Java. Efektywne programowanie. Wydanie II

Autor: Joshua Bloch
Tłumaczenie: Paweł Gonera
ISBN: 978-83-246-2084-5
Tytuł oryginału: Effective Java (2nd Edition)
Format: 158x235 , stron: 352

Poznaj specyfikę języka Java i zostań mistrzem programowania

- Jak korzystać z bibliotek języka Java?
- Jak pisać funkcjonalny i klarowny kod?
- Jak stworzyć profesjonalny i efektowny program?

Język Java jest językiem obiektowym z dziedziczeniem jednobazowym. Wewnątrz każdej metody korzysta on ze zorientowanego na instrukcje stylu kodowania. Aby dobrze poznać jakikolwiek język, należy nauczyć się posługiwać jego regułami, zasadami i składnią – podobnie jest z językiem programowania. Jeśli chcesz zyskać możliwość efektywnego programowania w języku Java, powinieneś poznać struktury danych, operacje i udogodnienia, oferowane przez biblioteki standardowe, a także często stosowane i efektywne sposoby tworzenia kodu. Całą potrzebną Ci wiedzę znajdziesz właśnie w tym podręczniku.

W książce „Java. Efektywne programowanie" w sposób zrozumiały i klarowny przedstawiono zasady opisujące mechanizmy używane w najlepszych technikach programowania. Ten podręcznik podpowie Ci, jak najbardziej racjonalnie korzystać z języka Java oraz jego podstawowych bibliotek. Dowiesz się, jak stosować wyjątki przechwytywalne i wyjątki czasu wykonania, poznasz także zalety stosowania statycznych klas składowych. Opanujesz metody sprawdzania poprawności parametrów i projektowania sygnatur oraz wszelkie instrukcje, które pozwolą Ci na wydajne i profesjonalne programowanie.

- Tworzenie i usuwanie obiektów
- Klasy i interfejsy
- Zapewnianie niezmienności obiektu
- Projektowanie i dokumentowanie klas przeznaczonych do dziedziczenia
- Zalety stosowania statycznych klas składowych
- Typy ogólne
- Typy wyliczeniowe i adnotacje
- Metody
- Programowanie
- Wykorzystanie ogólnie przyjętych konwencji nazewnictwa
- Wyjątki
- Współbieżność i serializacja
- Dokumentowanie bezpieczeństwa dla wątków

Nie wystarczy samo poznanie języka Java.
Trzeba wiedzieć, jak z niego efektywnie korzystać!

Spis treści

9

Wyjątki

Korzystając z najlepszych cech wyjątków, można dzięki nim poprawić czytelność programu, jego solidność i łatwość konserwacji. Użyte niewłaściwie będą miały odwrotny efekt. W rozdziale tym zamieszczamy wskazówki, pomagające efektywnie wykorzystywać wyjątki.

Temat 57. Wykorzystanie wyjątków w sytuacjach nadzwyczajnych

Być może kiedyś spotkasz się z fragmentem kodu, który będzie wyglądał następująco:

```
// Nadużycie wyjątków. Nigdy tak nie rób!
try {
  int i = 0 ;
  while (true)
    range[i++].climb();
} catch (ArrayIndexOutOfBoundsException e) {
}
```

Co robi ten fragment? Działanie to nie jest oczywiste na pierwszy rzut oka i jest to wystarczający powód, aby odradzić stosowanie takiego stylu programowania (temat 55.). Jest to bardzo zły sposób na przeglądanie tablicy. Pętla nieskończona jest przerywana przez zgłoszenie, przechwycenie i zignorowanie wyjątku `ArrayIndex`
`↪OutOfBoundsException`, gdy program będzie próbował odwołać się do pierwszego elementu spoza zakresu tablicy. Jest to koślawy odpowiednik standardowej metody przeglądania tablicy, natychmiast rozpoznawany przez każdego programistę:

```
for (Mountain m : range)
  m.climb();
```

Skąd wziął się więc pomysł zastąpienia wypróbowanej metody przeglądania tablicy metodą wykorzystującą wyjątki? Jest to nieudolna próba poprawy wydajności, oparta o nieprawidłowe założenie, że maszyna wirtualna kontroluje dostęp do elementów spoza zakresu tablicy przy wszystkich odwołaniach, więc zwykły test zakończenia pętli — ukryty przez kompilator, ale nadal występujących w pętli `for-each` — jest nadmiarowy i powinien być usunięty. Założenie to powoduje wystąpienie trzech niekorzystnych zjawisk:

- Ponieważ wyjątki są zaprojektowane do obsługi sytuacji wyjątkowych, implementacje JVM słabo optymalizują kod obsługi wyjątku. Tworzenie, zgłaszanie i przechwytywanie wyjątków to operacje stosunkowo kosztowne.
- Umieszczenie kodu w bloku `try-catch` wyklucza niektóre optymalizacje, które nowoczesne maszyny wirtualne mogą zastosować.
- Standardowa metoda przeglądania elementów tablicy niekoniecznie musi wykonywać nadmiarowe sprawdzenia — nowoczesne implementacje JVM potrafią to zoptymalizować.

W rzeczywistości zaprezentowana metoda przeglądania tablicy z wykorzystaniem wyjątku na niemal wszystkich implementacjach JVM jest o wiele wolniejsza, niż metoda najczęściej wykorzystywana. Na moim komputerze kod, przeglądający stuelementową tablicę metodą wykorzystującą wyjątki, wykonywał się dwa razy dłużej niż fragment korzystający ze standardowej metody.

Metoda wykorzystująca wyjątki nie tylko zaciemnia kod i spowalnia program, ale również nie gwarantuje poprawnej pracy. W przypadku wystąpienia całkowicie innego błędu metoda ta może zawieść bez żadnego ostrzeżenia, maskując błąd, co ogromnie skomplikuje proces uruchamiania programu. Załóżmy, że obliczenia wykonywane w pętli zawierają błąd przekroczenia zakresu całkowicie innej tablicy. Jeżeli zostałaby wykorzystana standardowa metoda przeglądania tablicy, błąd ten spowodowałby zgłoszenie wyjątku, co z kolei natychmiast zakończyłoby wątek wygenerowaniem odpowiedniego komunikatu błędu. Jeżeli jednak zostanie użyta metoda z przechwytywaniem wyjątku, to wyjątek związany z błędem zostanie mylnie zinterpretowany jako normalne zakończenie pętli.

Przykład ten ilustruje, że **wyjątki, tak jak ich nazwa wskazuje, powinny być wykorzystywane do obsługi sytuacji wyjątkowych — nie powinny być nigdy używane do sterowania normalnym przebiegiem kodu programu**. Mówiąc ogólniej, powinieneś stosować standardowe, rozpoznawalne konstrukcje programowe zamiast sprytnych, stosowanych jedynie w celu osiągnięcia lepszej wydajności. Nawet, gdy zysk wydajności jest znaczny, może szybko zostać zniwelowany w kolejnych wersjach szybko rozwijających się implementacji JVM. Pozostaną jednak kłopoty przy usuwaniu subtelnych błędów, występujące w przypadku zastosowania sprytnych metod.

Zasada ta powinna być również stosowana przy projektowaniu API. **Dobrze zaprojektowane API nie powinno wymuszać na klientach wykorzystania wyjątków do sterowania zwykłym przebiegiem programu.** Klasa z metodami „zależnymi od stanu", które są wywoływane jedynie w określonych, nieprzewidywalnych warunkach, powinny posiadać zwykle oddzielne metody „testujące stan", czyli sprawdzające, czy należy wywołać pierwszą metodę. Na przykład, klasa `Iterator` posiada zależną od stanu metodę `next`, która zwraca następny element wyliczenia oraz skojarzoną z nią metodę `hasNext`, testującą stan. Pozwala to na skorzystanie ze standardowego idiomu, służącego do przeglądania kolekcji w standardowych pętlach `for` (jak również w pętlach `for-each`, w których metoda `hasNext` jest wykorzystywana wewnętrznie):

```
for (Iterator<Foo> i = collection.iterator(); i.hasNext(); ) {
  Foo foo = i.next();
  // ...
}
```

Jeżeli klasa `Iterator` nie posiadałaby metody `hasNext`, klient zostałby zmuszony do skorzystania z poniższej metody:

```
// Nie korzystaj z tej okropnej metody przeglądania kolekcji!
try {
  Iterator<Foo> i = collection.iterator();
  while (true) {
    Foo foo = i.next();
    // ...
  }
} catch (NoSuchElementException e) {
}
```

Przedstawiony kod wygląda podobnie do przykładu rozpoczynającego ten temat. Oprócz tego, że przykład jest rozwlekły, działa dużo wolniej niż standardowy oraz może ukrywać błędy, pochodzące z innych części systemu.

Innym sposobem zapewnienia osobnej metody testującej stan jest zwracanie przez metodę zależną od stanu specjalnej wartości, jak na przykład `null`, jeżeli obiekt jest w nieprawidłowym stanie. Technika ta nie nadaje się dla klasy `Iterator`, ponieważ wartość `null` jest prawidłową wartością zwracaną przez metodę `next`.

Warto wymienić kilka sugestii, dotyczących wyboru pomiędzy metodą testującą stan i zwracaniem wartości specjalnej. Jeżeli do obiektu można się odwoływać równolegle bez zewnętrznej synchronizacji lub jego stan zmienia się pod wpływem zewnętrznych czynników, wykorzystanie specjalnej wartości może być kluczowe dla prawidłowego działania, ponieważ obiekt może zmienić swój stan pomiędzy wywołaniem metody testującej stan i odpowiadającej jej metody zależnej od stanu. Jeżeli ważna jest wydajność, należy skorzystać ze zwracania wartości specjalnej (jeżeli wykonanie osobnej metody testującej stan wiąże się z wykonaniem tych

samych operacji, co w odpowiadającej jej metodzie zależnej od stanu). Jeżeli nie zachodzą te przypadki, zalecane jest tworzenie metody testującej stan. Pozwala ona osiągnąć lepszą czytelność kodu, a w przypadku jej niewłaściwego stosowania łatwiej można odszukać i poprawić błąd.

Podsumujmy. Wyjątki są zaprojektowane do wykorzystania w sytuacjach wyjątkowych. Nie należy ich używać do zwykłej kontroli przepływu sterowania i nie należy pisać API wymuszających takie działanie.

Temat 58. Stosowanie wyjątków przechwytywanych i wyjątków czasu wykonania

Język Java pozwala na stosowanie trzech rodzajów wyjątków: *wyjątków przechwytywanych*, *wyjątków czasu wykonania* i *błędów*. Często pojawiają się wątpliwości, dotyczące zastosowania odpowiedniego rodzaju wyjątku. Wybór nie zawsze jest jednoznaczny, ale istnieją ogólne zasady, ułatwiające podjęcie decyzji.

Podstawową zasadą jest określenie, czy wyjątek jest przechwytywany czy nieprzechwytywany. **Wyjątki przechwytywane należy stosować w przypadkach, w których spodziewamy się, że wywołujący może przywrócić normalny stan.** Zgłaszając wyjątek przechwytywany, zmuszasz wywołującego do obsłużenia tego wyjątku w klauzuli `catch` lub do jego propagacji na wyższy poziom. Każdy przechwytywany wyjątek deklarowany przez metodę jest wskazówką dla użytkownika API, że w trakcie wykonywania metody można spodziewać się, że taki będzie wynik wykonania danej metody.

Przekazując przechwytywany wyjątek użytkownikowi API, projektant nakazuje wykonanie operacji przywracających normalny stan systemu. Użytkownik może zgodzić się na ten nakaz, przechwytując wyjątek, lub zignorować go, choć nie jest to najlepszym pomysłem (temat 65.).

Istnieją dwa rodzaje wyjątków nieprzechwytywanych — wyjątki czasu wykonania i błędy. Ich działanie jest identyczne — oba rodzaje nie mogą i zwykle nie powinny być przechwytywane. Jeżeli program zgłasza wyjątek nieprzechwytywany lub błąd, zwykle przywrócenie właściwego stanu systemu jest niemożliwe i kontynuowanie działania programu może spowodować więcej złego niż dobrego. Jeżeli program nie przechwyci takiego wyjątku, spowoduje to zakończenie bieżącego wątku z odpowiednim komunikatem błędu.

Do wskazywania błędów programistycznych należy używać wyjątków czasu wykonania. Większość wyjątków czasu wykonania wskazuje na *naruszenia założeń*. Naruszenie założenia występuje, gdy klient API nie może dotrzymać założeń, wymienionych w specyfikacji API. Dla przykładu, założenia dla dostępu do tablicy określają, że indeks tablicy musi być liczbą od zera do wielkości tablicy minus jeden. Wyjątek `ArrayIndexOutOfBoundsException` wskazuje na niedotrzymanie tego założenia.

Mimo że specyfikacja języka Java nie wymusza tego, istnieje bardzo silna konwencja, mówiąca, że błędy są zarezerwowane przez maszynę wirtualną do wskazywania braku zasobów, naruszenia niezmienników lub innych błędów, powodujących niemożliwość kontynuacji wykonania programu. Z powodu niemal całkowitej akceptacji tej konwencji najlepiej nie tworzyć nowych podklas dla klasy `Error`. **Wszystkie nieprzechwytywane wyjątki definiowane przez Ciebie powinny dziedziczyć po `RuntimeException`** (bezpośrednio bądź pośrednio).

Możliwe jest zdefiniowanie wyjątku, który nie dziedziczy po klasie `Exception`, `Run`
`↪TimeException` lub `Error`. Specyfikacja języka nie zajmuje się bezpośrednio takimi wyjątkami, ale niejawnie określa, że mają one identyczne działanie, jak zwykłe, przechwytywane wyjątki (które dziedziczą po `Exception`, a nie `RunTimeException`). Kiedy więc powinniśmy z nich skorzystać? Prawdopodobnie nigdy. Nie mają one żadnych zalet w porównaniu ze zwykłymi przechwytywanymi wyjątkami i prawdopodobnie będą jedynie myliły użytkowników Twojego API.

Podsumujmy. Wyjątków przechwytywanych należy używać do obsługi przypadków, w których można naprawić błąd, a wyjątków nieprzechwytywanych należy używać do wykonania obsługi błędów programowych. Oczywiście sytuacja nie zawsze musi być czarno-biała. Na przykład wyczerpanie się zasobów może być spowodowane przez błąd programu, taki jak niepotrzebne przydzielenie zbyt dużej tablicy, albo przez zwykły brak zasobów. Jeżeli brak zasobów jest spowodowany przez tymczasowy niedobór lub przez tymczasowo zwiększone potrzeby, operacja ma szansę powodzenia. W gestii projektanta API leży decyzja, czy dana sytuacja braku zasobów pozwala na przywrócenie normalnego stanu. Jeżeli uważasz, że sytuację da się naprawić, należy skorzystać z wyjątku przechwytywanego — jeżeli nie, wyjątku czasu wykonania. Jeżeli nie jesteś pewien, czy możliwe jest przywrócenie normalnego stanu, z powodów omówionych w temacie 59. lepiej skorzystać z wyjątku nieprzechwytywanego.

Projektanci API często zapominają, że wyjątki są w pełni funkcjonalnymi obiektami, w których mogą być definiowane normalne metody. Podstawowym zastosowaniem takich metod jest dostarczenie kodu przechwytującego wyjątek i dodatkowych informacji na temat zdarzenia, które spowodowało zgłoszenie wyjątku. W przypadku braku takich metod programiści muszą analizować ciąg zwracany przez wyjątek w celu wyłuskania dodatkowych danych. Jest to bardzo zła praktyka

(temat 10.). Klasy rzadko określają szczegóły ich reprezentacji jako ciąg znaków. Reprezentacja ta może różnić się w różnych implementacjach i wersjach. Dlatego kod analizujący ciąg reprezentujący wyjątek najprawdopodobniej będzie nieprzenośny i podatny na błędy.

Ponieważ wyjątki przechwytywane najczęściej wskazują sytuacje naprawialne, szczególnie ważne jest definiowanie w klasach wyjątków metod, które dostarczają danych pomagających przywrócić prawidłowy stan. Na przykład, przechwytywany wyjątek może być zgłaszany w sytuacji, gdy wykonana została próba zadzwonienia z automatu telefonicznego bez wrzucenia odpowiedniej kwoty. Wyjątek ten powinien zawierać metodę zwracającą brakującą kwotę, dzięki czemu niedobór może być wyświetlony użytkownikowi telefonu.

Temat 59. Unikanie niepotrzebnych wyjątków przechwytywanych

Przechwytywane wyjątki są niezwykle użytecznym mechanizmem języka Java. W przeciwieństwie do zwracanych wartości *wymuszają* one na programiście zajęcie się sytuacją wyjątkową, co niezmiernie zwiększa solidność aplikacji. Jednak nadużywanie wyjątków może spowodować, że API będzie mało wygodne. Jeżeli metoda zgłasza jeden lub więcej wyjątków, kod wywołujący ją musi obsługiwać te wyjątki w jednym lub większej liczby bloków `catch` albo musi zadeklarować, że będzie zgłaszał te wyjątki, co umożliwi ich propagację. Obie czynności powodują zwiększone obciążenie programistów.

Obciążenie to jest uzasadnione, gdy nie można uniknąć sytuacji wyjątkowych przy właściwym stosowaniu API *oraz* gdy programista wykorzystujący to API ma możliwość przedsięwziąć sensowne akcje po wystąpieniu takiego wyjątku. W sytuacji, gdy nie są spełnione oba te warunki, bardziej właściwe jest zastosowanie wyjątku nieprzechwytywanego. Musimy odpowiedzieć sobie na pytanie, w jaki sposób programista ma obsłużyć wyjątek. Czy jest to najlepsze, co może być wykonane?

```
} catch(TheCheckedException e) {
    throw new AssertionError();  // Nie powinien się zdarzyć!
}
```

I drugi przypadek.

```
} catch (TheCheckedException e) {
    e.printStackTrace();              // To już koniec.
    System.exit(1);
}
```

Jeżeli programista nie może lepiej obsłużyć wyjątku, lepszy będzie wyjątek nieprzechwytywany. Przykładem wyjątku, który nie spełnia tej zasady, jest `CloneNot ↪SupportedException`. Jest on zgłaszany przez metodę `Object.clone`, która powinna być wywoływana jedynie przez obiekt implementujący interfejs `Cloneable` (temat 11.). W praktyce blok `catch` nieomal zawsze ma charakter niepowodzenia asercji. Przechwycenie tego wyjątku nie daje programiście żadnych dodatkowych możliwości, wymaga jedynie dodatkowej pracy i komplikuje program.

Dodatkowo, obciążenie programisty przez obsługę wyjątku jest nieco większe, jeżeli metoda zgłasza *jeden* wyjątek. Jeżeli zgłaszane jest więcej wyjątków, metoda musi być umieszczona w bloku `try`, a obsługa poszczególnych wyjątków jest umieszczona w kolejnych blokach `catch`. Jeżeli metoda zgłasza tylko jeden wyjątek, jest on odpowiedzialny za umieszczenie wywołania tej metody w bloku `try`. W tej sytuacji opłaca się pomyśleć o sposobach uniknięcia przechwytywania wyjątków.

Jedną z technik zmiany wyjątku przechwytywanego na nieprzechwytywany jest podział metody na dwie części — pierwsza zwraca wartość `boolean`, wskazującą na to, czy powinien być zgłoszony wyjątek. Stosując tę metodę powodujemy, że poniższa sekwencja wywołań:

```
// Wywołanie ze sprawdzaniem wyjątków
try {
    obj.action(args);
} catch (TheCheckedException e) {
    // Obsługa sytuacji wyjątkowej
    // ...
}
```

zmienia się na:

```
// Wywołanie z użyciem metody testującej stan i wyjątku nieprzechwytywanego
if (obj.actionPermitted(args)) {
    obj.action(args);
} Else {
    // Obsługa sytuacji wyjątkowej
    // ...
}
```

Transformacja taka nie jest zawsze prawidłowa, ale tam, gdzie można ją zastosować, może ona ułatwić używanie API. Choć druga sekwencja wywołań nie jest ładniejsza od pierwszej, to jednak API jest bardziej elastyczne. W przypadkach, gdy programista wie, że wywołanie na pewno się uda lub pozwala na zakończenie programu w przypadku niepowodzenia, pokazana transformacja pozwala na zastosowanie najprostszej sekwencji wywołań:

```
obj.action(args);
```

Jeżeli podejrzewasz, że najprostsza sekwencja wywołania będzie powszechnie stosowana, to taka modyfikacja API jest właściwa. Po takiej modyfikacji API będzie nieomal identyczne, jak w przypadku zastosowania metody testującej stan, przedstawionej w temacie 57. i należy tu przestrzegać tych samych zasad. Jeżeli do obiektu występują jednoczesne odwołania bez zewnętrznej synchronizacji lub jego stan zmienia się pod wpływem czynników zewnętrznych, przedstawiona modyfikacja jest nieprawidłowa, ponieważ stan obiektu może zmienić się pomiędzy wywołaniem metody `actionPermitted` i `action`. Jeżeli metoda `actionPermitted` będzie musiała powtórzyć operacje wykonywane w metodzie `action`, transformacja ta spowoduje obniżenie wydajności systemu.

Temat 60. Wykorzystanie wyjątków standardowych

Jednym z atrybutów najlepiej odróżniających ekspertów w dziedzinie programowania od mniej zaawansowanych programistów jest dbałość ekspertów o wysoki stopień ponownego wykorzystania kodu. Wyjątki nie są tutaj wyjątkiem od zasady powtórnego wykorzystania kodu. Biblioteki platformy Java zawierają podstawowy zbiór wyjątków nieprzechwytywanych, który obejmuje większość sytuacji wyjątkowych spotykanych w API. Temat ten zawiera omówienie tych powszechnie wykorzystywanych wyjątków.

Ponowne wykorzystanie istniejących wyjątków niesie ze sobą wiele korzyści. Najważniejszą z nich jest ułatwienie nauki stosowania API, ponieważ korzysta ono z dobrze ugruntowanych konwencji, z którymi programista jest już zaznajomiony. Nie mniej ważną korzyścią jest czytelność programów korzystających z takiego API, ponieważ kod nie jest zabałaganiony nieznanymi wyjątkami. Na koniec, mniej klas wyjątków powoduje zmniejszenie ilości zużywanej pamięci i skraca czas ładowania klas.

Najczęściej wykorzystywanym wyjątkiem jest `IllegalArgumentException`. Najczęściej jest on zgłaszany, gdy wywołujący przekaże argument o niewłaściwej wartości. Na przykład, można skorzystać z tego wyjątku, jeżeli jako parametr reprezentujący liczbę powtórzeń zostanie przekazana wartość ujemna.

Innym często stosowanym wyjątkiem jest `IllegalStateException`. Jest to wyjątek zgłaszany, jeżeli w aktualnym stanie obiektu wywołanie jest nielegalne. Na przykład, jeżeli wywołujący będzie próbował skorzystać z obiektu przed jego prawidłowym zainicjowaniem.

Można uznać, że wszystkie błędne wywołania metod sprowadzają się do nieprawidłowego argumentu i do niewłaściwego stanu obiektu, ale do sygnalizowania niektórych rodzajów nieprawidłowych stanów i argumentów wykorzystywane są najczęściej specjalizowane wyjątki. Jeżeli wartością parametru będzie `null` w miejscu, gdzie wartości `null` są niedozwolone, konwencja nakazuje zgłosić wyjątek `NullPointer↪Exception` a nie `IllegalArgumentException`. Podobnie, jeżeli przekazana wartość parametru oznaczającego indeks wykracza poza dozwolony zakres, należy zgłosić wyjątek `IndexOutOfBoundsException`, a nie `IllegalArgumentException`.

Innym wyjątkiem ogólnego przeznaczenia, jaki powinniśmy znać, jest `Concurrent↪ModificationException`. Wyjątek ten powinien być zgłaszany, jeżeli obiekt zaprojektowany do wykorzystywania przez jeden wątek lub korzystający z zewnętrznej synchronizacji wykryje, że jest (lub był) równolegle modyfikowany.

Ostatnim wyjątkiem ogólnego przeznaczenia, jaki tu omówimy, jest `Unsupported↪OperationException`. Wyjątek ten jest zgłaszany, gdy obiekt nie obsługuje wywoływanej operacji. Jest on dosyć rzadko wykorzystywany, ponieważ większość obiektów obsługuje wszystkie implementowane przez siebie metody. Wykorzystywany jest w sytuacjach, gdy implementacja interfejsu nie posiada obsługi jakiejś opcjonalnej operacji definiowanej przez interfejs. Na przykład, implementacja interfejsu `List`, przeznaczona tylko do dołączania elementów, zgłosi ten wyjątek, gdy ktoś będzie próbował usunąć element.

W tabeli 7.1 przedstawione jest zestawienie często wykorzystywanych wyjątków.

Tabela 7.1. *Często wykorzystywane wyjątki*

Wyjątek	Zastosowanie
`IllegalArgumentException`	Nieprawidłowa wartość parametru
`IllegalStateException`	Nieprawidłowy stan obiektu dla wywoływanej metody
`NullPointerException`	Zabronione stosowanie wartości `null` w parametrze
`IndexOutOfBoundsException`	Wartość indeksu poza zakresem
`ConcurrentModificationException`	Wykryta równoległa modyfikacja stanu obiektu w przypadku, gdy jest ona zabroniona
`UnsupportedOperationException`	Obiekt nie obsługuje metody

Choć są to najczęściej wykorzystywane wyjątki w bibliotekach platformy Java, to w przypadkach, gdy sytuacja upoważnia do użycia innego wyjątku, należy go zastosować. Na przykład, w przypadku tworzenia obiektów matematycznych, takich jak liczby zespolone lub macierze, bardziej właściwe będzie zastosowanie wyjątków `ArihtmeticException` i `NumberFormatException`. Jeżeli wyjątek spełnia Twoje potrzeby — wykorzystaj go, jednak jedynie wtedy, gdy warunki, w których zgłaszasz

ten wyjątek, są zgodne z warunkami opisanymi w jego dokumentacji. Wykorzystanie wyjątków musi być uzasadnione semantyką, a nie jedynie nazwą. W przypadkach, gdy chcesz uzupełnić wyjątek o dodatkowe informacje, możesz również dziedziczyć po istniejących wyjątkach (temat 63.).

Musisz wiedzieć, że wybór właściwego wyjątku do wykorzystania nie jest czasami jednoznaczny, a „Zastosowania" z tabeli 7.1 nie zawsze wzajemnie się wykluczają. Pomyśl na przykład o przypadku obiektu reprezentującego talię kart. Załóżmy, że mamy metodę wydającą karty, która jako argument posiada liczbę kart. Załóżmy, że wywołujący tę metodę przekazał wartość parametru większą niż liczba kart pozostałych w talii. Może być to zinterpretowane jako `IllegalArgumentException` (wartość parametru jest zbyt duża) lub `IllegalStateException` (obiekt *talia* posiada za mało kart do zrealizowania żądania). W tym przypadku wydaje się, że właściwy będzie wyjątek `IllegalArgumentException`, ale nie ma tu żadnej sztywnej reguły.

Temat 61. Zgłaszanie wyjątków właściwych dla abstrakcji

Niewłaściwe jest zgłaszanie przez metodę wyjątku, który nie ma widocznego związku z zadaniem, jakie ta metoda wykonuje. Często się to zdarza, gdy metoda propaguje wyjątek zgłoszony przez mechanizmy niskiego poziomu. Oprócz wprowadzania zamieszania powoduje to zaśmiecenie API wyższej warstwy szczegółami implementacji warstwy niższej. Jeżeli w przyszłej wersji implementacja wyższej warstwy zostanie zmieniona, zgłaszane wyjątki również mogą się zmienić, co może spowodować, że programy klientów przestaną działać.

Aby uniknąć tego problemu, **wyższe warstwy abstrakcji powinny przechwytywać wyjątki z warstwy niższej i w ich miejsce zgłaszać wyjątki, które są odpowiednie dla wyższej warstwy**. Idiom ten, nazywany *translacją wyjątków*, wygląda następująco:

```
// Translacja wyjątków
try {
    // Wykorzystanie operacji niższej warstwy
    // ...
} catch (LowerLevelException e) {
    throw new HigherLevelException(...);
}
```

Poniżej przedstawiamy przykład translacji wyjątków, zaczerpnięty z klasy `AbstractSequentialList`, która jest *szkieletową implementacją* (temat 18.) interfejsu `List`. W przykładzie tym wykorzystanie translacji wyjątków jest wymuszone przez specyfikację metody `get` z interfejsu `List<E>`.

```
/**
 * Zwraca element z podanej pozycji w tej liście.
 * @throws IndexOutOfBoundsException, jeżeli indeks jest poza zakresem
 *       (index < 0 || index >= size()).
 */
public E get(int index) {
  ListIterator<E> i = listIterator(index);
  try {
      return i.next();
  } catch(NoSuchElementException e) {
      throw new IndexOutOfBoundsException("Indeks: " + index);
  }
}
```

W przypadku, gdy wyjątki niższego poziomu mogą być przydatne w procesie szukania przyczyny wyjątku, można zastosować odmianę translacji wyjątków, nazywaną *łączeniem wyjątków*. W przypadku tego podejścia wyjątek niższego poziomu (*przyczyna*) jest zapamiętywany przez wyjątek wyższego poziomu, który zapewnia publiczną metodę (`Throwable.getCause`), umożliwiającą pobranie wyjątku niższego poziomu.

```
// Łączenie wyjątków
try {
    // Wykorzystanie operacji niższej warstwy
    // ...
} catch (LowerLevelException cause) {
    throw new HigherLevelException(cause);
}
```

Konstruktor wyjątku wyższego poziomu przekazuje przyczynę do konstruktora klasy bazowej *obsługującego łączenie*, więc jest ostatecznie przekazywany do jednego z konstruktorów `Throwable` obsługujących łączenie, takiego jak `Throwable ↪(Throwable)`:

```
// Wyjątki z konstruktorem obsługującym łączenie
class HigherLevelException extends Exception {
    HigherLevelException(Throwable cause) {
        super(t);
    }
}
```

Większość ze standardowych wyjątków posiada konstruktor obsługujący łączenie. Dla wyjątków, które ich nie mają, można ustawić przyczynę przy użyciu metody `initCause` z `Throwable`. Łączenie wyjątków nie tylko pozwala nam na programowy dostęp do przyczyny (za pomocą `getCause`), ale również integruje zapis śladu przyczyny z wyjątkiem wyższego poziomu.

Ponieważ translacja wyjątków doskonale nadaje się do bezmyślnej propagacji wyjątków między warstwami aplikacji, nie powinna być nadużywana. Tam, gdzie jest to możliwe, najlepszą metodą obsługi wyjątków pochodzących z niższych warstw

jest ich unikanie, przy upewnieniu się przed wywołaniem metody niższej warstwy, że zostanie ona prawidłowo wykonana. Czasami można to zrealizować przez sprawdzenie poprawności argumentów metody wyższej warstwy przed ich przekazaniem do metody niższej warstwy.

Jeżeli niemożliwe jest zabezpieczenie przed zgłaszaniem wyjątków z niższej warstwy, kolejnym rozwiązaniem jest obejście tego wyjątku w wyższej warstwie, dzięki czemu izoluje się metody wyższej warstwy od problemów, pochodzących z warstwy niższej. W takich przypadkach właściwe jest zapisanie informacji o wystąpieniu wyjątku przy wykorzystaniu mechanizmu rejestrującego, na przykład `java.util.logging`. Pozwala to administratorowi prześledzić problem bez zakłócania pracy kodu klienta i końcowego użytkownika.

Jeżeli zatem niemożliwe jest uniemożliwienie wyjątków z niższych warstw lub ich obsługa, należy użyć tłumaczenia wyjątków, o ile metoda niższego poziomu gwarantuje, że wszystkie jej wyjątki są propagowane do wyższego poziomu. Łączenie zapewnia najlepsze cechy obu metod — pozwala na zgłaszanie odpowiedniego wyjątku wyższego poziomu oraz przechwytywanie przyczyny w celu późniejszej analizy awarii (temat 63.).

Temat 62. Dokumentowanie wyjątków zgłaszanych przez metodę

Ważną częścią dokumentacji, wymaganą do właściwego wykorzystywania metod, jest opis wyjątków zgłaszanych przez metodę. Dlatego niezwykle ważne jest poświęcenie pewnego czasu w celu jasnego udokumentowania wszystkich wyjątków zgłaszanych przez metody.

Zawsze deklaruj (za pomocą znacznika Javadoc `@throws`) poszczególne przechwytywane wyjątki i opisuj precyzyjnie warunki, w których są one zgłaszane. Nie ułatwiaj sobie zadania pisząc, że metoda zgłasza wyjątek pewnej klasy bazowej, która jest bazą dla wielu klas wyjątków. Jako ekstremalny przykład można przytoczyć deklarowanie, że metoda zgłasza wyjątek `Exception` lub, co gorsza, `Throwable` — jest to niedopuszczalne. Oprócz tego, że deklaracja taka nie daje żadnych wskazówek programiście wykorzystującemu metodę, to jeszcze utrudnia wykorzystanie tej metody, ponieważ zasłania wszystkie wyjątki, jakie mogą być zgłoszone w tym samym kontekście.

Choć język nie wymaga deklarowania nieprzechwytywanych wyjątków, jakie może zgłaszać metoda, warto udokumentować je tak samo dokładnie, jak wyjątki przechwytywane. Wyjątki nieprzechwytywane z reguły reprezentują błędy programowe (temat 58.) i zapoznanie programistów z nimi pomaga im unikać tych błędów.

Dobrze opisana lista wyjątków nieprzechwytywanych, jakie mogą być zgłaszane przez metodę, efektywnie opisuje *warunki wstępne* jej prawidłowego wykonania. Niezmiernie ważne jest, aby każda dokumentacja metody zawierała warunki wstępne, a dokumentowanie wyjątków nieprzechwytywanych jest najlepszym sposobem spełnienia tego wymagania.

Szczególnie ważne jest udokumentowanie nieprzechwytywanych wyjątków, jakie mogą zgłaszać metody zadeklarowane w interfejsie. Dokumentacja ta staje się fragmentem *ogólnych założeń* interfejsu i pozwala na jednakowe działanie różnych implementacji tego samego interfejsu.

Należy skorzystać ze znacznika Javadoc `@throws` do udokumentowania wszystkich nieprzechwytywanych wyjątków zgłaszanych przez metodę, ale *nie* należy umieszczać tych wyjątków w deklaracji metody po słowie kluczowym `throws`. Ważne jest, aby programista korzystający z API wiedział, które wyjątki są przechwytywane, a które nie, ponieważ jego działania różnią się w tych dwóch przypadkach. Dokumentacja generowana przez znacznik Javadoc `@throws` w przypadku braku nagłówka metody wygenerowanego przez deklarację `throws` zapewnia silny wizualny sygnał pomagający programiście odróżnić przechwytywane wyjątki od nieprzechwytywanych.

Trzeba zauważyć, że dokumentowanie wszystkich nieprzechwytywanych wyjątków zgłaszanych przez metodę jest sytuacją idealną, ale nie zawsze osiągalną w praktyce. Gdy klasa jest modyfikowana, dodanie kolejnych nieprzechwytywanych wyjątków nie jest naruszeniem zgodności źródłowej ani binarnej. Załóżmy, że klasa wywołuje metodę z innej, niezależnie tworzonej klasy. Autorzy pierwszej klasy mogą dokładnie udokumentować wszystkie nieprzechwytywane wyjątki zgłaszane w swojej klasie, ale jeżeli druga klasa została zmodyfikowana i zgłasza nowe wyjątki, najprawdopodobniej pierwsza klasa (niepodlegająca modyfikacji) dokona propagacji nowego nieprzechwytywanego wyjątku, choć nie został on zadeklarowany.

Jeżeli wyjątek jest zgłaszany przez wiele metod klasy przy tych samych warunkach, dopuszczalne jest udokumentowanie takiego wyjątku w komentarzach do klasy zamiast umieszczania go przy każdej metodzie. Przykładem takiego wyjątku jest `NullPointerException`. Można w komentarzu dokumentującym klasę umieścić zdanie podobne do: „wszystkie metody tej klasy zgłaszają wyjątek `NullPointer` ↪`Exception`, jeżeli do dowolnego parametru przekazana jest wartość `null`".

Należy zatem dokumentować każdy wyjątek, który może być zgłaszany przez pisaną metodę. Odnosi się to zarówno do przechwytywanych, jak i nieprzechwytywanych wyjątków oraz zarówno do metod abstrakcyjnych, jak i konkretnych. Należy napisać osobne klauzule `throws` dla każdego przechwytywanego wyjątku i nie pisać klauzul `throws` dla wyjątków nieprzechwytywanych. Jeżeli nie udokumentujemy wyjątków zgłaszanych przez naszą metodę, dla innych programistów będzie trudno używać efektywnie naszych klas i interfejsów lub będzie to nawet niemożliwe.

Temat 63. Udostępnianie danych o błędzie

Gdy program zostanie zatrzymany przez nieprzechwycony wyjątek, system automatycznie drukuje informacje ze stosu wyjątków. Stos ten zawiera *reprezentację wyjątku* w postaci ciągu znaków — wynik metody `toString`. Dane te zwykle składają się z nazwy klasy, po której następują *informacje szczegółowe*. Często jest to jedyny fragment informacji, jakie otrzyma programista lub serwis oprogramowania w przypadku wystąpienia błędu w programie. Jeżeli trudno powtórzyć ten błąd, bardzo trudne może być uzyskanie dodatkowych danych. Dlatego niezwykle ważne jest, aby metoda `toString` dla wyjątku zwracała możliwie dużo informacji o przyczynie błędu. Inaczej mówiąc, ciąg reprezentujący wyjątek powinien umożliwiać dalszą analizę błędu.

Aby odszukać błąd, ciąg reprezentujący wyjątek powinien zawierać wartości wszystkich parametrów i pól „mających udział w przyczynie wyjątku". Na przykład, w przypadku wyjątku `IndexOutOfBounds` ciąg reprezentujący wyjątek powinien zawierać wartość dolnego i górnego zakresu oraz wartość indeksu, który nie zmieścił się w tych granicach. Informacje te niosą wiele informacji na temat przyczyny błędu. Każda z tych wartości może być nieprawidłowa. Aktualna wartość indeksu może być o jeden mniejsza od dolnego zakresu lub równa górnemu zakresowi lub może być to wartość całkowicie spoza zakresu w jedną lub drugą stronę. Dolna granica może być większa od górnej (poważne naruszenie niezmienników). Każda z tych sytuacji wskazuje na inny problem i informacja o tym niezmiernie ułatwia programiście poszukiwanie właściwego błędu.

Choć niezmiernie ważne jest dołączanie stosownych „twardych danych" do ciągu reprezentującego wyjątek, zwykle nie warto jest dodawać zbyt wielu informacji. Zrzut stosu przeznaczony jest do analizy kodu źródłowego i zazwyczaj zawiera dokładne nazwy plików i numery wierszy, w których rozpoczynają się wszystkie metody umieszczone na stosie. Długie opisy błędów są zwykle nadmiarowe — większość danych może być zebrana na podstawie analizy kodu źródłowego.

Ciąg reprezentujący wyjątek nie powinien zawierać komunikatów błędów zrozumiałych dla użytkownika. W przeciwieństwie do komunikatów dla użytkowników ciągi te są przeznaczone dla programistów lub serwisu przy analizie awarii. Dlatego dołączone dane są ważniejsze niż czytelność komunikatu.

Jednym ze sposobów upewnienia się, że ciąg reprezentujący wyjątek zawiera odpowiednie informacje pomagające odszukać błąd, jest wymaganie tych danych zamiast ciągu reprezentującego wyjątek. Na podstawie podanych informacji komunikat

może być generowany automatycznie. Na przykład, zamiast konstruktora z parametrem `String`, wyjątek `IndexOutOfBoundsException` mógłby mieć konstruktor, wyglądający następująco:

```
/**
 * Tworzy IndexOutOfBoundsException.
 *
 * @param lowerBound - najmniejsza dopuszczalna wartość indeksu.
 * @param upperBound - największa dopuszczalna wartość indeksu plus jeden.
 * @param index      - aktualna wartość parametru.
 */
public IndexOutOfBoundException(int lowerBound, int upperBound,
↪int index) {
  // Generowanie komunikatu opisującego błąd
  super(  "Dolny zakres: " + lowerBound +
        ", Górny zakres: " + upperBound +
        ", Indeks: "       + index);

  // Zapisanie informacji i błędzie dla programowego dostępu
  this.lowerBound = lowerBound;
  this.upperBound = upperBound;
  this.index = index;
}
```

Niestety, biblioteki platformy Java nie korzystają z tego idiomu zbyt często, ale polecamy gorąco jego stosowanie. Ułatwia to programiście odszukanie błędu w przypadku zgłoszenia wyjątku, a właściwie trudno jest w takiej sytuacji nie znaleźć błędu. W efekcie stosowania tego idiomu generowanie wysokiej klasy ciągu reprezentującego wyjątek jest przeniesione do klasy wyjątku, nie powodując nadmiarowego generowania takiego ciągu przez wszystkich programistów, korzystających z klasy.

W temacie 58. sugerowaliśmy, że właściwe jest udostępnienie metod w klasie wyjątku, pozwalających na odczytanie danych zapamiętanych w czasie generowania wyjątku (`lowerBound`, `upperBound` i `index` z powyższego przykładu). Metody takie mają większy sens w przypadku wyjątków przechwytywanych niż w przypadku nieprzechwytywanych, ponieważ w trakcie obsługi wyjątku przechwytywanego dane te mogą zostać wykorzystane do przywrócenia normalnego działania. Rzadko zdarza się (choć nie jest to niemożliwe), że programista będzie potrzebował dostępu do szczegółów wyjątku nieprzechwytywanego. Nawet w przypadku wyjątków nieprzechwytywanych jako ogólną zasadę można jednak zalecić tworzenie takich metod (temat 10.).

Temat 64. Zachowanie atomowości w przypadku błędu

Po zgłoszeniu wyjątku przez obiekt zwykle pożądane jest, aby obiekt cały czas był w dobrze zdefiniowanym stanie, nawet w przypadku wystąpienia błędu w czasie wykonywania operacji. Jest to szczególnie ważne w przypadku wyjątków przechwytywanych, które wywołujący może obsłużyć. **Nieudane wykonanie metody nie powinno powodować zmiany stanu obiektu sprzed wywołania tej metody.** Metoda o tej własności nazywana jest *atomową w przypadku błędu*.

Istnieje kilka sposobów na osiągnięcie tego efektu. Najprostszym jest projektowanie obiektów niezmiennych (temat 15.). Jeżeli obiekt jest niezmienny, atomowość w przypadku błędu jest zachowana bez żadnych nakładów. Jeżeli wykonanie operacji się nie powiedzie, może to spowodować, że nie zostanie utworzony wynikowy obiekt, lecz nigdy nie spowoduje powstania nieustalonego stanu istniejącego obiektu, ponieważ stan każdego z obiektów jest ustalany w trakcie tworzenia i nie może być później modyfikowany.

W przypadku metod operujących na obiektach modyfikowalnych najczęstszą metodą osiągnięcia atomowości w przypadku błędu jest sprawdzanie poprawności parametrów przed wykonaniem operacji (temat 38.). Dzięki temu wyjątki są zgłaszane przed rozpoczęciem modyfikacji obiektu. Dla przykładu przeanalizujmy metodę `Stack.pop` z tematu 6.

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

Jeżeli usuniemy początkowe sprawdzanie rozmiaru stosu, metoda nadal będzie zgłaszała wyjątek w przypadku próby pobrania elementu z pustego stosu. Jednak będzie pozostawiała pole `size` w niespójnym (ujemnym) stanie, powodując, że wszystkie kolejne wywołania metody zakończą się błędem. Dodatkowo, wyjątek zgłaszany przez metodę `pop` może być niezgodny z abstrakcją (temat 61.).

Podobnym podejściem przy uzyskiwaniu atomowości w przypadku błędu jest porządkowanie obliczeń w taki sposób, aby wszystkie fragmenty, w których mogą wystąpić błędy, były wykonywane przed zmianą stanu obiektu. Podejście to jest

naturalnym rozszerzeniem poprzedniego, gdy argumenty nie mogą być sprawdzone bez wykonania fragmentów operacji. Na przykład, w przypadku `TreeMap` elementy są sortowane według określonego porządku. Aby dodać element do `TreeMap`, element ten musi mieć typ porównywalny przy użyciu porządkowania, wykorzystanego w `TreeMap`. Próba dodania elementu niewłaściwego typu spowoduje zgłoszenie wyjątku `ClassCastException` w wyniku operacji wyszukiwania elementu w drzewie, zanim drzewo zostanie w jakikolwiek sposób zmienione.

Trzecim, najrzadszym podejściem jest tworzenie *kodu odtwarzającego*, który wykrywa błędy w czasie operacji i powoduje wycofanie obiektu do stanu z przed rozpoczęcia operacji. Podejście to jest najczęściej używane w przypadku trwałych struktur danych.

Ostatnim podejściem przy uzyskiwaniu atomowości w przypadku błędu jest wykonywanie operacji na kopii obiektu i zamiana zawartości obiektu źródłowego zawartością kopii po udanym zakończeniu operacji. Podejście to jest naturalne w przypadku, gdy obliczenia na tymczasowych strukturach danych mogą być przeprowadzone z dużą szybkością. Dla przykładu, metoda `Collections.sort` kopiuje wejściową listę do tablicy przed rozpoczęciem sortowania w celu zmniejszenia kosztu dostępu do elementów w wewnętrznej pętli sortowania. Jest to zrobione w celu poprawienia wydajności, ale dodatkowym zyskiem jest to, że w przypadku wystąpienia błędu wejściowa lista będzie niezmieniona.

Choć atomowość w przypadku błędu jest korzystna, nie zawsze może być osiągnięta. Na przykład, jeżeli dwa wątki próbują równolegle zmodyfikować obiekt bez właściwej synchronizacji, może być on pozostawiony w niewłaściwym stanie. Dlatego niewłaściwe może być założenie, że obiekt będzie nadawał się do użytku po przechwyceniu wyjątku `ConcurrentModificationException`. W przypadku błędów (w przeciwieństwie do wyjątków) nie jest możliwe odtworzenie stanu obiektu, więc metody nie muszą próbować osiągnięcia atomowości w przypadku zgłoszenia błędu.

Nawet, gdy atomowość w przypadku błędu jest możliwa do osiągnięcia, nie zawsze jest pożądana. W przypadku niektórych operacji będzie ona znacznie zwiększała koszt i stopień skomplikowania kodu. Jednak zwracając uwagę na to zagadnienie, przekonasz się, że najczęściej jest ona łatwa do osiągnięcia.

Jako zasadę należy przyjąć, że wyjątki będące częścią specyfikacji metody nie powinny zmieniać stanu obiektu sprzed wywołania tej metody. Gdy zasada ta nie jest spełniona, dokumentacja API powinna jasno wskazywać, w jakim stanie są pozostawiane obiekty. Niestety, wiele istniejących dokumentacji API nie spełnia tego warunku.

Temat 65. Nie ignoruj wyjątków

Uwaga ta może się wydawać oczywista, jednak jest tak często naruszana, że wymaga ciągłego powtarzania. Gdy projektanci API deklarują, że metoda zgłasza wyjątek, chcą przez to coś przekazać. Nie należy tego ignorować! Łatwo zignorować wyjątek, umieszczając wywołanie metody wewnątrz instrukcji `try` z pustym blokiem `catch`:

```
// Pusty blok catch powoduje zignorowanie wyjątku - wysoce podejrzane!
try {
    // ...
} catch (SomeException e) {
}
```

Pusty blok `catch` powoduje zignorowanie przeznaczenia wyjątków, którym jest wymuszenie obsługi sytuacji wyjątkowej. Zignorowanie wyjątku jest analogiczne do zignorowania alarmu przeciwpożarowego i jego wyłączanie, przez co nikt nie będzie miał szansy na zaalarmowanie o prawdziwym zagrożeniu. Gdy zobaczysz pusty blok `catch`, powinieneś natychmiast podjąć środki zaradcze. **Blok `catch` powinien zawierać co najmniej komentarz, wyjaśniający, dlaczego wyjątek ten jest ignorowany.**

Przykładem sytuacji, w której można zignorować wyjątek, jest renderowanie rysunku do animacji. Jeżeli ekran jest uaktualniany w regularnych odstępach czasu, najlepszą metodą obsługi nietrwałego błędu jest jego zignorowanie i poczekanie na następne uaktualnienie ekranu.

Porada z tego tematu odnosi się zarówno do wyjątków przechwytywanych, jak i nieprzechwytywanych. Ponieważ wyjątki reprezentują przewidywalne sytuacje wyjątkowe lub błędy w programie, ich zignorowanie spowoduje, że program będzie kontynuował działanie bez żadnego komunikatu. Program ten może zatrzymać się w dowolnym czasie w przyszłości w takim miejscu kodu, które nie jest w żaden sposób związane ze źródłem problemu. Właściwa obsługa błędów może całkowicie zapobiec awarii. Zwykłe pozostawienie nieprzechwytywanego wyjątku powoduje jego propagację i zatrzymanie jego działania, zachowując dane potrzebne do odszukania źródła błędu.