SZYBKA PRZYPOMINAJKA Z JAVY

Odnośnie różnych haczyków i pułapek, które są małe, wredne, ale lubią się pojawiać

Spis zawartości

Autor: Maciek Trzciński

SZYBKIE SPRAWDZENIE KODU W 10 KROKACH

1. Sprawdź implementację interfejsów i klas
    1. Czy nie-defaultowe metody są zdefiniowane w klasach?
    2. Czy prawa dostępu nie są ograniczane (uwaga na interfejsy)?
    3. Czy przesłaniane funkcje nie różnią się samym typem zwracanym?
    4. Czy tam, gdzie powinien, pojawia się konstruktor super?
    5. Czy klasy dziedziczą po jednej klasie i implementują jeden/wiele interfejsach, a interfejsy dziedziczą po jednym/wielu interfejsach?
    6. Które metody zostają przysłonięte?
2. Sprawdź wywołania metod
    1. Czy każda referencja ma prawo wywołać metodę, którą wywołuje?
    2. Czy w metodach zgadzają się argumenty (uwaga na argumenty, które są zawsze typu Object, np. w equals!)?
    3. Dla której klasy w hierarchii dziedziczenia referencja wywoła metodę?
    4. Czy metody, które rzucają sprawdzane wyjątki, są odpowiednio handlowane?
    5. Sprawdź operatory: czy ==, != są używane tylko dla pasujących typów?
3. Sprawdź referencje i wartości
    1. Operujemy na typie prostym (wartość) czy obiekcie (referencja)?
    2. Co się stanie, gdy zmienimy referencję?
4. Sprawdź składniki statyczne
    1. Czy nie ma próby odwołania się do niestatycznego składnika ze statycznej metody?
5. Sprawdź składniki finalne – czy nigdzie nie są nadpisywane? Jaka jest ich wartość?
6. Sprawdź przypisania
    1. Czy mamy prawo przypisać ten obiekt do tej referencji?
    2. Czy zachodzi niejawna konwersja (może nielegalna)?
7. Sprawdź wyjątki
    1. Czy bloki try mają swoje catche? Odpowiednie catche?
    2. Czy kolejność catch nie spowoduje błędu kompilacji?
    3. Które wyjątki są niesprawdzane, a które sprawdzane?
8. Sprawdź klasy wewnętrzne
    1. Czy tworzenie klas wewnętrznych jest robione na rzecz instancji klas zewnętrznych?
    2. Czy klasy wewnętrzne mają dostęp do niefinalnych pól statycznych klasy zewnętrznej?
    3. Czy statyczne klasy wewnętrzne mają dostęp do niestatycznych pól klasy zewnętrznej?
9. Sprawdź typy sparametryzowane
    1. Czy nie ma próby stworzenia tablic lub obiektów typów generycznych?
    2. W wypadku parametrów dedukowanych – jaka jest ich wartość?
    3. Czy nie ma prób parametryzowania typem prymitywnym?
    4. Czy gdzieś są raw types? Jeśli tak, to jak się zachowają?
10. Spokój, zrób inne zadanko, a potem wróć do tego i sprawdź ponownie ;)

Primitive types

- Prymitywne typy są **inicjalizowane** (zerem, falsem)
- Konwersja:
  - **konwersja tracąca dane musi być jawna!** (zabronione: float f = 1.2 albo int i = 2/2.f)
  - Przy wykonywaniu operacji zachodzi automatyczna konwersja do bardziej skomplikowanych typów (double d = 2/2.f → tu najpierw int → float, potem float → double)
  - Operacje na intach są operacjami na intach (float f = 3/2 będzie miał wartość 1)
- Typy prymitywne są przekazywane/porównywane przez wartość, nie przez referencję!

Obiekty: podstawy

- Obiekty funkcjonują na **referencjach** – przypisanie, porównanie odbywa się przez referencję, nie przez zawartość
- Inicjalizowane nullem
- Klasa **Object**
  - protected Object clone( ) → zwraca kopię obiektu taką, że klon != oryginał
  - public boolean equals(Object obj) → porównuje z **obiektem** obj, który nie musi być tego samego typu co przesłaniający element(!) dlatego **domyślnie zwraca false** dla różnych dziwactw
  - protected void finalize( )
  - public String toString( ) → ma swoją domyślną implementację
  - public **final** Class getClass( ) → zwraca klasę obiektu (NIE PRZESŁANIAMY!)
- this → referencja do tego obiektu

Interfejsy

- przed interfejsem stoi niejawne abstract
- każda metoda jest domyślnie public
  - interfejs może mieć lokalne metody private, ale muszą być one zdefiniowane (java 9 +)
  - mogą istnieć metody public static, które są zdefiniowane
- każde pole jest domyślnie public static final

Dziedziczenie

- Klasa extenduje klasy i implementuje interfejsy
- Interfejs extenduje interfejsy
- **super**( ) jest konstruktorem klasy bazowej
- **super.pole** → odnosi się do pola jakby było polem klasy bazowej (działa na przesłony)
- metody **private** – są związane z daną klasą i nie mogą być przesłanianie (co więcej, jeśli w klasie bazowej jest final private f( ), to w klasie pochodnej może się pojawić private f( ) a nawet protected f( ) / public f( )! i to będzie inna funkcja)
- w klasie muszą zostać rozwinięte wszystkie nie-defaultowe metody implementowanego interfejsu (defaultowe **nie muszą**)
- **nie można ograniczać widoczności metod** (uwaga! W interfejsach są publiczne!)
- konstruktory:
  - użycie odniesienia do nie-finalnych metod klasy bazowej jest niebezpieczne, bo mogły one zostać przesłonięte
  - użycie super( ) blokuje użycie this( )
- Do polimorfizmu uwaga ogólna: **trzeba uważać na to, gdzie co jest. Nie wolno wywołać z referencji funkcji, której nie ma z poziomu tej referencji.**

Stringi

- Coś w stylu ”AAA” jest konstruktorem Stringa i jest równoznaczne z new String(”AAA”)
- null toString = ”null”
- boolean toString = ”true” albo ”false”
- ENUM.X toString = ”X”
- Stringi są stałe i nie da się ich zmieniać (tak naprawdę). Można co najwyżej przypisać referencję do nowego Stringa, ale poprzedni zostanie (ważne, bo każdy łańcuch tekstowy stworzony nawiasami ma jedno miejsce w programie i == będzie działało! Nie dotyczy to Stringów tworzonych operatorem new)
- double to String nie pisze zer na końcu
- NIE zachodzi automatyczna konwersja z Obiektów na Stringi (niedopuszczalne String s = mojaInstancjaMojegoObiektu), tylko tam, gdzie String jest argumentem (np. w System.out.print(String))

Static

- nie-statyczny dostęp do statica (na rzecz instancji) wygeneruje ostrzeżenie
- nie-statyczny dostęp do statica patrzy tylko na referencję (jej wartość może być null!)
- static są inicjalizowane w compile time, mogą mieć swój blok inicjalizacyjny
- statyczne metody i pola nie są częścią obiektu → nie są np. serializowane

Final

- metoda: nie da się jej przesłonić
- klasa: nie da się po niej dziedziczyć
- pole: jego referencja staje się niezmienna (ale nie nie-modyfikowalna)
  - dla pól prymitywnych uzyskujemy taki faktyczny const

Enumy

- pola są static final
- elementy mogą mieć więcej niż tylko nazwę (potrzebny konstruktor)
- elementy mogą mieć swoje funkcje
- elementy **nie rzutują się do intów jak w cepie**!

Klasy w klasach

- **Zwykłe**
  - mają dostęp do wszystkich (również prywatnych) pól klasy zewnętrznej
  - mogą istnieć tylko na rzecz instancji klasy zewnętrznej (tworzymy je np. tak: (new Zewn( )).new Wewn( )
  - Jej pola **przesłaniają** pola klasy zewnętrznej
  - Mogą zawierać pola statyczne, ale tylko final
- **Statyczne**
  - mają dostęp jedynie do statycznych składników klasy zewnętrznej (niezależnie od ich praw dostępu), istnieją niezależnie (tworzymy je np. tak: new Zewn.StaticWewn( ))
  - mogą występować w interfejsach
  - mogą zawierać nie-finalne pola statyczne
  - Służą np. do grupowania klas (wiele klas o podobnym działaniu w jednym interfejsie)
- **Lokalne**
  - wewnątrz metod
  - nie mają dostępu do nie-finalnych pól klasy zewnętrznej

- **Anonimowe**
  - Tworzymy je np. myInteface mi = new myInterface { …… };
  - nie mają dostępu do nie-finalnych pól klasy zewnętrznej
- **Dziedziczenie**
  - po zwykłej klasie wewnętrznej: musimy przekazać do konstruktora klasy dziedziczącej klasę zewnętrzną i w konstruktorze klasy dziedziczącej wywołujemy zewn.super( ) ← konstruktora

Lambdy
- Przy tylko jednym argumencie można pominąć ( ), przy tylko jednej instrukcji zwracanej można pominąć { return …. }
- argumenty lambdy mogą być domyślne
- Przykład **każda z tych lambd jest równoznaczna**:
  - (double x) - > {return x*x;};
  - (x) - > x*x;
  - x - > x*x;
  - x - > {return x*x;};
- Lambdę możemy przypisać do interfejsu funkcjonalnego → interfejs funkcjonalny ma tylko jedną metodę!

Referencje do metod
- Obiekt::metoda → metoda na rzecz obiektu
- Klasa::metoda → metoda statyczna
- Klasa::new → konstruktor

Metody o zmiennej liczbie argumentów
- typZwracany nazwa(argumentyZwykłe, typ … nazwaTablicy)

Wyjątki
- Wyjątki muszą dziedziczyć po Exception, a żeby dało się je rzucać, to implementować Throwable
- Hierarchia handlera musi być możliwa (lub błąd kompilacji)
- try bez catcha to błąd kompilacji
- Wyjątki dziedziczące po RuntimeException nie muszą być deklarowane jako rzucane
- finally { } zostanie wywołane zawsze, przed przerzuceniem wyjątku → tu mamy takie dobre rzeczy jak np. zamykanie połączenia
- uwaga: jeśli używasz catch(Exception e) { } to płoń w piekle

Klasa Class
- Uzyskujemy obiekt Class:
  - MyObject.getClass( )
  - MyClass.class
  - Class.forName
- jest **gwarancja**, że boolean Class::equals(Object obj) działa poprawnie
- Metody:
  - String getName( )
  - boolean isInterface( )
  - boolean isArray( )
  - boolean isInstance(Object obj)

Refleksje

- Constructor[ ] getConstructors( ), Field[ ] getFields( ), Method[ ] getMethods ( ), Method[ ] getDeclaredMethods
- Metody wywołujemy metodą methodName.invoke(Object, arguments) → uwaga, dla metod, które nie przyjmują argumentów, drugi argument jest nullem
- Do pól mamy dostęp przy użyciu fieldName.set(Object, value)

Strumienie

- **Bajtowe:**
  - FileInputStream, FileOutputStream (w konstruktorze dajemy zmienną File)
  - BufferedInputStream, BufferedOutputStream → opakowania
    - Mają metody write i read
- **Znakowe**
  - FileReader, FileWriter, StringReader, StringWriter
  - BufferedReader, BufferedWriter → opakowania
    - Mają metody do czytania i pisania znaków, linii
- **Scanner** → opakowujemy nim źródło, InputStream, String lub File
  - String next( ), int nextInt( ), double nextDouble, String nextLine( )

Serializacja

- Transient ją wyłącza
- niejawnie jest tworzone pole final static long serialVersionUID
- Korzystamy z ObjectInputStream i ObjectOutputStream, które opakowują (w konstruktorze) InputStream/OutputStream
- Serializable nie uruchomia konstruktora przy czytaniu obiektu, Externalizable włącza
- Brak odpowiedniej klasy powoduje ClassNotFoundException

Klasy parametryzowane

- przy metodach piszemy parametryzację przed lub po nazwie
- przy klasach piszemy po nazwie
- **nie można:**
  - utworzyć nowego obiektu typu parametryzowanego
  - tworzyć statycznych zmiennych typu sparametryzowanego ani statycznych metod, które go zwracają
  - utworzyć tablicy obiektów typu parametryzowanego
  - parametryzować typami prymitywnymi (można Wrapperami)
  - parametryzować typem implementującym Throwable
  - nie można rzutować do typów parametryzowanych
  - nie można używać instanceof dla typów parametryzowanych
  - przy zwracaniu typu parametryzowanego nie zachodzi automatyczna konwersja
- **można**:
  - parametryzować konstruktory
  - używać parametryzacji w parametryzacji (np. class ComparableBox<T implements Comparable<T>>)
- **można, ale się nie powinno:**
  - korzystać z raw types (obiekt bez parametryzacji) → wtedy parametry są traktowane jako obiekty Objects

Collections

- Ulepszona pętla **for(Drzewo : Las) { Drzewo.dodajRok( ); }**
  - Dla typów prostych dostaje **wartość** (niezmienność!)
  - Dla obiektów dostaje **niezmienną** referencję (ale można zmieniać jej pola, jej samej nie można zmienić. W sensie można, ale to nic nie da)
- **Iterator<E>**
  - gdy coś implementuje Iterable<E>, to można dostać Iterator<E> iterator( )
  - metody: E next( ) → jeśli nie może, to rzuca wyjątek NoSuchElementException, boolean hasNext( ), void remove( ) → może być użyte **tylko raz dla każdego wywołania** next( )
- **Collection**
  - implementuje Iterable
  - Object[ ] toArray
  - int size( )
  - boolean contains(Object o)
  - void clear( )
  - boolean add(E x)
- **List** (interface)
  - obsługuje indeksowanie elementów od zera, w tym:
    - E get(int i)
    - E set(int i, E x)
    - E remove(int i)
- **Set** (elementy nie mogą się powtarzać)
- **SortedSet**
  - E first( )
  - E last ( )
- **Queue**
  - boolean offer(E x)
  - E element( ) → rzuca wyjątek, E peek( ) → zwraca null zamiast wyjątku, E remove ( ) i E poll ( ) jak element i peek
- **Implementacje:**
  - ArrayList
  - LinkedList
  - TreeSet
- **Mapa**
  - Jest <K, V>
  - Nie ma iteratorów
  - metody V put (K key, V value), V get (Object key) → key musi spełniać zależność equals, nie musi być typu k!
  - Implementacja np. w **TreeMap<K,V>**
- **Comparator**: klasa implementująca Comparator<T> w środku jest public int compare(T t1, T t2), która zwraca 0, gdy obiekty są równe

Thread

- myRunnable implements Runnable musi mieć public void run ( )
- Thread myThread = new Thread(myRunnable)
- Thread.start( ) → zaczyna, Thread.join([int m]) czeka [maksymalnie m milisekund] na zakończenie wątku, throws InterruptedException
- Thread.stop( ) → kończy wątek
- synchronized(Object obj) → synchronizuje blok na obiekcie, rzuca NullPointerException jeśli podamy mu nulla

    - synchronized typ nazwa(argumenty) → synchronizuje metodę na argumentach, jeśli podamy mu nulla to nic się nie dzieje (jeśli metoda da sobie z nim radę)
- Priorytety
    - Między Thread.MIN_PRIORITY a Thread.MAX_PRIORITY (zazwyczaj 1 do 10)
    - Domyślnie Thread.NORM_PRIORITY = 5
    - w rzeczywistości to, co się z tym stanie, zależy od systemu

Adnotacje

- Definiujemy jako @interface nazwaAdnotacji { <Typ> <nazwa>( ); … }
- sprawdzamy obecność Adnotacji: naszObiekt.isAnnotationPresent(naszaAdnotacja.class)
- możemy wydobyć adnotacje Annotation an = obj.getAnnotation (nazwaAdnotacji.class) i zrzutować, a potem po prostu wyciągać z adnotacji pola, które nas interesują