Zaktualizowane do wersji Java 17

Java

Przewodnik doświadczonego programisty

Wydanie III

Cay S. Horstmann

Helion

Tytuł oryginału: Core Java for the Impatient, 3rd Edition

Tłumaczenie: Krzysztof Bąbol z wykorzystaniem fragmentów książki „Java 8. Przewodnik doświadczonego programisty" w przekładzie Andrzeja Stefańskiego.

ISBN: 978-83-289-0140-7

Authorized translation from the English language edition, entitled Core Java for the Impatient, 3rd Edition by Cay Horstmann, published by Pearson Education, Inc, publishing as Addison-Wesley Professional, Copyright © 2023 Pearson Education, Inc.

Grafika na okładce: Morphart Creation/Shutterstock
Rysunki 1.1, 1.3: Microsoft
Rysunek 1.2: Eclipse Foundation
Rysunki 1.4, 1.5, 11.1: Oracle Corporation

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
https://helion.pl/user/opinie/jav8p3
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
https://ftp.helion.pl/przyklady/jav8p3.zip

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *https://helion.pl* (księgarnia internetowa, katalog książek)

Printed in Poland.

Spis treści

ROZDZIAŁ

5

Wyjątki, asercje i logi

W tym rozdziale:

- 5.1. Obsługa wyjątków
- 5.2. Asercje
- 5.3. Rejestrowanie danych
- Ćwiczenia

W wielu programach obsługa nieoczekiwanych zdarzeń może być bardziej skomplikowana niż implementacja optymistycznego scenariusza. Tak jak większość nowoczesnych języków programowania, Java ma rozbudowane mechanizmy obsługi wyjątków, umożliwiające przekazanie sterowania z miejsca wystąpienia błędu do miejsca, gdzie będzie on odpowiednio obsłużony. Dodatkowo wyrażenie `assert` dostarcza ustrukturyzowanego i wydajnego sposobu wyrażania wewnętrznych założeń. Na koniec zobaczysz, w jaki sposób za pomocą API do tworzenia logów zapisywać informacje na temat różnych zdarzeń, zarówno typowych, jak i podejrzanych, zachodzących podczas działania Twoich programów.

Najważniejsze punkty tego rozdziału:

1. Jeśli wyrzucasz wyjątek, sterowanie jest przekazywane do najbliższej procedury obsługi wyjątku.
2. W języku Java wyjątki kontrolowane są śledzone przez kompilator.
3. Do obsługi wyjątków służy konstrukcja `try/catch`.
4. Wyrażenie `try(`*`zasoby`*`)` automatycznie zamyka zasoby po normalnym wykonaniu lub po wystąpieniu wyjątku.
5. Użyj konstrukcji `try/finally` do obsługi innych akcji, które muszą zostać wykonane niezależnie od tego, czy wcześniejszy kod został wykonany poprawnie, czy nie.
6. Możesz przechwycić i ponownie wyrzucić ten sam wyjątek lub utworzyć łańcuch, wyrzucając inny wyjątek.
7. Zrzut stosu opisuje wszystkie wywołania metod trwające w danej chwili działania.
8. Asercja sprawdza warunek, jeśli sprawdzanie asercji jest włączone w klasie, i wyrzuca błąd, jeśli warunek nie jest spełniony.

9. Mechanizmy zapisujące dane dotyczące działania (ang. *loggers*) są zorganizowane w hierarchię i mogą gromadzić komunikaty do logowania opisane poziomami od `SEVERE` (ang. ciężki) do `FINEST` (ang. najmniejszy).
10. Mechanizmy obsługujące zapisywanie danych dotyczących działania mogą wysyłać gromadzone komunikaty do różnych miejsc i kontrolować format komunikatu.
11. Możesz kontrolować właściwości mechanizmów zapisujących dane dotyczące logowania za pomocą plików konfiguracyjnych dla logów.

5.1. Obsługa wyjątków

Co powinna zrobić metoda, gdy zdarzy się sytuacja, w której nie będzie w stanie wykonać swojego zadania? Typową odpowiedzią będzie, że metoda powinna zwrócić kod błędu. Jest to jednak niewygodne dla programisty wywołującego metodę. Kod wywołujący metodę jest wtedy zobowiązany sprawdzić, czy nie wystąpił błąd, i jeśli nie jest w stanie sobie z nim poradzić, zwrócić kod błędu do miejsca, z którego został wywołany. Nie jest niespodzianką, że programiści nie zawsze sprawdzają i przekazują zwracane kody, przez co błędy mogą przejść niezauważone i siać spustoszenie podczas dalszego działania programu.

Zamiast wymuszać przekazywanie kodów błędów w górę łańcucha wywołań metod, Java wspiera **obsługę wyjątków** — metoda może informować o poważnym problemie, „wyrzucając” (ang. *throw*) wyjątek. Jedna z metod w łańcuchu wywołań, choć niekoniecznie bezpośrednio wywołująca dany kod, jest odpowiedzialna za obsłużenie wyjątku poprzez jego „przechwycenie” (ang. *catch*). Najważniejszą korzyścią z obsługi wyjątków jest to, że rozdziela ona procesy wykrywania i obsługi błędów. W kolejnych punktach zobaczysz, w jaki sposób należy korzystać z wyjątków w języku Java.

5.1.1. Wyrzucanie wyjątków

Metoda może znaleźć się w takiej sytuacji, że nie będzie mogła wykonać zamierzonego zadania. Może brakować potrzebnych zasobów albo mogła zostać wywołana z niespójnym zestawem argumentów. W takim wypadku najlepiej jest wyrzucić wyjątek.

Przyjmijmy, że implementujesz metodę zwracającą losową liczbę całkowitą z zadanego zakresu:

```
public static int randInt(int low, int high) {
    return low + (int) (Math.random() * (high - low + 1));
}
```

Co powinno się stać, jeśli ktoś wywoła ją, pisząc `randInt(10, 5)`? Próba poprawienia tego prawdopodobnie nie jest najlepszym pomysłem, ponieważ takie wywołanie może być efektem więcej niż jednego problemu. Zamiast tego wyrzuć odpowiedni wyjątek:

```
if (low > high)
  throw new IllegalArgumentException(
    "dolne ograniczenie powinno być <= górne, ale dolne wynosi %d, a górne %d".formatted(low, high));
```

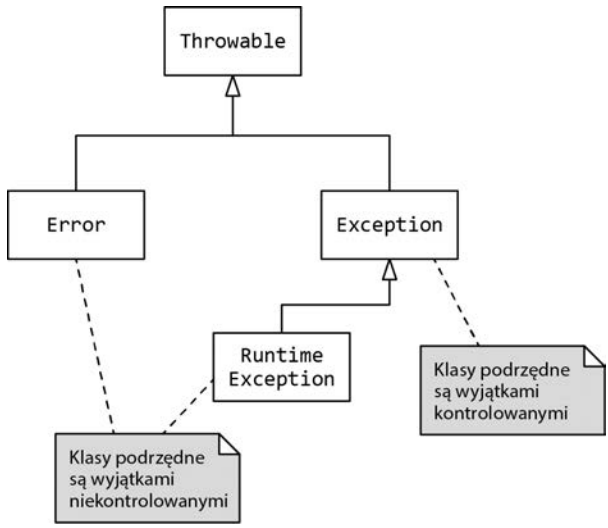
Jak widzisz, wyrażenie `throw` zostało użyte do „wyrzucenia” obiektu klasy `IllegalArgument`
`↪Exception`. Obiekt ten jest utworzony z komunikatem o błędzie. W następnym punkcie zobaczysz, jak wybrać odpowiednią klasę wyjątku.

Gdy wykonywane jest wyrażenie `throw`, normalny ciąg wywołań jest natychmiastowo przerywany. Metoda `randInt` przerywa działanie i nie zwraca wartości do wywołującego ją kodu. Zamiast tego sterowanie jest przekazywane do kodu obsługującego wyjątek w sposób zademonstrowany w punkcie 5.1.4, „Przechwytywanie wyjątków”.

5.1.2. Hierarchia wyjątków

Rysunek 5.1 pokazuje hierarchię wyjątków w języku Java. Wszystkie wyjątki są klasami podrzędnymi klasy `Throwable`. Klasy podrzędne klasy `Error` są wyjątkami wyrzucanymi, gdy zdarzy się coś wyjątkowego i nie jest możliwe, by program mógł to obsłużyć, na przykład brak pamięci. W razie błędów niewiele możesz zrobić poza wyświetleniem użytkownikowi komunikatu, że stało się coś bardzo złego.

Rysunek 5.1.
Hierarchia wyjątków



Wyjątki zgłaszane programiście są klasami podrzędnymi klasy `Exception`. Te wyjątki należą do dwóch kategorii:

- wyjątki **niekontrolowane** są klasami podrzędnymi klasy `RuntimeException`;
- wszystkie inne wyjątki są wyjątkami **kontrolowanymi**.

Jak zobaczysz w kolejnym punkcie, programiści muszą przechwytywać wyjątki kontrolowane lub deklarować je w nagłówku metody. Kompilator sprawdza, czy tego typu wyjątki są prawidłowo obsługiwane.

Nazwa `RuntimeException` nie jest zbyt trafna. Oczywiście wszystkie wyjątki powstają w czasie działania (ang. *runtime*). Jednak wyjątki klas podrzędnych klasy `RuntimeException` nie są sprawdzane podczas kompilacji.

Wyjątki kontrolowane są stosowane w sytuacjach, gdy wystąpienie błędu powinno zostać przewidziane. Typowym powodem wystąpienia błędu jest pobieranie lub wyświetlanie informacji. Pliki mogą być uszkodzone, a połączenia sieciowe mogą zostać przerwane. Wiele klas wyjątków rozszerza `IOException` i powinieneś używać odpowiedniej do zgłaszania błędów, które się pojawią. Na przykład gdy plik, który powinien znajdować się w danym miejscu, nie znajduje się tam, należy wyrzucić `FileNotFoundException`.

Wyjątki niekontrolowane wskazują błędy logiczne spowodowane przez programistów, a nie przez nieuniknione zagrożenia zewnętrzne. Na przykład wyjątek `NullPointerException` nie jest kontrolowany. Praktycznie każda metoda może go wyrzucić i programiści nie powinni tracić czasu na jego przechwytywanie. Zamiast tego powinni przede wszystkim upewniać się, że nie odwołują się do zmiennych zawierających `null`.

Czasem podczas implementacji trzeba na podstawie własnej oceny rozróżniać kontrolowane i niekontrolowane wyjątki. Rozważmy wywołanie `Integer.parseInt(str)`. Wyrzuca ono niekontrolowany wyjątek `NumberFormatException`, jeśli `str` nie zawiera poprawnej wartości całkowitej. Z drugiej strony `Class.forName(str)` wyrzuca kontrolowany wyjątek `ClassNotFoundException`, jeśli `str` nie zawiera poprawnej nazwy klasy.

Skąd taka różnica? Powodem jest to, że możliwe jest sprawdzenie, czy ciąg znaków jest poprawną liczbą całkowitą przed wywołaniem `Integer.parseInt`, ale nie jest możliwe ustalenie, czy klasa może zostać wczytana przed wykonaniem próby jej wczytania.

API języka Java dostarcza wielu klas wyjątków, takich jak `IOException`, `IllegalArgumentException` itd. Powinieneś korzystać z nich w odpowiednich miejscach. Jeśli jednak żadna ze standardowych klas wyjątków nie pasuje w danym miejscu, możesz utworzyć własną, rozszerzając `Exception`, `RuntimeException` lub inną istniejącą klasę wyjątku.

Gdy już to robisz, dobrym pomysłem jest utworzenie zarówno konstruktora bezargumentowego, jak i konstruktora z ciągiem znaków zawierających komunikat. Na przykład:

```
public class FileFormatException extends IOException {
    public FileFormatException() {}
    public FileFormatException(String message) {
        super(message);
    }
    // Dodaj też konstruktory dla przekierowanych wyjątków — patrz punkt 5.1.7
}
```

5.1.3. Deklarowanie wyjątków kontrolowanych

Każda metoda, która może wyrzucić wyjątek kontrolowany, musi zadeklarować go w nagłówku metody za pomocą wyrażenia `throws`:

```
public void write(Object obj, String filename)
    throws IOException, ReflectiveOperationException
```

Należy wypisać wyjątki, które metoda może wyrzucić za pomocą wyrażenia `throw` lub wywołania innej metody zawierającej wyrażenie `throws`.

W wyrażeniu `throws` możesz połączyć wyjątki we wspólnych klasach nadrzędnych. Czy to jest dobry pomysł, czy nie — to już zależy od konkretnych wyjątków. Na przykład: jeśli metoda może wyrzucić wiele klas podrzędnych `IOException`, ma sens zebranie ich wszystkich w wyrażeniu `throws IOException`. Jeśli jednak wyjątki nie mają ze sobą nic wspólnego, nie należy łączyć ich w `throws Exception` — psuje to cały sens sprawdzania wyjątków.

Niektórzy programiści wstydzą się przyznać, że metoda może wyrzucić wyjątek. Nie lepiej byłoby go obsłużyć? Wręcz przeciwnie. Powinieneś każdemu wyjątkowi utorować drogę do odpowiedniego kodu obsługi. Złotą zasadą dla wyjątków jest „wyrzucaj wcześnie, przechwytuj późno".

Gdy przesłaniasz metodę, nie może ona wyrzucać większej liczby kontrolowanych wyjątków, niż zadeklarowano w metodzie klasy nadrzędnej. Na przykład: jeśli rozszerzasz metodę `write` opisaną na początku tego punktu, metoda przesłaniająca może wyrzucać mniej wyjątków:

```
public void write(Object obj, String filename)
    throws FileNotFoundException
```

Ale jeśli metoda spróbowałaby wyrzucić niezwiązany wyjątek kontrolowany, taki jak `InterruptedException`, kod się nie skompiluje.

Jeśli metoda klasy nadrzędnej nie ma wyrażenia `throws`, żadna metoda przesłaniająca nie może wyrzucać wyjątku kontrolowanego.

Możesz użyć znacznika `@throws` z narzędzia `javadoc`, aby zapisywać, kiedy metoda wyrzuca (kontrolowany lub niekontrolowany) wyjątek. Większość programistów robi to tylko wtedy, gdy jest do opisania coś ważnego. Na przykład niewiele wnosi informowanie użytkowników, że wyrzucany jest wyjątek `IOException`, gdy pojawi się problem z pobieraniem lub wysyłaniem danych. Ale znaczące mogą być komentarze takie jak poniżej:

```
@throws NullPointerException, jeśli zmienna filename ma wartość null
@throws FileNotFoundException, jeśli plik o nazwie zapisanej w zmiennej filename nie istnieje
```

Nigdy nie określa się typu wyjątku wyrażenia lambda. Jeśli jednak wyrażenie lambda może wyrzucić kontrolowany wyjątek, możesz go jedynie przekazać do interfejsu funkcyjnego, którego metoda deklaruje ten wyjątek. Na przykład wywołanie

```
list.forEach(obj -> write(obj, "output.dat"));
```

jest błędem. Typem parametru metody `forEach` jest interfejs funkcyjny

```
public interface Consumer<T> {
    void accept(T t);
}
```

Metoda `accept` jest zadeklarowana w taki sposób, że nie może wyrzucić żadnego wyjątku kontrolowanego.

5.1.4. Przechwytywanie wyjątków

Aby przechwycić wyjątek, musisz przygotować blok `try`. W najprostszej postaci wygląda to tak:

```
try {
    wyrażenia
} catch (KlasaWyjątku ex) {
    obsługa wyjątku
}
```

Gdy podczas wykonywania wyrażeń z bloku `try` pojawi się wyjątek danej klasy, sterowanie jest przekazywane do kodu obsługującego wyjątek. Zmienna opisująca wyjątek (`ex` w naszym przykładzie) wskazuje na obiekt wyjątku, który w razie potrzeby może zostać dodatkowo przeanalizowany przez kod obsługujący wyjątek.

Możesz tę najprostszą strukturę zmodyfikować na dwa sposoby. Możesz mieć wiele segmentów kodu obsługujących wyjątki różnych klas:

```
try {
    wyrażenia
} catch (KlasaWyjątku1 ex) {
    obsługa wyjątku1
} catch (KlasaWyjątku2 ex) {
    obsługa wyjątku2
} catch (KlasaWyjątku3 ex) {
    obsługa wyjątku3
}
```

Wyrażenia `catch` są przeszukiwane od góry do dołu, dlatego na początku należy umieścić klasy najbardziej precyzyjnie wskazujące rodzaj błędu.

Opcjonalnie można za pomocą jednego kodu obsługiwać wiele klas wyjątków:

```
try {
    wyrażenia
} catch (KlasaWyjątku1 | KlasaWyjątku2 | KlasaWyjątku3) {
    obsługa wyjątków
}
```

W takim wypadku kod obsługujący wyjątki może korzystać tylko z takich metod klas wyjątków, które znajdują się we wszystkich klasach wyjątków.

5.1.5. Wyrażenie try z określeniem zasobów

Problemem przy obsłudze wyjątków jest zarządzanie zasobami. Załóżmy, że zapisujesz do pliku i zamykasz go po zakończeniu:

```
ArrayList<String> lines = ...;
var out = new PrintWriter("output.txt");
for (String line : lines) {
    out.println(line.toLowerCase());
}
out.close();
```

W kodzie tym ukryte jest niebezpieczeństwo. Jeśli któraś z metod wyrzuci wyjątek, wywołanie `out.close()` nie zostanie wykonane. To źle. Zapisywane dane mogą zostać utracone albo, jeśli wyjątek zostanie wyrzucony wiele razy, w systemie mogą skończyć się deskryptory plików.

Specjalna odmiana wyrażenia `try` może rozwiązać ten problem. Możesz określić dowolną liczbę zasobów w nagłówku wyrażenia `try`. Każdy zasób musi należeć do klasy implementującej interfejs `AutoCloseable`. Możesz deklarować zmienne w nagłówku bloku `try`:

```
ArrayList<String> lines = ...;
try (var out = new PrintWriter("output.txt")) { // Deklaracja zmiennej
    for (String line : lines)
        out.println(line.toLowerCase());
}
```

Alternatywnie możesz dostarczyć wcześniej zadeklarowaną zmienną do nagłówka:

```
var out = new PrintWriter("output.txt");
try (out) { // Zmienna, której wartość nie będzie się zmieniała
    for (String line : lines)
        out.println(line.toLowerCase());
}
```

Interfejs `AutoCloseable` ma jedną metodę

```
public void close() throws Exception
```

Istnieje też interfejs `Closeable`. Jest to interfejs podrzędny interfejsu `AutoCloseable`, również zawierający pojedynczą metodę `close`. Tamta metoda jednak ma zadeklarowaną możliwość wyrzucania wyjątku `IOException`.

Po zakończeniu działania bloku `try`, niezależnie od tego, czy działanie w normalny sposób dotarło do końca, czy z powodu wyrzucenia wyjątku, wywoływane są metody `close` obiektów reprezentujących zasoby. Na przykład:

```
try (var out = new PrintWriter("output.txt")) {
    for (String line : lines) {
        out.println(line.toLowerCase());
    }
} // Tutaj wywoływana jest metoda out.close()
```

Możesz zadeklarować wiele zasobów oddzielonych średnikami. Oto przykład z deklaracjami dwóch zasobów:

```
try (var in = new Scanner(Paths.get("/usr/share/dict/words"));
        var out = new PrintWriter("output.txt")) {
    while (in.hasNext())
        out.println(in.next().toLowerCase());
}
```

Zasoby są zamykane w odwrotnej kolejności, niż były inicjalizowane, czyli instrukcja `out.close()` zostanie wywołana przed `in.close()`.

Załóżmy, że konstruktor `PrintWriter` wyrzuca wyjątek. W tym momencie zasób `in` jest już zainicjalizowany, ale zasób `out` nie jest. Wyrażenie `try` poradzi sobie z tym: wywoła `in.close()` i przekaże dalej wyjątek.

Niektóre metody `close` mogą wyrzucać wyjątki. Jeśli tak się stanie przy normalnym zakończeniu bloku `try`, wyjątek zostanie przekazany do kodu wywołującego. Jeśli jednak wyrzucony był wcześniej inny wyjątek powodujący wywołanie metod `close` na zasobach i jedna z tych metod wyrzuciła wyjątek, ten wyjątek prawdopodobnie jest mniej ważny niż oryginalny.

W takiej sytuacji oryginalny wyjątek zostaje ponownie wyrzucony, a wyjątki z wywoływanej metody `close` są przechwytywane i dołączane jako „wyciszone" wyjątki. Jest to bardzo użyteczny mechanizm, który trudno byłoby samodzielnie zaimplementować (patrz ćwiczenie 5.). Gdy przechwytujesz pierwotny wyjątek, możesz pobrać wyciszone wyjątki za pomocą metody `getSuppressed`:

```
try {
    ...
} catch (IOException ex) {
    Throwable[] secondaryExceptions = ex.getSuppressed();
    ...
}
```

Jeśli chcesz samodzielnie zaimplementować taki mechanizm w (miejmy nadzieję rzadkiej) sytuacji, gdy nie możesz użyć wyrażenia `try` z zadeklarowanymi zasobami, wywołaj `ex.addSup` ↪`pressed(`*`wyciszonyWyjątek`*`)`.

Wyrażenie `try` z zadeklarowanymi zasobami może opcjonalnie mieć wyrażenia `catch`, przechwytujące wszystkie wyjątki z wyrażenia.

5.1.6. Klauzula finally

Jak już widziałeś, wyrażenie `try` z zadeklarowanymi zasobami automatycznie zamyka zasoby niezależnie od tego, czy wystąpi wyjątek. Czasem musisz zrobić porządek z czymś, co nie jest `AutoCloseable`. W takim wypadku skorzystaj z klauzuli `finally`:

```
try {
    Praca
} finally {
    Sprzątanie
}
```

Klauzula `finally` jest wykonywana, gdy kończone jest wykonanie bloku `try`, niezależnie od tego, czy zakończył się on normalnie, czy z powodu wyjątku.

Ten wzorzec występuje, gdy musisz ustawić i zwolnić blokadę, zwiększyć i zmniejszyć licznik lub umieścić coś na stosie i pobrać to ze stosu po zakończeniu. Zechcesz mieć pewność, że takie operacje zostaną wykonane niezależnie od tego, jaki wyjątek zostanie wyrzucony.

Powinieneś unikać wyrzucania wyjątku w klauzuli `finally`. Jeśli działanie bloku `try` zakończyło się wyrzuceniem wyjątku, zostanie to przesłonięte wyjątkiem z klauzuli `finally`. Mechanizm tłumienia wyjątków, który widziałeś w poprzednim punkcie, działa tylko w przypadku wyrażeń `try` z zadeklarowanymi zasobami.

Podobnie klauzula `finally` nie powinna zawierać wyrażenia `return`. Jeśli kod z bloku `try` również zawiera wyrażenie `return`, takie samo wyrażenie umieszczone w klauzuli `finally` zamieni zwróconą wcześniej wartość. Przyjrzyj się temu przykładowi:

```
public static int parseInt(String s) {
    try {
        return Integer.parseInt(s);
    } finally {
        return 0; // Błąd
    }
}
```

Wydawałoby się, że po wywołaniu `parseInt("42")` z ciała bloku `try` zostanie zwrócona liczba 42. Zanim jednak nastąpi faktyczny powrót z metody, wywołana zostanie klauzula `finally`, wskutek czego metoda zwróci liczbę 0, a pierwotnie zwracana wartość zostanie zignorowana.

Może być jeszcze gorzej. Weź po uwagę wywołanie `parseInt("zero")`. Metoda `Integer.parseInt` wyrzuca wyjątek `NumberFormatException`. Potem wykonywana jest klauzula `finally`, a instrukcja `return` wchłania wyjątek!

Przeznaczeniem ciała klauzuli `finally` jest sprzątanie zasobów. Nie umieszczaj w niej instrukcji, które zmieniają przepływ sterowania (`return`, `throw`, `break`, `continue`).

Możliwe jest utworzenie wyrażeń `try` z klauzulami `catch` i klauzulą `finally`. Musisz jednak uważać na wyjątki w klauzuli `finally`. Na przykład spójrz na ten blok `try` pochodzący z tutoriala *online*:

```
BufferedReader in = null;
try {
    in = Files.newBufferedReader(path, StandardCharsets.UTF_8);
    Wczytywanie z in
} catch (IOException ex) {
    System.err.println("Przechwycony wyjątek: " + ex.getMessage());
} finally {
    if (in != null) {
        in.close(); // Ostrzeżenie — może wyrzucić wyjątek
    }
}
```

Oczywiście programista miał na myśli przypadek, gdy metoda `Files.newBufferedReader` wyrzuca wyjątek. Wygląda na to, że ten kod powinien przechwytywać i wyświetlać wszystkie wyjątki związane z pobieraniem i wysyłaniem danych, ale w rzeczywistości jeden jest pominięty: ten, który może być wyrzucony przez `in.close()`. Często lepiej jest przepisać skomplikowane wyrażenie `try/catch/finally` i zastąpić je wyrażeniem `try` z zadeklarowanymi zasobami albo zagnieżdżonym wyrażeniem `try/finally` wewnątrz wyrażenia `try/catch` — patrz ćwiczenie 6.

5.1.7. Ponowne wyrzucanie wyjątków i łączenie ich w łańcuchy

Gdy pojawia się wyjątek, możesz nie wiedzieć, co z nim zrobić, ale możesz zechcieć zapisać informacje o błędzie. W takim wypadku ponownie wyrzuć wyjątek, tak by odpowiedni kod obsługi mógł go obsłużyć:

```
try {
    Zadania
}
catch (Exception ex) {
    logger.log(level, message, ex);
    throw ex;
}
```

Coś nietypowego dzieje się, jeśli ten kod znajduje się wewnątrz metody, która może wyrzucić wyjątek kontrolowany. Przypuśćmy, że zawierająca go metoda jest zadeklarowana jako:

```
public void read(String filename) throws IOException
```

Na pierwszy rzut oka wygląda na to, że konieczne będzie zmodyfikowanie klauzuli `throws` na `throws Exception`. Kompilator języka Java jednak skrupulatnie śledzi przepływ i ustala, że `ex` może być wyjątkiem wyrzuconym przez jedno z wyrażeń w bloku `try`, a nie dowolnym wyjątkiem.

Czasem będziesz chciał zmienić klasę wyrzucanego wyjątku. Na przykład konieczne może być zgłoszenie problemu z podsystemem za pomocą wyjątku takiej klasy, którą użytkownik podsystemu będzie mógł zinterpretować. Załóżmy, że pojawił się błąd z bazą danych w serwlecie. Kod uruchamiający serwlet może nie chcieć znać szczegółów problemu, ale na pewno chce wiedzieć o tym, że w serwlecie pojawił się błąd. W takim wypadku przechwyć oryginalny wyjątek i zamień go na wyjątek wyższego rzędu, tworząc łańcuch wyjątków:

```
try {
    Dostęp do bazy danych
}
catch (SQLException ex) {
    throw new ServletException("błąd bazy danych", ex);
}
```

Po przechwyceniu `ServletException` oryginalny wyjątek może zostać pobrany w taki sposób:

```
Throwable cause = ex.getCause();
```

Klasa `ServletException` ma konstruktor, który przez parametr pobiera przyczynę wyrzucenia wyjątku. Nie wszystkie klasy wyjątków to robią. W takim wypadku będziesz musiał wywołać metodę `initCause` w taki sposób:

```
try {
    Dostęp do bazy danych
}
catch (SQLException ex) {
    var ex2 = new CruftyOldException("błąd bazy danych");
    ex2.initCause(ex);
    throw ex2;
}
```

Jeśli utworzysz swoją własną klasę wyjątku, powinieneś dostarczyć też, poza dwoma konstruktorami opisanymi w punkcie 5.1.2, „Hierarchia wyjątków”, dodatkowe konstruktory:

```
public class FileFormatException extends IOException {
    ...
    public FileFormatException(Throwable cause) { initCause(cause); }
    public FileFormatException(String message, Throwable cause) {
        super(message);
        initCause(cause);
    }
}
```

Tworzenie łańcucha wyjątków przydaje się też w sytuacji, gdy wyjątek kontrolowany pojawia się w metodzie, która nie może wyrzucać wyjątków kontrolowanych. Możesz przechwycić wyjątek kontrolowany i zamienić go na wyjątek niekontrolowany.

5.1.8. Nieprzechwycone wyjątki i ślad stosu wywołań

Jeśli wyjątek nie zostanie nigdzie przechwycony, zostanie wyświetlony **ślad stosu wywołań** (ang. *stack trace*) — lista wszystkich wywołań metod wykonywanych w chwili wyrzucenia wyjątku. Ślad stosu jest przesyłany do `System.err`, strumienia z komunikatami o błędach.

Jeśli chcesz zapisać wyjątek w innym miejscu, na przykład w celu zbadania przez pomoc techniczną, przygotuj domyślny kod obsługujący nieprzechwycone wyjątki:

```
Thread.setDefaultUncaughtExceptionHandler((thread, ex) -> {
    Zapisz wyjątek
});
```


Uwaga

Nieprzechwycone wyjątki kończą działanie wątku, w którym zostały wyrzucone. Jeśli Twoja aplikacja ma tylko jeden wątek (czyli tak jak w programach, które dotąd widziałeś), działanie programu kończy się po wywołaniu kodu obsługującego nieprzechwycone wyjątki.

Czasem jesteś zmuszony przechwycić wyjątek i nie za bardzo wiadomo, co z nim zrobić. Na przykład metoda `Class.forName` wyrzuca wyjątek kontrolowany, który musisz obsłużyć. Zamiast ignorować wyjątek, przynajmniej wyświetl ślad stosu wywołań:

```
try {
    Class<?> cl = Class.forName(className);
    ...
} catch (ClassNotFoundException ex) {
    ex.printStackTrace();
}
```

Jeśli chcesz zapisać ślad stosu dla wyjątku, musisz umieścić go w ciągu znaków w taki sposób:

```
var out = new ByteArrayOutputStream();
ex.printStackTrace(new PrintWriter(out));
String description = out.toString();
```


Uwaga

Jeśli musisz dokładniej przeanalizować ślad stosu, użyj klasy `StackWalker`. Przykładowo poniższe polecenia wyświetlają wszystkie elementy stosu:

```
StackWalker walker = StackWalker.getInstance();
walker.forEach(frame -> System.err.println("Frame: " + frame));
```

Możesz też przeanalizować szczegółowo instancje `StackWalker.StackFrame`. Szczegóły znajdziesz w dokumentacji API.

5.1.9. Metody API pozwalające wyrzucać wyjątki

Klasa `Objects` zawiera metodę pozwalającą na wygodne sprawdzanie wartości `null` parametrów. Przykład jej wykorzystania:

```
public void process(String direction) {
    this.direction = Objects.requireNonNull(direction);
    ...
}
```

Jeśli zmienna `direction` ma wartość `null`, wyrzucany jest wyjątek `NullPointerException`. Na pierwszy rzut oka wygląda to na trochę bezcelowe, gdyż w razie użycia tej wartości i tak wyrzucony zostałby wyjątek `NullPointerException`. Popatrz jednak na to od strony analizowania śladu stosu wywołań. Gdy widzisz, że źródłem problemu jest wywołanie `requireNonNull`, wiesz już, co robisz źle.

Możesz też dodać ciąg znaków z informacją na temat wyjątku:

```
this.direction = Objects.requireNonNull(direction,
    "zmienna direction nie może mieć wartości null");
```

Inny wariant tej metody pozwala Ci podstawić alternatywną wartość zamiast wyrzucania wyjątku:

```
this.direction = Objects.requireNonNullElse(direction, "Północ");
```

Jeśli obliczenie wartości domyślnej jest kosztowne, możesz zastosować jeszcze inny sposób:

```
this.direction = Objects.requireNonNullElseGet(direction,
      () -> System.getProperty("com.horstmann.direction.default"));
```

Wyrażenie lambda jest obliczane jedynie, jeśli zmienna `direction` ma wartość `null`.

Istnieje też metoda, która ułatwia sprawdzanie zakresów:

```
this.index = Objects.checkIndex(index, length);
```

Jeśli wartość index mieści się w przedziale od 0 do length – 1, metoda zwraca tę wartość, a w przeciwnym wypadku wyrzuca wyjątek IndexOutOfBoundsException.

5.2. Asercje

Asercja to często używany idiom związany z programowaniem defensywnym. Załóżmy, że jesteś przekonany, że pewna właściwość ma wartość, i korzystasz z niej w swoim kodzie. Na przykład możesz obliczać

```
double y = Math.sqrt(x);
```

Jesteś pewien, że x nie jest liczbą ujemną. Mimo to wolisz sprawdzić dwa razy, niż ryzykować pojawienie się w dalszej części obliczeń wartości zmiennoprzecinkowej NaN. Mógłbyś oczywiście wyrzucić wyjątek:

```
if (x < 0) throw new IllegalStateException(x + " < 0");
```

Ten warunek jednak pozostaje w programie, nawet po zakończeniu testowania i spowalnia wykonanie kodu. Mechanizm asercji pozwala na umieszczenie poleceń do testowania wraz z możliwością automatycznego usunięcia ich w kodzie produkcyjnym.

W języku Java asercje służą do wspomagania testowania poprzez sprawdzanie, czy spełnione są wewnętrzne założenia, a nie jako mechanizm do wymuszania spełnienia warunków brzegowych. Na przykład: jeśli chcesz zgłosić, że metoda publiczna otrzymała nieodpowiedni argument, nie korzystaj z asercji, tylko wyrzuć wyjątek IllegalArgumentException.

5.2.1. Użycie asercji

Istnieją dwa rodzaje asercji w języku Java:

```
assert warunek;
assert warunek : wyrażenie;
```

Wyrażenie assert sprawdza warunek i wyrzuca błąd AssertionError, jeśli zwraca on wartość false. W drugiej postaci wyrażenie jest zamieniane na ciąg znaków, który jest dołączany jako komunikat do obiektu reprezentującego błąd.

Jeśli wyrażenie jest Throwable, jest ono też ustawiane jako przyczyna błędu asercji (patrz punkt 5.1.7, „Ponowne wyrzucanie wyjątków i łączenie ich w łańcuchy”).

Aby na przykład upewnić się, że x jest liczbą nieujemną, możesz po prostu użyć wyrażenia

```
assert x >= 0;
```

Możesz też przekazać rzeczywistą wartość x do obiektu AssertionError, by mogła być później wyświetlona:

```
assert x >= 0 : x;
```

5.2.2. Włączanie i wyłączanie asercji

Domyślnie asercje są wyłączone. Można je włączyć, uruchamiając program z parametrami -enableassertions lub -ea:

```
java -ea MainClass
```

Nie musisz rekompilować swojego programu, ponieważ włączanie i wyłączanie asercji jest wykonywane przez program wczytujący klasy (ang. *class loader*). Gdy asercje są wyłączone, program wczytujący klasy wycina kod asercji, by nie opóźniał on działania kodu. Możesz nawet włączyć asercje w wybranych klasach lub całych pakietach. Na przykład:

```
java -ea:MyClass -ea:com.mojafirma.mojabiblioteka... MainClass
```

Takie polecenie włączy asercje dla klasy `MyClass` i wszystkich klas w pakiecie `com.mojafirma.mojabiblioteka` *i pakietach niższego rzędu*. Opcja `-ea...` włącza asercje we wszystkich klasach domyślnego pakietu.

Możesz też wyłączyć asercje w wybranych klasach i pakietach za pomocą parametru `-disableassertions` lub `-da`:

```
java -ea:... -da:MyClass MainClass
```

Gdy korzystasz z przełączników `-ea` i `-da`, aby włączyć i wyłączyć wszystkie asercje (a nie tylko określone klasy czy pakiety), nie są one stosowane do „klas systemowych", które są ładowane bez użycia programów ładujących klasy. Użyj przełącznika `-enablesystemassertions` lub `-esa`, by włączyć asercje w klasach systemowych.

Możliwe jest też programowe kontrolowanie statusu asercji w programach ładujących klasy za pomocą następujących metod:

```
void ClassLoader.setDefaultAssertionStatus(boolean włączony);
void ClassLoader.setClassAssertionStatus(String nazwaKlasy, boolean włączony);
void ClassLoader.setPackageAssertionStatus(String nazwaPakietu, boolean włączony);
```

Tak jak w przypadku opcji wiersza poleceń `-enableassertions`, metoda `setPackageAssertionStatus` ustawia status asercji dla wybranego pakietu i pakietów niższego rzędu.

5.3. Rejestrowanie danych

Każdy programista języka Java zna proces wstawiania wywołań `System.out.println` do sprawiającego problemy kodu w celu zbadania zachowania programu. Oczywiście po ustaleniu przyczyny problemu usuwasz wyrażenia `print` — tylko po to, by wstawić je ponownie, gdy pojawi się kolejny problem. Frameworki logowania pozwalają na rozwiązanie tego problemu.

5.3.1. Czy w Javie warto korzystać z frameworka rejestrowania danych?

Java ma standardowy framework rejestrowania danych, zwykle nazywany tak jak jego pakiet — `java.util.logging` (nazwa ta jest czasami skracana do `j.u.l`). Często jednak używane są inne frameworki, które mają więcej funkcji, na przykład Log4j (*https://logging.apache.org/log4j/2.x*) i Logback (*https://logback.qos.ch*).

Jeśli chcesz pozwolić użytkownikom swojego kodu na wybór frameworka rejestrowania danych, skorzystaj z biblioteki — „fasady", która będzie wysyłać rejestrowane komunikaty do preferowanego przez nich frameworka. Do często używanych fasad z przyjemnym API należy SLF4J (*https://www.slf4j.org*). Inna fasada to wprowadzony przez JEP 246 interfejs Platform Logging API. Jest on bardzo prosty, ale stanowi część JDK. Fasada jest czasem nazywana *frontendem*. Zapewnia ona interfejs API, za pomocą którego programiści rejestrują komunikaty. *Backend* (zaplecze) odpowiada za filtrowanie i formatowanie komunikatów, a także umieszczanie ich w odpowiednim miejscu. Backend musi być możliwy do skonfigurowania przez osoby przeprowadzające wdrożenie, zazwyczaj w drodze edytowania plików konfiguracyjnych.

W kolejnych punktach przedstawię użycie interfejsu Platform Logging API jako frontendu, a mechanizmu `java.util.logging` jako backendu. Jeśli zadowala Cię taki interfejs API frontendu, jest to rozsądny wybór, ponieważ backend zawsze można wymienić.

Backend `java.util.logging` ma mniejsze możliwości niż jego bardziej popularne odpowiedniki, ale w wielu zastosowaniach będzie wystarczający. Z powodu prostoty jest mniej narażony na ataki, w przeciwieństwie do frameworka Log4j. Jego zapoznane funkcje pozwoliły hakerom preparować dane wejściowe, których rejestrowanie w programie skutkowało uruchomieniem szkodliwego kodu. Dzięki przeanalizowaniu interfejsu Platform Logging API i backendu `java.util.logging` uzyskasz solidne podstawy wiedzy o możliwościach frameworków rejestrowania danych, niezależnie od tego, czy będziesz ich potem używać, czy nie.

5.3.2. ABC rejestrowania danych

Mechanizmy rejestrujące platformy implementują interfejs `System.Logger`. Każdy z nich ma nazwę. Może ona być dowolna, ale często jest to nazwa pakietu z klasą, której metody generują rejestrowane komunikaty. Dostęp do mechanizmu rejestrującego platformy uzyskuje się następująco:

```
System.Logger logger = System.getLogger("com.mojafirma.mojaaplikacja");
```

Gdy odwołasz się do mechanizmu logowania z wybraną nazwą po raz pierwszy, zostanie on utworzony. Kolejne wywołania z żądaniem mechanizmu o tej samej nazwie zwrócą ten sam obiekt.

Teraz możesz już rejestrować dane:

```
logger.log(System.Logger.Level.INFO, "Otwarcie pliku " + nazwaPliku);
```

Dane zapisywane są w takiej postaci:

```
Aug 04, 2022 09:53:34 AM com.mojafirma.MojaKlasa read
INFO: Otwarcie pliku data.txt
```

Zauważ, że automatycznie dołączane są informacje o czasie, a także nazwie wywołującej klasy i metody.

Aby wyłączyć te komunikaty informacyjne podczas wdrażania programu, należy skonfigurować backend. W wypadku backendu `java.util.logging` przygotuj plik *logging.properties* o takiej zawartości:

```
handlers=java.util.logging.ConsoleHandler
com.mojafirma.mojaaplikacja.level=WARNING
```

Następnie uruchom aplikację następująco:

```
java -Djava.util.logging.config.file=logging.properties com.mojafirma.mojaaplikacja.Main
```

Ponieważ poziom `INFO` jest niższy niż `WARNING`, komunikat już więcej się nie pokaże.

Interfejs API służący do pobrania mechanizmu rejestrującego i zarejestrowania komunikatów jest częścią frontendu — w tym wypadku interfejsu Platform Logging API. Jeśli korzystasz z innego frontendu, interfejs API będzie odmienny.

Miejsce docelowe komunikatu, jego formatowanie i filtrowanie, a także mechanizmy konfiguracyjne są częścią backendu — tutaj `java.util.logging`. Jeśli korzystasz z innego backendu, skonfiguruj je według dostępnych instrukcji.

5.3.3. Interfejs Platform Logging API

W poprzednim punkcie była mowa o tym, że każdy rejestrowany komunikat ma przypisany pewien poziom. Wyliczenie `System.Logger.Level` zawiera takie wartości, uporządkowane według malejącej ważności: `ERROR`, `WARNING`, `INFO`, `DEBUG` i `TRACE`.

Dzięki instrukcji `import`:

```
import static java.lang.System.Logger.Level.*;
```

zapis poziomów można skrócić:

```
logger.log(INFO, "Opening file " + filename);
    // Zamiast System.Logger.Level.INFO
```

W powyższym przykładzie komunikat `"Otwarcie pliku " + nazwaPliku` jest tworzony nawet wtedy, gdy rejestrowanie danych jest wyłączone. Jeśli problemem jest koszt utworzenia komunikatu, możesz w tym miejscu użyć wyrażenia lambda:

```
logger.log(INFO,() -> "Otwarcie pliku " + nazwaPliku);
```

Wówczas treść komunikatu jest wyznaczana tylko wtedy, gdy jest on faktycznie rejestrowany.

Często spotykane jest rejestrowanie wyjątków:

```
catch (IOException ex) {
    logger.log(WARNING, "Nie można otworzyć pliku " + nazwaPliku, ex);
```

Komunikaty można sformatować przy użyciu klasy `MessageFormat`, którą poznasz w rozdziale 13.:

```
logger.log(WARNING, "Nie można otworzyć pliku {0}", nazwaPliku);
```

Rejestrowane komunikaty można przetłumaczyć na różne języki przy użyciu mechanizmu pakietów z zasobami, który również będzie przedstawiony w rozdziale 13.

W tym celu należy określić pakiet i klucz ciągu formatowania:

```
logger.log(WARNING, bundle, "file.bad", filename);
    // Wyszukuje w pakiecie klucz file.bad
```

Opcjonalnie dostęp do mechanizmu rejestrującego możesz uzyskać następująco:

```
System.Logger logger = System.getLogger("com.mojafirma.mojaaplikacja", bundle);
```

Nie musisz wtedy przekazywać pakietu w każdym wywołaniu.

Obsługiwane są nie wszystkie, a tylko niektóre kombinacje wspomnianych funkcji (odroczone wyznaczanie treści komunikatu, dodawanie obiektu `Throwable`, formatowanie, użycie pakietów). Cały interfejs API przedstawia tabela 5.1.

5.3.4. Konfiguracja mechanizmów rejestrowania danych

Przejdźmy teraz do backendu rejestrowania. Jak już wspominano, domyślnym backendem interfejsu Platform Logging API jest `java.util.logging`. Informacje zawarte w kolejnych punktach będą specyficzne dla tego backendu.

Możesz zmieniać różne właściwości systemu rejestrowania danych poprzez edycję pliku konfiguracyjnego. Domyślny plik konfiguracyjny znajduje się w JDK, w katalogu *conf/logging.properties*. Aby użyć innego pliku, zapisz lokalizację pliku we właściwości `java.util.logging.config.file`, uruchamiając aplikację z parametrem:

```
java -Djava.util.logging.config.file=plikKonfiguracyjny MainClass
```

Wywołanie `System.setProperty("java.util.logging.config.file", plikKonfiguracyjny)` w metodzie `main` nie przyniesie efektu, ponieważ program zarządzający rejestrowaniem danych jest inicjalizowany podczas uruchamiania maszyny wirtualnej, przed wykonaniem procedury `main`.

Tabela 5.1. *Interfejs API klasy System.Logger*

Metoda	Opis
`String getName()`	Nazwa mechanizmu rejestrującego
`boolean isLoggable(System.Logger.Level level)`	Zwraca `true`, jeśli dany mechanizm rejestrujący przetwarza zapisywane pliki na podanym poziomie
`void log(System.Logger.Level level, String msg)`	Rejestruje w pliku podany komunikat
`void log(System.Logger.Level level, Object obj)`	Rejestruje w pliku komunikat uzyskany metodą `obj.toString()`
`void log(System.Logger.Level level, String msg, Throwable thrown)`	Rejestruje w pliku podany komunikat i obiekt `Throwable`
`void log(System.Logger.Level level, Supplier<String> msgSupplier)`	Jeśli dany mechanizm rejestrujący przetwarza zapisywane pliki na podanym poziomie, metoda ta wywołuje dostawcę i rejestruje w pliku wynik
`void log(System.Logger.Level level, Supplier<String> msgSupplier, Throwable thrown)`	Jeśli dany mechanizm rejestrujący przetwarza zapisywane pliki na podanym poziomie, metoda ta wywołuje dostawcę i rejestruje w pliku wynik i obiekt `Throwable`
`void log(System.Logger.Level level, String format, Object... params)`	Jeśli dany mechanizm rejestrujący przetwarza zapisywane pliki na podanym poziomie, metoda ta rejestruje w pliku sformatowany komunikat z przekazanymi parametrami
`void log(System.Logger.Level level, ResourceBundle bundle, String key, Throwable thrown)`	Rejestruje w pliku zawarty w pakiecie komunikat odpowiadający podanemu kluczowi wraz z obiektem `Throwable`
`void log(System.Logger.Level level, ResourceBundle bundle, String key, Object... params)`	Jeśli dany mechanizm rejestrujący przetwarza zapisywane pliki na podanym poziomie, metoda ta wyszukuje w pakiecie format dla podanego klucza i rejestruje w pliku sformatowany komunikat z przekazanymi parametrami

Aby zmienić domyślny poziom rejestrowania danych, należy w pliku konfiguracyjnym zmodyfikować wiersz

```
.level=INFO
```

Możesz określić poziomy logowania dla swoich własnych mechanizmów, dodając wiersze takie jak

```
com.mojafirma.mojaaplikacja.level=WARNING
```

W tym wypadku dodaj `.level` do nazwy mechanizmu logowania.

Z przyczyn historycznych w interfejsie Platform Logging API i we frameworku `java.util.logging` niektóre poziomy mają inne nazwy. Odpowiadające sobie poziomy przedstawiono w tabeli 5.2.

Tabela 5.2. *Odpowiadające sobie poziomy rejestrowania danych platformy i Java Logging Framework*

Platform Logging API	java.util.logging
`ERROR`	`SEVERE`
`WARNING`	`WARNING`
`INFO`	`INFO`
`DEBUG`	`FINE`
`TRACE`	`FINER`

Podobnie jak w przypadku nazw pakietów, nazwy mechanizmów rejestrujących dane są hierarchiczne. W rzeczywistości ich hierarchizacja jest nawet silniejsza. Nie ma semantycznego związku pomiędzy pakietem i jego pakietem nadrzędnym, a mechanizmy logowania wyższego rzędu i niższego rzędu mają wspólne wybrane właściwości. Na przykład: jeśli wyłączysz zapisywanie komunikatów w mechanizmie `"com.mojafirma"`, zapisywanie komunikatów w mechanizmach niższego rzędu również zostanie zatrzymane.

Jak zobaczysz w kolejnym punkcie, mechanizmy logowania w rzeczywistości nie wysyłają komunikatów na konsolę — jest to zadanie dodatkowych programów obsługujących (ang. *handlers*). Również one są przypisane do poziomów. Aby zobaczyć komunikaty poziomu `DEBUG/FINE` na konsoli, musisz także ustawić

```
java.util.logging.ConsoleHandler.level=FINE
```

Ustawienia w konfiguracji programu zarządzającego mechanizmami rejestrowania danych nie są właściwościami systemowymi. Uruchomienie programu z parametrem `-Dcom.mojafirma.mojaaplikacja.level=FINE` nie wpłynie w żaden sposób na mechanizm rejestrujący dane.

Możliwe jest również zmodyfikowanie poziomów rejestrowania danych w działającym programie za pomocą programu `jconsole`. Szczegóły znajdziesz pod adresem *http://www.oracle.com/technetwork/articles/java/jconsole-1564139.html#LoggingControl*.

5.3.5. Programy obsługujące rejestrowanie danych

W domyślnej konfiguracji mechanizmy rejestrujące dane przesyłają rekordy do klasy `ConsoleHandler`, która wysyła je do strumienia `System.err`. Mówiąc dokładniej, mechanizm rejestrujący dane przesyła rekord do nadrzędnego programu obsługującego, a ten bezpośredni przodek (o nazwie `""`) ma `ConsoleHandler`.

Tak jak mechanizmy rejestrujące dane, programy obsługujące mają przypisane poziomy logowania. Aby rekord został zapisany, przypisany mu poziom musi być powyżej progu zarówno mechanizmu rejestrującego (ang. *logger*) jak i programu obsługującego (ang. *handler*). Plik konfiguracyjny menedżera rejestrowania danych ustawia poziom logowania domyślnego programu obsługującego konsolę

```
java.util.logging.ConsoleHandler.level=INFO
```

Aby zapisywać rekordy z poziomu `FINE`, należy zmienić domyślny poziom zarówno mechanizmu rejestrującego, jak i programu obsługującego w konfiguracji.

Aby przesyłać zapisywane rekordy w inne miejsce, należy dodać inny program obsługujący. API udostępnia w tym celu dwa programy obsługujące: `FileHandler` i `SocketHandler`. Program `SocketHandler` wysyła rekordy pod określony adres sieciowy i numer portu. Bardziej interesujący jest `FileHandler`, który zapisuje rekordy w pliku.

Możesz po prostu przesyłać rekordy do domyślnego programu obsługującego w taki sposób:

```
var handler = new FileHandler();
logger.addHandler(handler);
```

Rekordy są przesyłane do pliku *javan.log* w katalogu domowym użytkownika, gdzie ***n*** to liczba gwarantująca unikalność nazwy pliku. Domyślnie rekordy są zapisywane w formacie XML. Typowy rekord rejestrowanych danych jest postaci:

```
<record>
  <date>2014-08-04T09:53:34</date>
  <millis>1407146014072</millis>
  <sequence>1</sequence>
  <logger>com.mojafirma.mojaaplikacja</logger>
  <level>INFO</level>
```

```
  <class>com.horstmann.corejava.Employee</class>
  <method>read</method>
  <thread>10</thread>
  <message>Otwarcie pliku staff.txt</message>
</record>
```

Możesz zmienić domyślne zachowanie programu zapisującego dane do pliku, modyfikując różne parametry w konfiguracji menedżera rejestrowania danych (patrz tabela 5.3) lub używając jednego z poniższych konstruktorów:

```
FileHandler(String wzorzec)
FileHandler(String wzorzec, boolean czyDopisać)
FileHandler(String wzorzec, int limit, int licznik)
FileHandler(String wzorzec, int limit, int licznik, boolean czyDopisać)
```

W tabeli 5.3 opisane jest znaczenie parametrów konstruktorów.

Tabela 5.3. *Parametry konfiguracyjne programu zapisującego dane w plikach*

Właściwość	Opis	Wartość domyślna
`java.util.logging.FileHandler.level`	Poziom programu obsługującego	`Level.ALL`
`java.util.logging.FileHandler.append`	Jeśli `true`, zapisywane rekordy są dopisywane do istniejącego pliku; w przeciwnym wypadku przy każdym uruchomieniu programu tworzony jest nowy plik	`false`
`java.util.logging.FileHandler.limit`	Przybliżona maksymalna liczba bajtów do zapisania w pliku przed utworzeniem kolejnego (`0` = brak ograniczenia)	`0` w klasie `FileHandler`, `50000` w domyślnej konfiguracji menedżera rejestrowania danych
`java.util.logging.FileHandler.pattern`	Wzorzec nazwy pliku (patrz tabela 5.4)	`%h/java%u.log`
`java.util.logging.FileHandler.count`	Maksymalna liczba plików przy rotacji	1 (brak rotacji)
`java.util.logging.FileHandler.filter`	Filtr do selekcji zapisywanych rekordów (patrz punkt 5.3.6)	Brak filtrowania
`java.util.logging.FileHandler.encoding`	Kodowanie znaków	Kodowanie używane na platformie
`java.util.logging.FileHandler.formatter`	Formatowanie rejestrowanych rekordów danych	`java.util.logging.XMLFormatter`

Prawdopodobnie nie zechcesz korzystać z domyślnej nazwy pliku z rejestrowanymi danymi. Użyj wzorca takiego jak `%h/myapp.log` (wyjaśnienie znaczenia zmiennych wzorca znajduje się w tabeli 5.4).

Jeśli wiele aplikacji (lub wiele kopii tej samej aplikacji) używa tego samego pliku do rejestrowania danych, powinieneś ustawić flagę `append`. Opcjonalnie użyj `%u` we wzorcu nazwy pliku, tak by każda aplikacja tworzyła unikalną kopię pliku z rejestrowanymi danymi.

Dobrym pomysłem jest też włączenie rotacji plików. Pliki z rejestrowanymi danymi są zachowywane w sekwencji takiej jak *myapp.log.0*, *myapp.log.1*, *myapp.log.2* itd. Gdy zostanie przekroczony limit liczby plików w sekwencji, najstarszy plik jest kasowany, a nazwy pozostałych plików są zmieniane i tworzony jest plik z dopisaną cyfrą 0.

Tabela 5.4. *Zmienne wzorca nazwy pliku*

Zmienna	Opis
`%h`	Katalog domowy użytkownika (właściwość `user.home`)
`%t`	Systemowy katalog tymczasowy
`%u`	Unikalna liczba
`%g`	Numer pliku przy rotacji (dopisywany jest przyrostek `.%g`, jeśli włączona zostanie rotacja, a wzorzec nie zawiera `%g`)
`%%`	Znak procent

5.3.6. Filtry i formaty

Poza filtrowaniem za pomocą poziomów logowania każdy program obsługujący może mieć zdefiniowane dodatkowe filtry implementujące interfejs `Filter`, który jest interfejsem funkcyjnym z metodą

```
boolean isLoggable(LogRecord record)
```

Aby zainstalować filtr w programie obsługującym, dodaj do konfiguracji mechanizmów zapisywania danych taki element:

```
java.util.logging.ConsoleHandler.filter=com.mojafirma.mojaaplikacja.MyFilter
```

Klasy `ConsoleHandler` i `FileHandler` generują rekordy z danymi w formacie tekstowym i XML. Możesz jednak zdefiniować też własne formaty. Rozszerz klasę `Formatter` i przesłoń metodę

```
String format(LogRecord record)
```

Sformatuj rekord tak, jak chcesz, i zwróć utworzony ciąg znaków. W swojej metodzie formatującej możesz uzyskać informacje na temat obiektu klasy `LogRecord` dzięki wywołaniu jednej z metod podanych w tabeli 5.5.

Tabela 5.5. *Metody pobierające właściwości obiektu klasy LogRecord*

Metoda	Właściwość
`Level getLevel()`	Poziom rejestrowania danych dla rekordu
`String getLoggerName()`	Nazwa mechanizmu, który rejestruje dany rekord
`ResourceBundle getResourceBundle()` `String getResourceBundleName()`	Pakiet zasobów (lub jego nazwa), za pomocą którego ma być zlokalizowany komunikat, a jeśli nie określono żadnego pakietu — wartość `null`
`String getMessage()`	„Surowy” komunikat przed lokalizacją lub formatowaniem
`Object[] getParameters()`	Obiekty parametrów lub, jeśli żadnego nie podano, wartość `null`
`Throwable getThrown()`	Wyrzucany obiekt lub, jeśli go nie podano, wartość `null`
`String getSourceClassName()` `String getSourceMethodName()`	Lokalizacja kodu, który zarejestrował dany rekord. Informacja ta może być dostarczona przez kod rejestrujący dane albo wyciągnięta automatycznie ze stosu środowiska uruchomieniowego. Może być niepoprawna, jeśli kod rejestrujący dane dostarczył błędną wartość albo jeśli uruchomiony kod został tak zoptymalizowany, że nie da się wywnioskować jego dokładnej lokalizacji
`Instant getInstant()`	Czas utworzenia
`long getSequenceNumber()`	Unikalny numer sekwencyjny danego rekordu
`long getLongThreadID()`	Unikalny identyfikator wątku, w którym utworzony został dany rekord. Identyfikatory te są przydzielane przez klasę `LogRecord` i nie mają związku z innymi identyfikatorami wątków

W swojej metodzie formatującej możesz chcieć wywołać metodę

```
String formatMessage(LogRecord record)
```

Metoda ta formatuje komunikat zapisany w rekordzie, wyszukuje klucz komunikatu w pakiecie z zasobami i wypełnia parametrami strukturę komunikatu.

Wiele formatów plików (takich jak XML) wymaga nagłówka i ogona otaczających formatowane rekordy. Aby to uzyskać, przesłoń metody:

```
String getHead(Handler h)
String getTail(Handler h)
```

Na koniec ustaw formatowanie w konfiguracji mechanizmów zapisywania danych:

```
java.util.logging.FileHandler.formatter=com.mycompany.myapp.MyFormatter
```

Ćwiczenia

1. Napisz metodę `public ArrayList<Double> readValues(String filename) throws ...`, która odczyta plik zawierający liczby zmiennoprzecinkowe. Wyrzuć odpowiednie wyjątki, jeśli nie będzie możliwe otwarcie pliku lub jeśli trafisz na dane niebędące liczbami zmiennoprzecinkowymi.
2. Napisz metodę `public double sumOfValues(String filename) throws ...` wywołującą poprzednią metodę i zwracającą sumę wartości w pliku. Przekazuj wszystkie wyjątki do kodu wywołującego tworzoną metodę.
3. Napisz program wywołujący poprzednią metodę i wyświetlający wynik. Przechwyć wyjątki i dostarcz użytkownikowi informacje na temat błędów.
4. Powtórz poprzednie ćwiczenie, ale bez użycia wyjątków. Zamiast tego niech `readValues` i `sumOfValues` zwracają jakieś kody błędów.
5. Zaimplementuj metodę zawierającą kod klas `Scanner` i `PrintWriter` z punktu 5.1.5, „Wyrażenie try z określeniem zasobów”. Nie używaj jednak wyrażenia `try` z zadeklarowanymi zasobami. Zamiast tego użyj zwykłych klauzul `catch`. Upewnij się, że zamykasz oba obiekty, jeśli zostały poprawnie utworzone. Musisz wziąć pod uwagę następujące sytuacje:
   - konstruktor klasy `Scanner` wyrzuca wyjątek;
   - konstruktor klasy `PrintWriter` wyrzuca wyjątek;
   - metoda `hasNext`, `next` lub `println` wyrzuca wyjątek;
   - `out.close()` wyrzuca wyjątek;
   - `in.close()` wyrzuca wyjątek.
6. Punkt 5.1.6, „Klauzula finally”, zawiera przykład błędnego wyrażenia `try` z klauzulami `catch` i `finally`. Popraw ten kod poprzez: a) przechwytywanie wyjątku w klauzuli `finally`, b) wyrażenie `try/catch` zawierające zagnieżdżone wyrażenie `try/catch`, c) wyrażenie `try` z zadeklarowanymi zasobami i klauzulę `catch`.
7. Wyjaśnij, dlaczego rozwiązanie:

```
try (var in = new Scanner(Path.of("/usr/share/dict/words"));
        var out = new PrintWriter(outputFile)) {
    while (in.hasNext())
        out.println(in.next().toLowerCase());
}
```

jest lepsze niż:

```
var in = new Scanner(Path.of("/usr/share/dict/words"));
var out = new PrintWriter(outputFile);
try (in; out) {
    while (in.hasNext())
        out.println(in.next().toLowerCase());
}
```

8. Na potrzeby tego ćwiczenia będziesz musiał przeczytać kod źródłowy klasy `java.util.Scanner`. Jeśli podczas korzystania z klasy `Scanner` pojawi się problem z wczytywanymi danymi, klasa ta przechwytuje wyjątek dotyczący danych wejściowych i zamyka zasób, z którego pobiera dane. Co się stanie, jeśli przy zamykaniu zasobu zostanie wyrzucony wyjątek? Jak ta implementacja współpracuje z obsługą wyciszonych wyjątków w wyrażeniu `try` z zadeklarowanymi zasobami?
9. Zaprojektuj metodę pomocniczą, która pozwoli skorzystać z `ReentrantLock` w wyrażeniu `try` z zadeklarowanymi zasobami. Wywołaj `lock` i zwróć obiekt `AutoCloseable`, którego metoda `close` wywołuje `unlock` i nie wyrzuca wyjątków.
10. Metody klas `Scanner` i `PrintWriter` nie wyrzucają wyjątków kontrolowanych, aby ułatwić korzystanie z nich początkującym programistom. Jak ustalisz, czy błędy pojawiły się podczas odczytu lub zapisu? Zauważ, że konstruktory *mogą* wyrzucać wyjątki kontrolowane. Dlaczego rujnuje to plan uczynienia klas łatwiejszymi w użyciu dla początkujących.
11. Napisz rekurencyjną metodę `factorial`, w której wyświetlisz wszystkie ramki stosu przed zwróceniem wartości. Utwórz (ale nie wyrzucaj) obiekt wyjątku dowolnego rodzaju i pobierz ślad jego stosu w sposób opisany w punkcie 5.1.8, „Nieprzechwycone wyjątki oraz ślad stosu wywołań”.
12. Porównaj wykorzystanie `Objects.requireNonNull(obj)` i `assert obj != null`. Podaj przykład odpowiedniego użycia obu tych wyrażeń.
13. Napisz metodę `int min(int[] values)`, która przed zwróceniem najmniejszej wartości sprawdza dodatkowo za pomocą asercji, czy rzeczywiście ta wartość jest mniejsza albo równa wszystkim wartościom w tablicy. Użyj metody pomocniczej lub, jeśli zajrzałeś już do rozdziału 8., metody `Stream.allMatch`. Wywołuj metodę w pętli dla dużej tablicy i zmierz czas wykonania kodu z asercją włączoną, wyłączoną i usuniętą.
14. Zaimplementuj i przetestuj filtr rekordów rejestrowanych danych, który odrzuci rekordy zawierające brzydkie słowa, takie jak *seks*, *narkotyki* i *C++*.
15. Zaimplementuj i przetestuj kod formatujący rekordy rejestrowanych danych do pliku JSON.

Java: poznaj najnowsze mechanizmy i paradygmaty!

Java jest dojrzałym, rozbudowanym, wszechstronnym, a przy tym starannie zaprojektowanym i bezpiecznym językiem programowania. W jego nowej, 17. wersji wprowadzono sporo usprawnień podstawowych elementów języka i interfejsów API. Wiele starych idiomów straciło przydatność, a nowe mechanizmy i paradygmaty pozwalają zwiększyć efektywność tworzenia kodu. Zawodowy programista musi dobrze poznać te nowości i nauczyć się z nich korzystać.

Oto zaktualizowane i uzupełnione wydanie zwięzłego wprowadzenia do Javy SE 17, przeznaczonego dla profesjonalnych programistów Javy. Przedstawiono w nim wszystkie istotne zagadnienia, łącznie z takimi koncepcjami jak wyrażenia lambda i strumienie, nowoczesnymi konstrukcjami, jak rekordy i klasy zapieczętowane, a także zaawansowanymi technikami programowania współbieżnego. Książka została pomyślana w taki sposób, aby wszystkie ważne zmiany wprowadzone w najnowszej wersji Javy zostały zaprezentowane w esencjonalnej formie. Informacje uporządkowano tak, aby zapewnić szybki dostęp i ułatwić zrozumienie potrzebnego zagadnienia. W przewodniku znalazło się też mnóstwo praktycznych informacji wraz z przykładami kodu, pozwalającymi natychmiast przetestować wszystkie nowości.

W książce między innymi:

- rekordy i klasy zapieczętowane
- programowanie funkcyjne z wyrażeniami lambda
- optymalne zarządzanie danymi
- nowoczesne funkcje biblioteczne i bezpieczne struktury danych
- zmodularyzowane API języka Java
- interfejsy API służące do przetwarzania daty i czasu oraz internacjonalizacji

CAY S. HORSTMANN — autor najpopularniejszych w Polsce podręczników do nauki Javy. Jest emerytowanym profesorem informatyki na Uniwersytecie Stanowym w San José, został wyróżniony tytułem Java Champion. Często występuje na konferencjach poświęconych zagadnieniom programistycznym. Urodził się w północnych Niemczech, obecnie mieszka w USA.

Helion
helion.pl
HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl

KOD KORZYŚCI
Sięgnij po więcej!

ISBN 978-83-289-0140-7

Cena: 89,00 zł

Pearson
Addison-Wesley