**Java Efektywne Programowanie notatki:**

**Tworzenie i usuwanie obiektów**

***Temat 1. Tworzenie statycznych metod fabrycznych zamiast konstruktorów***

Statyczna metoda fabryczna – zwyczajna metoda statyczna zwracająca obiekt danej klasy.

Ex.

public static Boolean valueOf(boolean b){  
 return (b ? Boolean.*TRUE* : Boolean.*FALSE*);  
}

Zalety statycznych metod fabrycznych zamiast publicznych konstruktorów:

* Metody fabryczne posiadają nazwy – przez co kod jest łatwiejszy do analizy.

Zamiast konstruktora BigInteger (int,int,Random), który zwraca obiekt typu BigInteger, która prawdopodobnie jest liczbą pierwszą , lepiej zastosować statyczną metodę fabryczną o nazwie BigInteger.probablePrime(…)

Klasa może posiadać tylko jeden konstruktor o odpowiedniej sygnaturze, można obejść poprzez stworzenie dodatkowego konstruktora o innej kolejności typów parametrów, ale powoduje to potencjalne użycie nie tego konstruktora, którego się chce. Metody statyczne posiadają nazwy, więc można powtarzać sygnatury.

* W przeciwieństwie do konstruktorów, nie jest wymagane utworzenie nowegoobiektu podczas ich wywołania – unika powielania niepotrzebnie powielających się obiektów.

Wywołanie tego samego obiektu podczas kolejnych wywołań , może być wykorzystane do ścisłej kontroli nad istniejącymi w danej chwili obiektami – klasy,które tak robią nazywa się **kontrolowanymi przez instancję.** Pozwala to na zagwarantowanie, że klasa jest klasą typu singleton, pozwala upewnić się klasie niezmiennej, że istnieją dwa identyczne obiekty a.equals(b) jest spełnione tylko wtedy gdy a==b ( Podstawa wzorca Flyweight).

* Możliwość zwracania typu, który jest podtypem zwracanego typu – ukrycie implementacji klas pozwala na tworzenie niewielkich API. Tak działa np. java.util.Collections
* W przypadku metod fabrycznych, w zależności od tego, jakie wartości parametrów są przekazywane do metody, ulegać zmianie może też klasa zwracanego obiektu – można zwracać obiekty dowolnej klasy pochodnej do klasy zadeklarowanej jako zwracana. Klasa zwracanego obiektu może różnić się od jej odpowiednika w kolejnych wersjach pakietu. Przykładem jest Klasa EnumSet, która nie ma konstruktorów jedynie statyczne metody fabryczne. W implementacji JDK w zależności od ilości elementów do 64 zwracany jest RegularEnumSet , gdy typ wyliczeniowy zawiera ponad 64 elementy zwracany jest obiekt JumboEnumSet. Dla klienta jest to niewidoczne a może wpłynąć na wydajność.
* Klasa obiektu zwracanego przez statyczną metodę fabryczną może jeszcze nie istnieć, kiedy pisana jest klasa zawierająca metodę statyczną – taka elastyczność wykorzystywana jest w *bibliotekach dostawcy usług* np. JDBC. Biblioteka dostawcy usług jest systemem, w którym dostawcy tworzą implementację interfejsu API, dostępnego dla użytkowników biblioteki, izolując ich od implementacji. Biblioteka dostawcy usług zawiera : interfejs usługi(implementowany przez dostawce), API rejestracji dostawcy(rejestracja implementacji dająca klientom dostęp do niej), API dostępu do usług(uzyskanie instancji usługi)

Wady statycznych metod fabrycznych zamiast publicznych konstruktorów:

* Klasy nieposiadające public/protected konstruktora nie mogą być dziedziczone – Nie można na przykład dziedziczyć po żadnej z klas Collections.
* Nie można ich od razu odróżnić od innych metod statycznych – nie są umieszczane w dokumentacji wraz z konstruktorami, ciężko się zorientować po dokumentacji klasy jak skonstruować obiekt.

***Temat 2. Zastosowanie budowniczego do obsługi wielu parametrów konstruktora***

W sytuacji gdy mamy parametry wymagane oraz pola opcjonalne. Programiści wykorzystują wzorzec **konstruktora teleskopowego**, w którym tworzy się konstruktor z parametrami wymaganymi oraz kolejny z dodatkowym jednym parametrem opcjonalnym, kolejny z dwoma i tak dalej. Wywołanie konstruktora wymaga wielu parametrów czasem przekazujemy wartość zero gdy nie ma odpowiedniej wartości opcjonalnej. Trudno jest pisać kod klienta w przypadku wielu parametrów oraz kod jest nieczytelny. Łatwo o powstanie subtelnych błędów( np. zamiana miejscami parametrów).

Innym sposobem radzenia sobie z takimi sytuacjami jest wzorzec **JavaBeans** – w którym wykorzystuje się konstruktor bezparametrowy do tworzenia obiektu, a następnie wywołuje się metody ustawiające wartości każdego z wymaganych i opcjonalnych parametrów. Wadą JavaBeans jest to, że w czasie tworzenia obiektu , jest on przejściowo w stanie niespójnym. A wywołanie obiektu w stanie niespójnym może powodować błąd, w momencie dalekim od kodu problemu. Dodatkowo wymaga dodatkowego nakładu pracy by zapewnić bezpieczeństwo dla wątków.

Alternatywą dla powyższych sposobów łączącą bezpieczeństwo wzorca konstruktora teleskopowego z czytelności wzorca JavaBeans jest **Builder**.

Wzorzec Builder jest dobrym wyborem przy projektowaniu klas, których konstruktorzy lub statyczne metody fabryczne mają więcej niż kilka parametrów, szczególnie gdy większość tych parametrów jest opcjonalna.

***Temat 3. Wymuszanie właściwości singleton za pomocą prywatnego konstruktora lub typu enum***

**Singleton** – to klasa posiadająca tylko jedną instancje

Istnieją dwa typowe podejścia do tworzenia obiektów Singleton:

- W pierwszej metodzie publiczny statyczny składnik klasy jest polem oznaczonym jako final:

public class Elvis {  
 **public static final Elvis *INSTANCE* = new Elvis();**  
  
 private Elvis() {...}  
  
 public void leaveTheBuild() {...}  
  
}

Prywawtny konstruktor jest wywoływwany jedynie raz w celu inicjalizacji pola Elvis.INSTANCE. Po zainicjowaniu będzie istnieć tylko jeden egzemplarz tej klasy. Zmienić to może tylko uprawniony klient wywołując konstruktor z użyciem refleksji przy pomocy metody AccessibleObject.setAccessible( Aby się przed tym zabezpieczyć należy zmodyfikować konstruktor, by zgłaszał wyjątek, jeżeli będzie próbował utworzyć drugą instancje)

- Druga metoda tworzenia singletonów korzysta z publicznej metody fabrycznej:

public class Elvis {  
 **private** static final Elvis *INSTANCE* = new Elvis();  
  
 private Elvis() {...}  
 **public static Elvis getInstance(){return *INSTANCE*;}**  
  
 public void leaveTheBuild() {...}  
}

Wywołanie metody Elvis.getInstance zwraca referencje do tego samego obiektu ; nie są tworzone kolejne klasy Elvis( z takim samym zastrzeżeniem jak w pierwszej metodzie).

Zaletami tego podejścia są:

- Deklaracja składników klasy od razu wskazuje, że klasa ta będzie typu singleton(publiczne statyczne pole jest opatrzone atrybutem final, więc pole zawiera referencje zawsze do tego samego obiektu)

- Prostota

- Możliwość zmiany typu klas(Jeżeli uznamy, że klasa nie musi być singleton, możemy ją zmodyfikować bez zmiany definicji API)

Wady:

- W celu umożliwienia serializacji, należy napisać metodę readResolve i zadeklarować wszystkie pola w instacji jako transient

**Istnieje, też trzecia możliwość: Typ wyliczeniowy z jednym elementem**

public enum Elvis {  
 *INSTANCE*;  
 public void leaveTheBuild() {...}  
}

Podejście to zapewnia od razu obsługę serializacji, oraz daje zabezpieczenie przeciwko tworzeniu wielu obiektów, nawet w przypadku ataku złożonej serializacji lub refleksji.

**Typ wyliczeniowy jest najlepszym sposobem na implementację singletonu.**

***Temat 4. Wykorzystanie konstruktora prywatnego w celu uniemożliwienia utworzenia obiektu***

Gdy potrzebujemy stworzyć klasę grupującą metody oraz pola statyczne( Przykład klasy do grupowania związanych ze sobą metod , działających na wartościach prostych lub tablicach to java.lang.Math i java.util.Arrays) należy zablokować możliwość tworzenia obiektów takich klas. Błędnym jest stosowanie w tym celu klas abstrakcyjnych, bo można utworzyć klasę dziedziczącą i utworzyć obiekt. Dodatkowo sugerujemy, ze taka klasa przeznaczona jest do dziedziczenia. Prostym rozwiązaniem jest zastosowanie prywatnego konstruktora(blokuje to także powstawanie domyślnego konstruktora).

*//Klasa użytkowa bez możliwości tworzenia obiektu*public class UtilityClass{  
 *//zablokowanie domyślnego konstruktora* private UtilityClass(){  
 throw new AssertionError();  
 }  
 ... *//pozostała część pominięta*}

Metoda ta jest mało intuicyjna ,dlatego dobrze jest określić jasno w komentarzu przeznaczenie tego konstruktora.

Efektem ubocznym tego rozwiązania jest zablokowanie możliwości dziedziczenia.( Wszystkie konstruktory jawnie lub niejawnie muszą wywoływać dostępne konstruktory klas bazowych, a w tym przypadku taki konstruktor nie istnieje.

***Temat 5. Stosuj wstrzykiwanie zależności zamiast odwoływania się do zasobów na sztywno.***

Wiele klas zależy od jednego lub większej liczby zasobów. Przykładowo mechanizm sprawdzania poprawności ortograficznej zależy od słownika. Statyczne klas użytkowe i singletony nie nadają się do stosowania w klasach, których zachowanie jest parametryzowane pewnymi zewnętrznymi zasobami( nieelastyczne i nietestowalne, ponadto słowniki się zmieniają). W takiej sytuacji potrzeba obsługi wielu instancji, z których każda korzysta z wskazanych zasobów( tutaj odpowiedni słownik). Prosty wzorzec, który spełnia to założenie to przekazanie zasobu w konstruktorze w momencie tworzenia instancji – jest to jedna z postaci wstrzykiwania zależności.

*//Wstrzykiwanie zależności zapewnia elastyczność i łatwość testowania*public class SpellChecker {  
 private final Lexicon dictionary;  
   
 **public SpellChecker(Lexicon dictionary)**{  
 this.dictionary = Objects.*requireNonNull*(dictonary);  
 }  
   
 public boolean isValid(String word){...}  
 public List<String> suggestions(String typo){...}  
   
}

Takie podejście zachowuje niezmienność , więc wielu klientów może współdzielić zależne obiekty. Może być stosowane dla konstruktorów statycznych oraz metod fabrycznych lub budowniczych.

***Temat 6. Unikanie powielania obiektów***

Prawidłową techniką jest powtórne wykorzystanie istniejącego obiektu zamiast tworzenia nowego. Powtórne użycie jest szybsze i prezentuje lepszy styl programowania. Można także unikać powielania elementów , korzystając z metod fabrycznych zamiast konstruktorów w przypadku klas niezmiennych dostarczających oba rozwiązania. Z kolei unikanie tworzenia obiektów przez utrzymywanie własnych pul obiektów jest złym pomysłem, chyba że obiekty w puli są niezwykle ciężkie. Klasycznym przykładem obiektów, które uzasadniają zastosowanie puli obiektów ,są połączenia z bazą danych. Koszt zestawienia połączenia jest na tyle wysoki, że sensowne jest ponowne użycie tych obiektów.

***Temat 7. Usuwanie niepotrzebnych referencji do obiektów***

Mimo, że Java posiada mechanizm zbierania nieużytków, nie należy jednak uważać, że posługując się nią nie należy się zajmować zarządzaniem pamięcią.

*//Klasa z wyciekiem pamięci*public class Stack {  
 private Object[] elements;  
 private int size = 0;  
 private static final int *DEAFUT\_INITIAL\_CAPACITY* = 16;  
  
 public Stack(int initialCapacity) {  
 this.elements = new Object[initialCapacity];  
 }  
 public void push(Object e) {  
 ensureCapacity();  
 elements[size++] = e;  
 }  
 public Object pop() {  
 if (size ==0)  
 throw new EmptyStackException();  
 return elements[--size];  
 }  
 */\*\*  
 \* Zapewnienie miejsca na co najmniej jeden element, przy dwukrotnym  
 \* zwiększeniu rozmiaru tablicy za każdym razem, gdy musi być ona powiększona  
 \*/* private void ensureCapacity() {  
 if (elements.length == size) {  
 elements = Arrays.*copyOf*(elements, 2 \* size +1);  
 }  
 }  
}

Zwracając uwagę na powyższy fragment kodu napotykamy tu problem popularnie nazywany **„wyciekiem pamięci”**, objawiający się zmniejszeniem wydajności z powodu zwiększonej aktywności procesu zbierania nieużytków lub zwiększonego zużycia pamięci. W skrajnym wypadku wyciek pamięci może doprowadzić do zatrzymania programu z błędem OutOfMemoryError , choć zdarza się to rzadko.

Wyciek pamięci w tym przypadku następuje gdy stos zwiększa się a następnie zmniejsza. Obiekty znajdujące się na stosie nie są niszczone, nawet jeśli program korzystający ze stosu nie przechowuje żadnych referencji(Stos przechowuję niepotrzebne referencje do obiektów, które nigdy nie zostaną wykorzystane – w tym wypadku elementy o indeksach większych niż wartość zmiennej size)

Wyciek pamięci – prawidłowo nazywamy nieświadomym przetrzymywaniem obiektów. Jeżeli referencja do obiektu jest nieświadomie przetrzymywana, to nie tylko obiekt do którego referencja się odwołuje ale i również wszystkie obiekty do których ten obiekt się odwołuje nie będą podlegać procesowi usuwania nieużytków.

Poprawić ten błąd można poprzez przypisanie wartości null do referencji, które przestają być wykorzystane – w tym przypadku zdjęte ze stosu.

public Object pop() {  
 if (size ==0)  
 throw new EmptyStackException();  
 Object result = elements[--size];  
 elements[size] = null; *//usuwanie niepotrzebnej referencji* return result;  
}

Dodatkową zaleta przypisywania wartości null do niepotrzebnych referencji jest to, że jeżeli program przez pomyłkę spróbuje odwołać się do nich odwołać, program zatrzyma się i wygeneruje wyjątek NullPointerException pozwalając szybko wykryć błąd. Przypisywanie wartości null do referencji jest wyjątkiem, a nie normą, najlepszą metodą jest powtórne wykorzystanie zmiennych referencji lub wyjście z zasięgu.

**We wszystkich przypadkach gdy klasa samodzielnie zarządza pamięcią, programista powinien uważać na wycieki pamięci.**

Innym częstym źródłem wycieków pamięci są mechanizmy buforowania lub pamięć podręczna. Kolejnym źródłem wycieków pamięci są nasłuchy i inne metody wywoływania zwrotnego. Wycieki pamięci są zwykle wykrywane podczas bardzo szczegółowej inspekcji kodu lub za pomocą programu wspomagającego uruchamianie, zwanego **profilerem stogu**.