` może zakończyć się przedwcześnie, co zostanie zakomunikowane wyjątkiem `InterruptedException`. W takim wypadku powyższy przykład mógłby zakończyć się wyświetlaniem tekstu „null".

Wątek który po wywołaniu bezargumentowej wersji metody `join()` oczekuje na zakończenie się innego wątku, albo po wywołaniu bezargumentowej wersji metody `wait()` oczekuje na wywołanie metody `notify()` lub `notifyAll()` przechodzi do stanu `WAITING`. Wątek który wywołując metodę `join(…)` lub `wait(…)` określa limit czasowym oczekiwania (ang. timeout) przechodzi do stanu `TIMED_WAITING`. W stanie `TIMED_WAITING` oczekiwał będzie także wątek który wywołał metodę `sleep(…)`.

Metoda `sleep(…)` to metoda statyczna zaimplementowana w klasie `Thread`. Służy ona do wstrzymania na określony czas wykonania wątku wywołującego tę metodę. Wariant jednoargumentowy umożliwia określenie liczby milisekund a wariant dwuargumentowy dodatkowo liczby nanosekund. Uśpienie wątku na określony czas nie gwarantuje jednakowoż że dokładnie po upływie podanego interwału wątek zacznie się ponownie wykonywać. Po upływie podanego interwału wątek przejdzie ze stanu `TIMED_WAITING` do stanu `RUNNABLE`, który to stan oznacza że wątek jest gotowy do bycia wykonanym, ale nie koniecznie, że jest on wykonywany. Ponadto wątek oczekujący na upływ określonego czasu może zostać wzbudzony rzucając jednocześnie wyjątek `InterruptedException`.

PRIORYTETY WĄTKÓW

Każdy wątek ma przypisany pewien priorytet. Priorytet to wartość całkowita z zakresu od `Thread.MIN_PRIORITY` do `Thread.MAX_PRIORITY`. Zazwyczaj odpowiada to przedziałowi 1-10. Klasa `Thread` udostępnia metodę `setPriority(…)` która pozwala zmienić domyślny priorytet wątku, oraz metodę `getPriority()` za pomocą której

priorytet ten możemy odczytać. Wartość domyślna priorytetu jest równa stałej `Thread.NORM_PRIORITY`.

Współbieżność opiera się na podziale dostępnego czasu procesora pomiędzy aktywne – znajdujące się w stanie `RUNNABLE` – wątki. To, jaki będzie algorytm przydziału kwantów tego czasu poszczególnym wątkom jest jednak cechą konkretnej implementacji Maszyny Wirtualnej i nie można czynić co do tego żadnych założeń – aspekt ten nie jest regulowany przez specyfikację Wirtualnej Maszyny Javy. Nie mamy pewności ani co do tego w jaki sposób zmiana priorytetu wątku wpłynie na jego wykonanie, ani co do tego kiedy wątek który raz zaczął się wykonywać zostanie wywłaszczony aby umożliwić wykonanie innym, konkurującym wątkom.

Nie mamy pewności co do tego w jaki sposób zmiana priorytetu wątku wpłynie na jego wykonanie ale możemy mniemać, że z pośród kilku aktywnych wątków do wykonania w danej chwili zostanie wybrany ten o najwyższym priorytecie. W oparciu o to domniemanie możemy starać się optymalizować implementowaną aplikację. Możemy także wspomóc mechanizm wywłaszczania i wskazać Maszynie Wirtualnej moment w którym dany wątek powinien być wstrzymany na rzecz innego wątku. W tym celu umieszczamy w kodzie wywołanie metody `yield()`.

Metoda `yield()` to statyczna metoda zaimplementowana w klasie `Thread`. Wywołanie tej metody stanowi wskazówkę dla Maszyny Wirtualnej, że wątek który ją wywołał powinien być wywłaszczony na rzecz innego wątku. Jest to jednak li tylko sugestia i nie ma żadnej pewności co do tego, czy zostanie ona wzięta pod uwagę.

KOMPILATOR ORAZ MASZYNA WIRTUALNA

Niniejszy rozdział pokrywa prezentowanym zakresem wiedzy następujące wymagania egzaminu 1Z0-851:

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

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

7.4 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 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.

ASERCJE

Asercje to mechanizm pozwalający upewnić się, że nasze założenia co do stanu programu w danej chwili są prawidłowe. Przykładowo, implementując pewną metodę możemy zakładać, że wszystkie parametry wejściowe są liczbami dodatnimi, albo że referencje nie mają wartości `null`. Możemy przecież założyć, że metoda wywołująca przekaże odpowiednie wartości, a jeśli nie, to oznacza to błąd w kodzie. Właśnie dokładnie do tego służą asercje – do upewnienia się, że w kodzie nie ma błędów. Ważną cechą asercji jest, że mogą być one włączane i wyłączane. Intencją asercji jest, by były one sprawdzane tylko w fazie implementacji i testów oprogramowania, zaś w gotowej aplikacji, uruchomionej w systemie produkcyjnym już nie. Domyślnie mechanizm asercji jest zatem wyłączony, tj. asercje nie są sprawdzane. Poniżej przykład użycia asercji w kodzie:

```
private static float div(float x, float y) {
  assert y != 0 : "y equals 0";

  return x / y;
}
```

Po słowie kluczowym `assert` umieszczamy wyrażenie typu logicznego, którego prawdziwość chcemy przetestować. Dwukropek i następujący po nim komentarz (tj. dowolne wyrażenie typu tekstowego, np. wywołanie funkcji zwracającej stringa) to elementy opcjonalne. Jeśli asercja nie jest prawdziwa Maszyna Wirtualna rzuca wyjątek `java.lang.AssertionError`. Komentarz asercji zostanie wykorzystany jako opis wyjątku. Wywołanie powyższej funkcji z drugim argumentem równym 0 spowoduje wyświetlenie następującego komunikatu:

```
java.lang.AssertionError: y equals 0
  at next.Test.div(Test.java:12)
```

Asercje zostały wprowadzone do języka Java począwszy od wersji 1.4. Oznacza to, że do tego momentu `assert` nie było słówkiem kluczowym a więc w szczególności słowo to mogło być używane jako identyfikator… i pewnie czasami było. Próba kompilacji takiego kodu kompilatorem w wersji 1.5 się naturalnie nie powiedzie, chyba że powiemy explicite, że mamy do czynienia ze starym kodem, tj. dodamy do polecenia kompilacji `javac` flagę `-source 1.3`. Kod wówczas

skompiluje się poprawnie, aczkolwiek wygenerowane będą ostrzeżenia.

Aby uruchomić program w trybie sprawdzania asercji należy do polecenia `java` dodać flagę `-enableassertions`, lub w formie skróconej `-ea`. Asercje można włączać także selektywnie, dla konkretnych klas lub pakietów, podając nazwę klasy lub pakietu po fladze. Polecenia te dotyczą wszystkich managerów ładowania klas (ang. class loader) a więc wszystkich klas, z wyjątkiem klas systemowych, które obsługiwane są przez inny mechanizm ładowania. Aby włączyć sprawdzanie asercji dla klas systemowych należy użyć flagi `-enablesystemassertions` lub `-esa`.

Asercje wyłączone są domyślnie, ale można je wyłączyć także explicite. Ma to sens wówczas, gdy włączamy asercje globalnie, lub dla pewnego pakietu klas, a wyłączamy selektywnie dla niektórych klas lub pod pakietów. Do wyłączenia asercji używamy flagi `-disableassertions` lub `-da`, oraz `-disablesystemassertions` lub `-dsa` dla klas systemowych. Kilka przykładów poniżej:

```
java -ea my.package.MainClass

java -ea:my.package.MyClass my.package.MainClass

  // uwaga na trójkropek po nazwie pakietu
java -ea:my.package… my.package.MainClass

java -esa -ea -da:my.package.MyClass my.package.MainClass
```

Wiemy już jak pracuje się z asercjami, przejdźmy więc do tego, co z największym prawdopodobieństwem może pojawić się na egzaminie OCPJP – wytycznych Oracle'a kiedy asercji używać, a kiedy nie. Oto garść prostych reguł:

- Nie używaj asercji do walidacji parametrów wywołania metod publicznych. Walidacja ta powinna być wykonywana zawsze, więc mechanizm asercji nie jest odpowiedni, jako że można go włączać i wyłączać, a co więcej, w gotowych aplikacjach przeważnie jest wyłączony.

- Nie używaj asercji do walidacji parametrów przekazanych z linii poleceń. Powód jest dokładnie taki sam, jak dla wywołań metod publicznych – w końcu metoda `main(…)` to metoda publiczna.

- Używaj natomiast asercji do walidacji parametrów wywołania metod prywatnych. W przeciwieństwie do metod publicznych, te wywołania są w pełni kontrolowane przez osobę, która zaimplementowała klasę. Walidacja parametrów ma tu znaczenie tylko kontrolne, na etapie testowania, więc asercje są tutaj OK.

- Używaj asercji dla upewnienia się, że nie wystąpiła sytuacja, która nie powinna była wystąpić w żadnych warunkach – jeśli wystąpiła to znaczy, że program zawiera błąd. Przykładowo, jeśli z naszych kalkulacji wynika, że nie jest możliwe, aby wykonanie programu dotarło do pewnego miejsca (np. w gąszczu instrukcji `if-else`) to umieśćmy w tym miejscu instrukcję `assert false;` aby zwalidować poprawność naszych kalkulacji.

Wyrażenia używane przez asercje nie powinny mieć efektów ubocznych. Zarówno wyrażenie logiczne jak i komentarz mogą być wywołaniami metod, a te mogą mieć skutki uboczne, jednak nie chcemy aby nasz program miał inne działanie w zależności od tego czy asercje są weryfikowane czy nie. Asercje nie powinny wywoływać instrukcji które zmieniają wartości zmiennych czy powodują wyświetlenie komunikatów.

MECHANIZM ODZYSKIWANIA PAMIĘCI

Obiekty w Javie, tak jak wszystko w przyrodzie, mają swój początek i koniec. Cykl jest prosty – tworzymy, używamy i… porzucamy. Zwróćmy uwagę na ostatnie słowo; porzucamy, ale nie niszczymy. Niszczeniem zajmuje się odśmiecacz (ang. garbage collector). Naturalnie, odśmiecacz niszczy tylko obiekty porzucone, a więc takie, które już nigdy nie będą mogły zostać użyte. Ktoś mógłby powiedzieć, że niepotrzebne obiekty to takie, na które nie wskazuje żadna referencja. Jest to prawda, ale tylko częściowa. Rzeczywiście, obiekty na które nie

wskazuje żadna referencja są już bezużyteczne i podlegają odśmiecaniu, ale nie jest to warunek konieczny. Zerknijmy na poniższy przykład:

```
public class Test {
  public static void main(String[] args) {
    SomeClass oneInstance = new SomeClass();

    oneInstance.otherInstance = new SomeClass();

    oneInstance = null;
  }
}

class SomeClass {
  public SomeClass otherInstance;
}
```

W pierwszej linii metody `main(…)` tworzymy nową instancję klasy `SomeClass` i przypisujemy ją do zadeklarowanej referencji `oneInstance`. W drugiej linii tworzymy kolejny obiekt i przypisujemy go do zmiennej instancyjnej obiektu utworzonego uprzednio. Zauważmy, że póki co oba obiekty są osiągalne z aktywnego wątku aplikacji – pierwszy obiekt jest osiągalny bezpośrednio, poprzez referencję `oneInstance` a drugi pośrednio. W trzeciej linii metody `main(…)` przypisujemy do zmiennej `oneInstance` wartość `null`; tym samym powodujemy, że żadna z dwu utworzonych uprzednio instancji nie jest już osiągalna z aktywnego wątku aplikacji. Co prawda żadna referencja nie wskazuje już na obiekt utworzony jako pierwszy, ale na drugi z obiektów cały czas wskazuje referencja `otherInstance`. Mimo to, oba obiekty są już bezużyteczne i podlegają procesowi odśmiecania. Definicja obiektu porzuconego, a więc podlegającego procesowi odśmiecania jest więc taka, że jest to obiekt który nie jest bezpośrednio ani pośrednio osiągalny z żadnego aktywnego wątku wykonania.

Pozostało jeszcze odpowiedzieć na pytanie – kiedy uruchamiany jest proces odśmiecacza? Generalnie rzecz biorąc… nie wiadomo – zależy to od konkretnej implementacji Maszyny Wirtualnej Javy. Co prawda mamy do dyspozycji metodę `java.lang.Runtime.getRuntime().gc()`, która służy do zgłoszenia prośby o uruchamianie odśmiecania, ale to tylko tyle – zgłoszenie prośby. Wywołanie tej metody wcale nie musi zakończyć się podjęciem jakiejkolwiek akcji, jest to tylko drobna sugestia, którą kierujemy do Maszyny Wirtualnej. Jeśli jednak Maszyna Wirtualna zdecyduje się na podjęcie w odpowiedzi na naszą

sugestię akcji oczyszczania pamięci to robi to synchronicznie – proces kończy się, zanim nastąpi powrót z wywołania metody `gc()`. Ekwiwalentem dla wywołania `Runtime.getRuntime().gc()` jest `java.lang.System.gc()`; obydwie metody działają w ten sam sposób.

Ze względu na brak możliwości założenia czegokolwiek na temat sposobu działania i momentu uruchomienia odśmiecacza – oraz braku gwarancji, że dany obiekt w ogóle zostanie kiedykolwiek usunięty z pamięci – język Java nie oferuje znanych skądinąd (np. z języka C++) destruktorów; oferuje jednak pewną namiastkę – metodę `finalize()` zdefiniowaną w klasie `Object`. Nie jest to metoda finalna, więc może być nadpisana (ang. overridden) w dowolnej podklasie. Znaczenie tej metody jest zbliżone do destruktorów w tym sensie, że jest to metoda wywoływana automatycznie przez proces odśmiecacza tuż przez usunięciem obiektu z pamięci, tyle, że jest jedno ale. Spójrzmy na poniższy przykład:

```
public class ResurrectionClass {
  public static Set<Object> collection;
}

class SomeClass {
  @Override
  public void finalize() {
    ResurrectionClass.collection.add(this);
  }
}
```

Kod ten jest jak najbardziej poprawny. Jako że metoda `finalize()` to metoda jak każda inna, to możliwe jest umieszczenie w niej kodu, który powoduje, że dana instancja zaczyna być osiągalna z aktywnego wątku a więc nie może być usunięta. I nie jest – proces usuwania obiektu z pamięci zostaje zaniechany. Nie czyni to jednak instancji tej klasy nieśmiertelnymi, bowiem metoda `finalize()` wywoływana jest dla każdego usuwanego obiektu tylko i wyłącznie raz! Proces odśmiecania wywoła metodę `finalize()` za pierwszym razem, gdy próbuje usunąć obiekt z pamięci i wtedy możliwe jest wykonanie operacji pokazanej powyżej, jednak już za drugim razem metoda ta nie jest wywoływana i obiekt jest nieuchronnie niszczony.

KOMPILACJA PROGRAMU

Programując w języku Java przeważnie używamy środowiska zintegrowanego w stylu Eclipse lub NetBeans IDE, które nie wymagają umiejętności posługiwania się kompilatorem standardowym Java SE (uruchamianym z linii poleceń), jednak egzamin na OCPJP, jak to często bywa z egzaminami jest nie do końca praktyczny. Musimy wiedzieć, zarówno jak kompilować jak i jak uruchamiać skompilowane już aplikacje posługując się linią poleceń i poleceniami `java` i `javac`.

Polecenie `javac` służy do uruchamiania kompilatora języka Java. Akceptuje ono szereg różnych parametrów, których listę wraz z opisem możemy zobaczyć wydając polecenie:

```
javac -help
```

Z punktu widzenia kompilatora `javac` nie ma czegoś takiego jak program czy aplikacja – są tylko pliki zawierające kod. Oznacza to tyle, że aby skompilować cały program musimy skompilować każdy z plików tworzących tenże i to my musimy zadbać, żeby każdy z plików został skompilowany. Załóżmy, że mamy klasy `ClassOne` (w pliku ClassOne.java) i `ClassTwo` (w pliku ClassTwo.java) w podkatalogu `my/pckg/` oraz klasę `OtherClass` (w pliku OtherClass.java) w podkatalogu `other/pckg/`, przy czym klasa `OtherClass` wygląda następująco:

```
public class OtherClass {
  private ClassOne classOne;

  private ClassTwo classTwo;
}
```

Klasy `ClassOne` i `ClassTwo` są puste (tj. nie zawierają żadnych metod czy zmiennych). Zacznijmy od skompilowania tych klas. W tym celu możemy użyć jednego z następujących poleceń:

```
javac my/pckg/*.java
javac my/pckg/ClassOne.java my/pckg/ClassTwo.java
```

Do polecenia `javac` przekazujemy listę plików które chcemy skompilować (nazwy plików oddzielamy spacjami), przy czym możemy wymienić ich nazwy explicite albo określić wzorzec nazwy z użyciem znaków wieloznacznych (ang. wildcards) , tj. `*` oraz `?`. Uwaga! Jeśli użyjemy wzorca `*` albo `*.*` a w katalogu znajdować się będą pliki nieposiadające rozszerzenia „.java" (np. skompilowane pliki z rozszerzeniem „.class") to uruchomienie kompilacji się nie powiedzie. Nieistotna jest nawet zawartość pliku – jeśli nazwa pliku nie jest zakończona sufiksem „.java" to kompilacja nie powiedzie się, nawet gdyby zawartość pliku była prawidłowa. Pliki kodu źródłowego zawsze muszą mieć rozszerzenie „.java" a pliki skompilowanych klas zawsze muszą mieć rozszerzenie „.class". Co to znaczy, że uruchomienie kompilacji się nie powiedzie? Otóż z kompilacją związane są dwa rodzaje błędów: błędy niepoprawnego uruchomienia kompilatora oraz błędy kompilacji plików. Jeśli spróbujemy uruchomić kompilator w sposób niepoprawny (np. każemy kompilować pliki o nieprawidłowej nazwie) to wyświetlony zostanie komunikat błędu uruchomienia programu oraz informacje identyczne z tymi które byśmy otrzymali wydając polecenie `javac -help`. Jeśli proces kompilacji zostanie uruchomiony, natomiast wystąpią błędy w czasie kompilacji plików to zostaną wyświetlone komunikaty błędów kompilacji związane z błędami implementacji.

Spróbujmy teraz skompilować klasę `OtherClass`. Zauważmy, że klasa ta używa dwu pozostałych klas jako typów swoich zmiennych – typy te muszą być w czasie kompilacji widoczne; albo jako pliki źródłowe albo już skompilowane. Możliwości są dwie: albo skompilujemy wszystkie klasy na raz, albo kompilując klasę `OtherClass` oddzielnie zadbamy o to by klasy `ClassOne` i `ClassTwo` znajdowały się na ścieżce klas (ang. classpath). Ścieżkę klas określa zmienna środowiskowa `CLASSPATH`, ale może być nadpisana z użyciem parametru `-classpath` albo `-cp`. Jeśli nie zdefiniowano zmiennej środowiskowej `CLASSPATH`, ani nie przekazano parametru `-classpath` czy `-cp` to za ścieżkę klas przyjmuje się katalog bieżący (ten w którym uruchamiamy polecenie `javac`). Klasę `OtherClass` możemy więc skompilować jednym z poleceń:

```
javac my/pckg/*.java other/pckg/*.java

javac -classpath . other/pckg/*.java
```

przy czym w drugim wypadku określenie parametru `-classpath` nie jest konieczne jeśli zmienna `CLASSPATH` nie jest zdefiniowana – ścieżką klas jest przecież wówczas katalog bieżący (kropka oznacza właśnie katalog bieżący).

Jeśli nie wskażemy inaczej, pliki powstałe w wyniku kompilacji zostaną umieszczone w tym samym katalogu co pliki źródłowe. Aby oddzielić pliki klas od plików źródłowych należy użyć parametru `-d` tak jak pokazano poniżej:

```
javac -d build/ my/pckg/*.java
```

Uruchomienie powyższego polecenia spowoduje skompilowanie wszystkich plików źródłowych z podkatalogu `pckg` katalogu `my` i umieszczenie plików klas w katalogu `build`. Skompilowane pliki klas zawsze umieszczane są w odpowiedniej – odzwierciedlającej pakiety – strukturze katalogowej. Parametr `-d` służy jedynie do określenia katalogu bazowego dla wyniku kompilacji. Kompilator utworzy wszystkie katalogi potrzebne do odzwierciedlenia struktury pakietów, a więc w powyższym przypadku katalog `my` oraz `pckg`, jednak katalog bazowy – wskazany parametrem `-d` – musimy utworzyć sami. Jeśli katalog wskazany parametrem `-d` nie istniałby kompilacja zakończyła by się błędem.

URUCHAMIANIE PROGRAMU

Skompilowaną aplikację uruchamiamy za pomocą polecenia `java`. Uruchomienie aplikacji to tak naprawdę wywołanie metody `main(…)` zdefiniowanej w jednej z klas. Uruchamiając aplikację podajemy nazwę klasy której metodę `main(…)` chcemy uruchomić, przekazując jednocześnie parametry wywołania tej metody. Załóżmy więc, że chcemy uruchomić metodę główną klasy `my.pckg.SomeClass`, nie przekazując jej żadnych parametrów. Jeśli skompilowane klasy (w odpowiedniej strukturze katalogowej) znajdują się w katalogu `/usr/java/build` to wydajemy w tym celu polecenie:

```
java -cp /usr/java/build my.pckg.SomeClass
```

Uruchamiając polecenie `java` wystarczy określić nazwę klasy (oraz zadbać o to aby klasa ta znajdowały się na ścieżce klas, np. używając parametru `-cp`); nie

podajemy nazwy metody, jako że uruchamiana jest zawsze publiczna, statyczna metoda `main(…)` typu `void` akceptująca tablicę stringów. Typowa deklaracja metody `main(…)` wygląda następująco:

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

jednak może być ona także nieco inna. Metoda musi być publiczna i statyczna, jednak nie ma obowiązku określania modyfikatorów w pokazanej powyżej kolejności, poprawna więc będzie także deklaracja:

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

Pyt.31 Jak powinien nazywać się parametr metody *main(…)*?

Metoda `main(…)` to koniec końców zwykła metoda, tak więc nie ma najmniejszego znaczenia jak nazwaliśmy parametr jej wywołania. Zwyczajowo używa się nazwy „args”, jednak może to być dowolny poprawny identyfikator. Co więcej, zamiast tablicy stringów możemy użyć argumentu wielo-arnego typu `String`; przykład poniżej:

```
static public void main(String... varArgParams) {
  // implementacja metody
}
```

Parametry określone w trakcie wywołania polecenia `java` przekazane są do metody `main(…)` właśnie za pomocą jej argumentu. Aby wywołać metodę:

```
public static void main(String... args) throws Exception {
  if (args.length != 2)
    throw new Exception("Podaj dwa parametry!");

  System.out.println(args[0]);
  System.out.println(args[1]);
}
```

przekazując jej wymagane dwa parametry powinniśmy wydać polecenie:

```
java -cp /usr/java/build my.pckg.SomeClass paramA paramB
```

Nie dajmy się nabrać na prosty chwyt egzaminacyjny – parametr przekazany jako pierwszy znajdował się będzie w tablicy w komórce o indeksie 0, drugi w komórce o indeksie 1 itd.; tablice są przecież numerowane od zera. Przy okazji zauważamy, że metoda `main(…)` – zupełnie jak każda inna metoda – może deklarować i rzucać wyjątki. W powyższym przykładzie rzucamy (uprzednio zadeklarowany) wyjątek `java.lang.Exception` jeśli program zostanie wywołany z nieodpowiednią liczbą parametrów.

Parametry dla aplikacji możemy przekazać także w formie właściwości systemowych (ang. system properties). Statyczna metoda `getProperties()` klasy `java.lang.System` zwraca instancję klasy `java.util.Properties`, która udostępnia właściwości systemowe w formie par nazwa-wartość. Wśród właściwości standardowych takich jak nazwa systemu operacyjnego czy wersja Wirtualnej Maszyny Javy znajdują się właściwości zdefiniowane z użyciem parametru `-D` w trakcie uruchamiania programu. Aby zdefiniować właściwość o nazwie „myName” i wartości „My name is Mariusz” należy wydać polecenie:

Odp.31 **Paramet ten może się nazywać dowolnie, byleby był to poprawny identyfikator. Jego nazwa nie ma żadnego znaczenia.**

```
  // po fladze -D i wokoło znaku = nie może być spacji
java -cp /usr/java/build -DmyName="My name is Mariusz" my.pckg.SomeClass
```

Właściwość tę możemy pobrać w aplikacji za pomocą metody `getProperty(…)` klasy `Properties`. Wariant jednoargumentowy tej metody zwraca dla podanej nazwy właściwości jej wartość lub `null` jeśli właściwość nie została zdefiniowana. Wariant dwuargumentowy pozwala określić (poprzez drugi argument) wartość domyślną właściwości, zwracaną zamiast wartości `null`. Przykład poniżej:

```
public static void main(String[] args) {
  Properties props = System.getProperties();

  System.out.println(props.getProperty("myName"));
}
```

Parametr `-D` możemy podać dowolną ilość razy, definiując tym samym dowolną ilość właściwości systemowych.

MECHANIZM WYSZUKIWANIA KLAS

Kompilując pliki aplikacji musimy zadbać o to, aby wszystkie typy używane przez kompilowany kod były dla kompilatora dostępne, by mógł on zweryfikować poprawność tego kodu. Analogicznie, uruchamiając program musimy zadbać o to, aby zarówno klasa uruchamiana explicite, jak i wszystkie używane przez nią typy były dostępne dla środowiska uruchomieniowego. Typy standardowe platformy Java SE są zawsze dostępne dla kompilatora i Maszyny Wirtualnej Javy, bez konieczności podejmowania jakichkolwiek kroków. Dodatkowo, Środowisko Uruchomieniowe (ang. Java Runtime Environment, w skrócie JRE) oferuje mechanizm rozszerzeń – wystarczy skopiować pliki JAR zawierające dowolne klasy do podkatalogu `lib/ext` katalogu domowego Środowiska Uruchomieniowego i staną się one dostępne dla wszystkich uruchamianych i kompilowanych aplikacji, zupełnie jakby były częścią biblioteki standardowej Java SE. W poszukiwaniu klas zostaną przejrzane także katalogi wymienione na ścieżce wyszukiwania klas. Mechanizm ładowania klas przeszukuje źródła definicji typów w wymienionej powyżej kolejności i kończy poszukiwanie w chwili znalezienia pierwszej odpowiadającej definicji. Oznacza to, że jeśli ten sam typ zdefiniowany jest zarówno w bibliotece umieszczonej w podkatalogu `lib/ext` jak i w którymś z plików znajdującym się w katalogu ze ścieżki poszukiwania klas to użyta zostanie definicja z podkatalogu `lib/ext`. Lista lokalizacji wymienionych na ścieżce wyszukiwania klas przeglądana jest w kolejności od lewej do prawej.

Ścieżki wyszukiwania klas możemy określić za pomocą parametru `-classpath` (lub `-cp`) dodawanego do poleceń `java` i `javac`. Możemy je określić także ustawiając zmienną środowiskową o nazwie `CLASSPATH`. Jeśli nie zdefiniowaliśmy zmiennej `CLASSPATH` ani nie użyliśmy parametru `-classpath` (lub `-cp`) to za ścieżkę wyszukiwania klas przyjmuje się katalog bieżący, tj. katalog z którego wykonano polecenie `java` lub `javac`. Jeśli zdefiniowano zmienną `CLASSPATH` ale nie użyto parametru `-classpath` to

przeszukiwane będą katalogi określone tą zmienną. Jeśli podano parametr `-classpath` to zmienna `CLASSPATH`, nawet jeśli jest zdefiniowana, zostanie zignorowana i przeszukiwane będą wyłącznie katalogi określone przez parametr `-classpath`. Jeśli chcemy określić więcej niż jeden katalog (a tak jest zazwyczaj) to wymieniamy wszystkie oddzielając je od siebie znakiem `:` (albo `;` w zależności od systemu operacyjnego, jednak na egzaminie używane są znaki `:`). Uwaga! Na liście katalogów w zmiennej `CLASSPATH` czy określonej poprzez parametr `-classpath` musimy explicite określić katalog bieżący, jeśli chcemy by pliki z tego katalogu były uwzględniane przy wyszukiwaniu typów. Jeśli więc określimy listę katalogów przeszukiwania jako:

```
-classpath /usr/java/build:/usr/home/build
```

to przeszukiwane będą katalogi `/usr/java/build` oraz `/usr/home/build` (w tej właśnie kolejności), ale nie katalog bieżący. Katalog bieżący oznaczamy poprzez kropkę, tak więc chcąc go uwzględnić powinniśmy napisać:

```
-classpath /usr/java/build:/usr/home/build:.
```

Wskazane katalogi zostaną przeglądnięte w poszukiwaniu plików klas (plików z rozszerzeniem „.class"), zaś aby uwzględnić także archiwa JAR trzeba wskazać konkretne pliki; nie wystarczy wskazać katalog z archiwami (z plikami „.jar"). Jeśli chcemy do ścieżki wyszukiwania klas dodać klasy z archiwum `mylib.jar` umieszczonego w katalogu `/usr/java/lib` to musimy napisać:

```
-classpath /usr/java/build:/usr/home/build:.:/usr/java/lib/mylib.jar
```

ARCHIWA JAR

Archiwum JAR to zbiór klas języka Java (plików w odpowiedniej strukturze katalogowej) spakowanych algorytmem ZIP do pojedynczego pliku z rozszerzeniem „.jar". Do tworzenia archiwów JAR służy program `jar` wchodzący w skład dystrybucji Java SE. Załóżmy, że zaimplementowaliśmy kilka klas w pakiecie `my.pckg` i `my.other`. Aby stworzyć archiwum o nazwie `arch.jar` zawierające wszystkie podkatalogi i pliki z katalogu `my` (a więc wszystkie klasy z

pakietów `my.pckg` i `my.other`) należy wykonać następujące polecenie (przy założeniu, że katalog `my` znajduje się w bieżącym katalogu):

```
jar -cf arch.jar my
```

A co by się stało, gdybyśmy spakowali poleceniem `jar` katalog `pckg` (podkatalog katalogu `my`)? W trakcie pakowania nic złego, tzn. archiwum JAR zostałoby utworzone, tyle że było by ono bezużyteczne. Załóżmy, że kompilujemy pewien kod, który używa klas z pakietu `my.pckg` i chcemy dostarczyć definicje tych klas poprzez dodanie do ścieżki klas zbudowanego wcześniej pliku `arch.jar`. Kompilator przeszukując ścieżkę klas zajrzy do wskazanego pliku `arch.jar`, ale nie znajdzie tam katalogu `my` a zatem stwierdzi, że archiwum to nie zawiera żadnych klas z pakietu `my.pckg` i kompilacja zakończy się błędem. Zasada, że plik zawierający definicje typów z pewnego pakietu musi znajdować się w strukturze katalogowej odzwierciedlającej ten pakiet dotyczy także plików JAR. Jeśli więc chcemy, aby archiwum JAR mogło służyć jako biblioteka, to musi ono zawierać odpowiednią, pełną strukturę katalogów odzwierciedlającą pakiety – ani mniej, ani więcej.