**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**.

***Temat 8. Unikanie finalizatorów oraz oczyszczaczy***

Finalizatory są nieprzewidywalne, często niebedzpieczne i zwykle niepotrzebne. Ich zastosowanie może powodować nieprawidłową pracę i obniżenie wydajności. Wraz z Java 9 finalizatory są wycofywane i zastępuje oczyszczaczami, które choć mniej niebezpieczne niż finalizatory nadal są nieprzywidywalne, powolne i niebezpieczne.

Nie ma gwarancji, że oczyszczacz/finalizator zostanie wykonany natychmiast , dlatego przez finalizator nie powinny być wykonywane żadne krytyczne czasowo operacje( np. zamykanie plików) ani zmeniające ważny stan aplikacji.

Również metody System.gc oraz System.runFinalization nie gwarantują zwiększenia częstosliwości wykonywania finalizatorów.

Jeżeli w czasie finalizacji wystąpi nieobsłużony wyjątek, jest on ignorowany, a finalizacja obiektu zostaje przerwana. Nieobsłużony wyjątek może spowodować powstanie uszkodzonego obiektu.

Użycie finalizatorów i oczyszczacy powoduje znaczny spadek wyddajnośći.

Finalizatory mają poważny problem z bezpieczeństwem – otwierają klasę na ataki finalizacji.

Zamiast finalizatorów, w przypadku klas, które korzystają z zasobów wymagających zwolnienia- takich jak pliki lub wyjątki, należy spowodować by klasa implementowała AutoCloasable i od klientów wymagać wywoływania metody close().

Stosowanie finalizatorów oraz oczyszczaczy sprawdza się w dwóch uzasadnionych przypadkach:

* Używanie ich jako „siatki bezpieczeństwa” na wypadek gdyby właściciel zapomniał wywołać metodę close
* Gdy mamy obiekty z zewnętrznymi obiektami macierzystymi. Obiekt ten odwołuje się bezpośrednio ro składnikow systemu operacyjnego, proces zbierania nieużytków „nie wie” o tym obiekcie i nie może przeprowadzić procesu usuwania obiektu zewnętrznego w trakcie usuwania głównego obiektu.

***Temat 9. Preferuj konstrukcję try z zasobami zamiast try-finally***

Wiele zasobów w bibliotekach Java musi być zamykanych ręcznie przez wywołanie metody *close* np. InputStream,OutPutStream lub java.sql.Connection. Dawniej konstrukcja try-finally była najlepszym sposobem zagwarantowania, że zasób zostanie zamknięty, nawet w przypadku zgłoszenia wyjątku lub szybszego wyjścia z metody. Wszystko zaczyna się jednak komplikować, gdy dochodzi więcej zasobów:

*//Konstrukcja try-finally wygląda brzydko,   
// gdy jest stosowana z więcej niż jednym zasobem*static void copy(String src, String dst) throws IOException {  
 InputStream in = new FileInputStream(src);  
 try{  
 OutputStream out = new FileInputStream(src);  
 try{  
 byte[] buf = new byte[BUFFER\_SIZE];  
 int n;  
 while (( n = in.read(buf)) >= 0)  
 out.write(buf, 0 , n);  
 } finally {  
 out.close();  
 }  
 }finally{  
 in.close();  
 }  
}

Bardzo łatwo pomylić się w takiej sytuacji. W 2007 roku dwie trzecie użyć metody close w bibliotekach Javy było napisanych nieprawidłowo. Dodatkową wadą tego rozwiązania jest to, że jeśli dwie metody zwracają wyjątek, w takiej sytuacji drugi wyjątek całkowicie zaciera pierwszy.

Problemy te udało się rozwiązać wraz z Java 7 za pomocą funkcji try z zasobami. Aby móc z niej korzystać, zasób musi implementować interfejs AutoCloseable.

Poniżej poprawiona wersja – konstrukcja z zasobami:

*//Konstrukcja try z wieloma zasobami*static void copy(String src, String dst) throws IOException {  
 try(InputStream in = new FileInputStream(src);  
 OutputStream out = new FileInputStream(src)) {  
 byte[] buf = new byte[BUFFER\_SIZE];  
 int n;  
 while ((n = in.read(buf)) >= 0)  
 out.write(buf, 0, n);  
 }  
}

Klauzule catch mogą być stosowane dla konstrukcji try z zasobami w taki sam sposób jak dla zwykłych konstrukcji try-finally.

**Metody wspólne dla wszystkich obiektów**

***Temat 10. Zachowanie założeń w trakcie przedefiniowania metody equals***

Nie powinniśmy przesłaniać metody equals gdy:

* Każdy obiekt ze swojej natury jest unikalny
* Nie musisz sprawdzać, czy klasa zapewnia test „logicznej równoważności”
* Klasa bazowa posiada już przesłoniętą metodę equals i jej działanie jest prawidłowe dla danej klasy
* Klasa jest prywatna lub jest prywatna w ramach pakietu i jesteś pewien, że metoda equals nie zostanie nigdy wywołana

Gdy predefiniuje się metodę equals, należy stosować się do ogólnych zasad(metoda equals realizuję relację równoważności):

* Jest zwrotna- dla dowolnej niepustej wartości referencji x, x.equals(y) zwraca true
* Jest symetryczna
* Jest przechodnia – dla dowolnych niepustych x,y,z , gdy x.equals(y) daje true , oraz

y.equals(z) daje true to x.equals(z) też daje true

* Jest spójna – wielokrotnie wywołana daje te same odpowiedzi
* Dla dowolnych niepustych referencji x różnych od null x.equals(null) musi zwrócić false

Relacja równoważności- operator, który dzieli zbiór na podzbiory, których elementy uważa się za równe sobie. Podzbiory nazywamy klasami równoważności.

Uwagi do przechodniości:

Nie ma łatwego sposobu na rozszerzenie definicji klasy o nowe składniki z zachowaniem zasad zdefiniowanych dla metody equals, chyba,że rezygnujemy z zalet abstrakcji obiektowej. W celu obejścia tego problemu należy wykorzystać „Zastępowanie dziedziczenia kompozycją”

Recepta na utworzenie metody equals wysokiej jakości:

1. Skorzystanie z operatora == w celu sprawdzenia, czy argument jest referencją do tego samego obiektu
2. Użyj operatora instaceof do sprawdzenia, czy argument jest odpowiedniego typu
3. Rzutuj argument na właściwy typ
4. Dla każdego „ważnego” pola w klasie sprawdź, czy wartość tego pola odpowiada wartości w obiekcie przekazanym jako argument

***Temat 11. Przedefiniowanie metody hashCode wraz z equals***

**Jeżeli w klasie została przedefiniowana metoda equals, musisz również przedefiniować metodę hashCode. ( bo inaczej łamie drugą z kluczowych wymagań)**

Poniżej założenia/wymagania dla hashCode ze specyfikacji klasy Object:

* Jeżeli meroda hashCode zostanie kilkakrotnie wywołana na rzecz tego samego obiektu, musi ona konsekwentnie zwracać tę samą liczbę, pod warunkiem, że nie zostaną zmodyfikowane, żadne dane wykorzystywane do porównywania obiektów w metodzie equals.
* Jeżeli metoda equals wykarze, że dwa obiekty są jednakowe, to metoda hashCode dla obu tych obiektów musi zwracać tą samą wartość
* Nie jest wymagane, aby dwa różne obiekty, według porównania za pomocą metody equals(Object), zwracały różne wartości metody hashCode. Jednak programista powinien sobie zdawać sprawę, że zwracanie różnych wartości dla różnych obiektów zwiększa szybkość operacji na tablicach mieszających

Przykładowy schemat budowania dobrego hashCode:

1. Zadeklaruj zmienną typu int, na przykład result, i ustaw ją na skrót c dla pierwszego znaczącego pola obiektu zgodnie z punktem 2.a. (Przypomnijmy, że zgodnie z poprzednim tematem pole znaczące to takie, które wpływa na wynik porównywania).

2. Dla każdego pozostałego znaczącego pola f w obiekcie wykonaj następujące czynności:

a) Oblicz wartość kodu mieszającego c typu int dla pola:

i. Jeżeli pole jest typu prostego, oblicz Typ.hashCode (f),

gdzie Typ to klasa osłonowa dla typu f.

ii. Jeżeli pole jest referencją do obiektu i metoda equals porównuje wartość tego pola poprzez rekurencyjne wywołanie metody equals, należy również rekurencyjnie wywołać metodę hashCode dla tego pola. Jeżeli potrzebne jest bardziej skomplikowane porównanie, wylicz,,reprezentację kanoniczną" dla tego pola i wywołaj hashCode dla tej reprezentacji. Jeżeli wartością pola jest null, zwróć 0 (może to być inna stała, ale tradycyjnie zwracamy 0).

iii. Jeżeli pole jest tablicą, potraktuj każdy element jak osobne pole i policz wartość klucza mieszającego dla każdego ważnego elementu, korzystając z przedstawionych zasad. Wyniki należy połączyć w sposób opisany w punkcie 2.b. Jeśli żaden element tablicy nie jest znaczący, użyj stałej, ale najlepiej innej niż 0. Jeżeli każdy element tablicy jest znaczący, użyj metody Array.hashCode .

b) Połącz wszystkie wartości kodów mieszających c z punktu w następujący sposób:

result = 31 \* result + c;

3. Zwróć wartość result.

*// Typowa metoda hashCode*@Override public int hashCode(){  
 int result = Short.*hashCode*(areaCode);  
 result = 31 \* result + Short.*hashCode*(prefix);  
 result = 31 \* result + Short.*hashCode*(lineNum);  
 return result;  
}

Jeżeli klasa jest niezmienna, a koszt obliczenia kodu mieszającego jest znaczny, możesz rozważyć buforowanie kody mieszającego obiektu zamiast obliczania go za każdym razem, gdy jest potrzebny.

@Override public int hashCode() {  
 int result = hashCode;  
 if ( result ==0) {  
 result =Short.*hashCode*(areaCode);  
 result = 31 \* result + Short.*hashCode*(prefix);  
 result = 31 \* result + Short.*hashCode*(lineNum);  
 hashCode = result;  
 }  
 return result;  
}

***Temat 12. Przedefiniowanie metody toString***

Założenia dla metody toString określają, że ciąg ten musi być „zwięzły, zawierający niezbędne dane w postaci czytelnej dla człowieka” Kolejne założenie dla metody toString mówi, że „zaleca się przedefiniowanie tej metody we wszystkich klasach podrzędnych”.

Utworzenie dobrej implementacji toString powoduje, że klasa jest przyjemniejsza w użyciu i czyni ją wygodniejszą w debugowaniu. Zalecane jest, aby metoda toString zwracała wszystkie ważne informacje zawarte w obiekcie , o ile to wykonalne z praktycznego punktu widzenia. Ważną decyzją do podjęcia jest określenie formatu zwracanego ciągu i zapisanie tego formatu w dokumentacji. Zaleca się wykonanie tej czynności dla klas posiadających wartość, na przykład numerów telefonów lub macierzy. Zaletą określenia formatu jest to ,że służy on jako standardowy, jednoznaczny i czytelny dla człowieka opis obiektów. Niezależnie od tego czy definiuje się format czy nie, powinno się jasno udokumentować swoje zamiary. Zawsze należy umożliwić programowy dostęp do wszystkich danych, zawartych w ciągu zwracanym przez metodę toString. Nie ma sensu pisanie metody toString w statycznej klasie użytkowej. Nie należy też jej dodawać do większości typów wyliczeniowych. Przesłoń implementacji metody toString z Object w każdej tworzonej klasie, chyba że zrobiła to już klasa nadrzędna. W ten sposób klasa stanie się bardziej przyjazna w użyciu i prostsza w debugowaniu. Metoda toString powinna zwrócić zwięzły i użyteczny opis obiektu w formacie możliwie najbardziej przyjazny dla oka.

***Temat 13. Rozsądne przedefiniowanie metody clone***

Interfejs Cloneable został zaprojektowany jako interfejs miesznay, dla obiektów, które chcą zaznaczyć, że pozwalają na klonowanie. Posiada on jednak wady. Po pierwsze brak metody clone, a metoda Object.clone jest zabezpieczona. Nie można bez wywoływania refleksji wywołać metody clone, mimo, że klasa implementująca interfejs Cloneable.

Interfejs Cloneable – określa działanie zabezpieczonej metody clone klasy Object. Jeżeli klasa implementuje ten interfejs , metoda clone zwraca kopię obiektu(pole po polu), w przeciwnym wypadku zgłasza wyjątek : CloneNotSupportedException. Choć specyfikacja tego nie wspomina, w praktyce oczekuje się, że klasa implementująca Cloneable zapewni w pełnu funkcjonalną, publiczną metodę clone.

Założenia dla metody clone:

Tworzy i zwraca kopię Obietu. Dokładne znaczenioe „ kopii” może być uzależnione od klasy obiektu. Generalne założenie jest takie, aby dla klasy obiektu x wyrażenie:

x.clone() != x

zwracało wartość true, a wyrażenie:

x.clone().getClass(0 == x.getClass()

również zwracało true, choć nie jest to absolutnie wymagane.

Zwykle wyrażenie

x.clone().equals(x)

zwraca wartość true, ale również nie jest to absolutnie wymagane.

Zgodnie z konwencją obiekt zwracany przez tę metodę powinien zostać pobrany poprzez wywołanie super.clone. Jeżeli klasa i wszystkie jej klasy nadrzędne (poza Object) stosują się do tej konwencji, dojdzie do sytuacji:

x.clone().getClass() == x.getClass()

Zgodnie z konwencją zwrócony obiekt powinien być niezależny od swojego oryginału.

Aby to osiągnąć konieczna może być modyfikacja jednego lub kilku pól obiektu zwróconego przez super.clone.

…

Jeżeli rozszerzasz klasę implementującą Cloneable, nie masz wyjścia musisz utworzyć odpowiednią metodę clone. W przeciwnym wypadku najprawdopodobniej lepiej zastosować inne mechanizmy kopiowania obiektów. Doskonałym sposobem na kopiowanie obiektów jest udostępnienie konstruktora kopiującego lub fabryki kopiującej. Konstruktor kopiujący jest to konstruktor, który ma jeden argument, którego typem jest klasa zawierająca ten konstruktor, na przykład:

*//Konstruktor kopiujący*public Yum( Yum yum);

Innym wariantem tej samej metody jest udostępnienie metody fabrycznej, wywoływanej zamiast konstruktora:

*//Fabryka kopiująca*public static Yum newInstance(Yum yum);

Zastosowanie konstruktora kopiującego lub statycznej metody fabrycznej jest o wiele bardziej korzystne niż stosowanie Cloneable i clone. Konstruktor i metoda fabryczna nie korzystają z ryzykownego, pozajęzykowego mechanizmu tworzenia obiektów, nie wymagają słabo udukomentowanych konwencji, nie są w konflikcie z właściwie wykorzystanymi polami typu final, nie wymagają od progrmaisty przechwytywania niepotrzebnych wyjątków oraz udostępniają klientom ustalone typy.

Konstruktory kopiujące nazywa się konstruktorami konwertującymi, a metody fabryczne, konwertujacymi metodami statycznymi.

Można powiedzieć,że nowe interfejsy nie powinny rozszerzać Cloneable , a nowe klasy zaprojektowane jako klasy bazowe nie powinny go implementować. Jako zasadę należy przyjąć, że funkcjonalność kopiowania powinna być oferowana za pomocą konstruktorów i fabryk. Istostnym wyjątkiem są tu tablice, które najlepiej kopiować za pomocą metody clone.

***Temat 14. Implementacja Interfejsu Comparable***

Interfejs Comparable umożliwia porównywanie obiektów oraz ustalanie ich naturalnego porządku. W przeciwieństwie do metody equals, która służy do testowania równości obiektów, metoda compareTo umożliwia zarówno testowanie równości, jak i określanie porządku między nimi.

Implementując interfejs Comparable, klasa informuje, że jej obiekty posiadają naturalny porządek, co pozwala na sortowanie tablicy obiektów za pomocą jednego wywołania **Arrays.sort(a)**. Możliwe jest także łatwe przeszukiwanie, wyszukiwanie granicznych wartości oraz tworzenie automatycznie sortowanych kolekcji obiektów.

Metoda compareTo określa porządek bieżącego obiektu w porównaniu z podanym obiektem, zwracając ujemną wartość, jeśli obiekt jest mniejszy, zero, gdy jest równy, i dodatnią wartość, gdy jest większy. Istnieją również założenia dotyczące tej metody, które muszą być spełnione, takie jak zapewnienie przemiennych, symetrycznych i przechodnich porównań.

Implementacja metody compareTo jest podobna do implementacji metody equals, ale istnieją pewne różnice, takie jak brak konieczności sprawdzania typu argumentu przed rzutowaniem. Porównania pól mają charakter sprawdzania kolejności, a nie równości. Jeżeli pole nie implementuje interfejsu Comparable lub wymaga niestandardowego porządku, można użyć Comparatora.

Kolejność porównywania pól jest krytyczna, zaczynając od najważniejszego pola i przechodząc do mniej istotnych. Jeżeli porównanie zwraca wartość różną od zera, porównanie jest zakończone, w przeciwnym razie należy przejść do kolejnych pól. Istnieje także technika optymalizacji kodu, która polega na sprawdzeniu różnicy pól i zwróceniu tej wartości, co może przyspieszyć działanie metody.

Jednakże należy używać tej techniki ostrożnie, ponieważ może ona prowadzić do problemów związanych z przepełnieniem. W przypadku dużych różnic między wartościami pól może to skutkować nieprawidłowymi wynikami porównań, co narusza założenia metody compareTo.

Implementując interfejs Comparable, klasa umożliwia wykorzystanie jej w wielu standardowych algorytmach i zbiorach, co jest szczególnie przydatne w sortowaniu i porządkowaniu danych.

**Klasy i interfejsy**

***Temat 15. Ograniczenie dostępności do klas i ich składników***

Właściwe projektowanie modułów oprogramowania polega na ukrywaniu szczegółów implementacji poprzez oddzielenie interfejsu (API) od implementacji. Ukrywanie informacji pozwala na efektywną separację modułów, co ułatwia tworzenie, testowanie, wykorzystywanie i modyfikowanie poszczególnych elementów systemu. Język Java posiada mechanizm kontroli dostępu, który umożliwia określanie poziomu dostępności klas, interfejsów i ich składników. Zasadą jest tworzenie możliwie mało dostępnych klas i składników, korzystając głównie z poziomu dostępu prywatnego w ramach pakietu. W przypadku składników istnieją cztery poziomy dostępu: prywatny, prywatny w ramach pakietu, zabezpieczony i publiczny.

Ważne jest, aby projektując publiczne API, skupić się na zmianie składników na prywatne, ograniczając dostępność do minimum. Testowanie może być wyjątkiem, ale zmniejszenie ograniczeń dostępu powinno być tymczasowe. Pola instancyjne i statyczne nie powinny być publiczne, z wyjątkiem pól final zawierających stałe, które są integralną częścią abstrakcji klasy. Warto zauważyć, że publiczne pola tablicowe mogą stanowić ryzyko dla bezpieczeństwa, dlatego zaleca się unikanie ich używania poprzez zastosowanie metod dostępowych lub kolekcji niemodyfikowalnych.

Podsumowując, należy dążyć do jak największego ograniczenia dostępności klas i ich składników, dbając jednocześnie o odpowiednie zaprojektowanie publicznego API oraz unikanie publicznych pól z wyjątkiem stałych.

.

***Temat 16. Stosowanie metod akcesoriów zamiast poł publicznych w klasach publicznych***

Klasy publiczne powinny unikać bezpośredniego udostępniania pól, ponieważ narusza to zasadę hermetyzacji. Zamiast tego, warto stosować metody dostępowe (akcesory) i modyfikujące (mutatory), które pozwalają na kontrolowany dostęp do pól klasy. Jednakże w przypadku klas prywatnych lub zagnieżdżonych, udostępnianie pól może być akceptowalne, pod warunkiem, że abstrakcja klasy jest odpowiednio zachowana.

***Temat 17. Zapewnianie niezmienności obiektu***

Klasa niezmienna to klasa, której obiekty nie mogą być modyfikowane po ich utworzeniu. Wszystkie dane zawarte w obiekcie klasy niezmiennych są ustalane w momencie tworzenia obiektu i pozostają niezmienne przez cały okres istnienia tego obiektu. W języku Java istnieje wiele wbudowanych klas niezmiennych, takich jak **String**, klasy otoczkowe dla typów prymitywnych oraz **BigInteger** i **BigDecimal**. Istnieje kilka powodów, dla których stosuje się obiekty niezmienne:

1. **Prostota**: Klasy niezmiennych są łatwiejsze w projektowaniu, tworzeniu i używaniu w porównaniu z klasami modyfikowalnymi. Zapewniają one klarowność i eliminują niektóre potencjalne błędy.
2. **Bezpieczeństwo**: Obiekty niezmienne są bezpieczne w kontekście wielowątkowości, ponieważ nie można ich przypadkowo zmienić z innego wątku.

Aby zapewnić niezmienność obiektu, istnieje kilka zasad:

1. **Brak metod modyfikujących**: Klasa nie powinna zawierać metod, które mogłyby zmieniać stan obiektu po jego utworzeniu.
2. **Oznaczenie klasy jako finalnej**: Oznaczenie klasy jako **final** uniemożliwia dziedziczenie po niej, co zapobiega modyfikacjom w klasach dziedziczących.
3. **Deklarowanie wszystkich pól jako finalnych**: Wszystkie pola klasy powinny być zadeklarowane jako **final**, aby uniemożliwić ich modyfikację.
4. **Deklarowanie wszystkich pól jako prywatnych**: Uniemożliwia to bezpośrednią modyfikację pól przez klientów klasy.
5. **Zapewnienie wyłącznego dostępu do modyfikowalnych komponentów**: Jeśli klasa zawiera pola odwołujące się do modyfikowalnych obiektów, należy zapewnić, że klienci klasy nie mogą odczytać tych referencji ani ich modyfikować.

Klasy niezmienne mają wiele zalet, ale mogą być mniej wydajne w niektórych przypadkach, zwłaszcza gdy operacje na nich wymagają tworzenia wielu nowych obiektów. W takich sytuacjach należy rozważyć inne strategie optymalizacji lub zastosowanie elastyczniejszych podejść, takich jak przekazywanie modyfikowalnych klas pomocniczych.

***Temat 18. Zastępowanie dziedziczenia kompozycją***

Dziedziczenie to potężne narzędzie, które ułatwia ponowne wykorzystanie kodu, ale może prowadzić do powstania kodu podatnego na błędy, gdy nie jest stosowane odpowiednio. Bezpiecznie można używać dziedziczenia w obrębie tego samego pakietu, gdzie implementacja klasy bazowej i pochodnej jest tworzona przez tego samego programistę. Bezpiecznie również można korzystać z dziedziczenia klas, które zostały zaprojektowane i udokumentowane z myślą o takim wykorzystaniu. Jednakże dziedziczenie z klasy z innego pakietu może być ryzykowne.

W przeciwieństwie do dziedziczenia interfejsu, gdzie klasa implementuje interfejs lub interfejs dziedziczy po innym interfejsie, dziedziczenie implementacji narusza hermetyzację. Klasy pochodne muszą polegać na szczegółach implementacji klasy bazowej, co oznacza, że zmiany w klasie bazowej mogą prowadzić do problemów w klasach pochodnych.

Jednym z podejść do rozwiązania tego problemu jest zastosowanie kompozycji. Zamiast dziedziczyć po istniejącej klasie, nowa klasa może zawierać prywatne pole, które odwołuje się do obiektu istniejącej klasy. Każda metoda w nowej klasie może wywoływać odpowiednie metody klasy zawartej i zwracać ich wyniki. Jest to znane jako przekazywanie, a metody w nowej klasie nazywane są metodami przekazującymi.

Korzyści z tego podejścia są znaczne. Klasy oparte na kompozycji są bardziej niezawodne, ponieważ nie są zależne od szczegółów implementacji innej klasy. Nawet zmiany w klasie bazowej nie mają wpływu na nową klasę.