Struktury danych i techniki obiektowe na przykładzie Javy 5.0

Autorzy: Elliot Koffman, Paul Wolfgang
Tłumaczenie: Rafał Jońca, Tomasz Żmijewski
ISBN: 83-246-0089-2
Tytuł oryginału: Objects, Abstraction, Data Structures and Design: Using Java version 5.0
Format: B5, stron: 936

Przy tworzeniu systemów informatycznych najważniejsze zadania wykonuje się, zanim powstanie pierwszy fragment kodu źródłowego. Wymogi stawiane współczesnym aplikacjom powodują, że inżynieria oprogramowania staje się kwestią kluczową. Opracowanie odpowiedniego projektu oraz właściwy dobór technologii i metodologii zapewniają szybką i efektywną pracę nad systemem. Niezwykle ważne jest poznanie dostępnych w języku Java struktur danych i umiejętność ich wykorzystania. Prawidłowo dobrana struktura danych znacznie przyspiesza nie tylko implementację aplikacji, ale również działanie gotowego systemu.

Książka „Struktury danych i techniki obiektowe na przykładzie Javy 5.0" przedstawia podstawowe struktury danych i sposoby ich wykorzystania podczas programowania obiektowego. Wszystkie wiadomości zostały zaprezentowane z uwzględnieniem reguł nowoczesnej inżynierii oprogramowania. Czytając kolejne rozdziały książki, poznasz najlepsze zastosowania różnych struktur danych oraz wady i zalety ich implementacji. Przede wszystkim jednak zrozumiesz potrzebę stosowania tak wielu struktur danych.

- Cykl życia oprogramowania
- Zastosowanie języka UML w projektowaniu systemów
- Obsługa błędów i wyjątków
- Testowanie oprogramowania
- Dziedziczenie i hierarchia klas
- Listy jedno- i dwukierunkowe
- Interfejs Collection
- Stosy i kolejki
- Algorytmy rekurencyjne
- Sortowanie danych
- Drzewa wyszukiwania
- Grafy

Po przeczytaniu tej książki zrozumiesz zasadę:
„Pomyśl, a dopiero potem pisz kod"

Wydawnictwo Helion
ul. Chopina 6
44-100 Gliwice
tel. (32)230-98-63
e-mail: helion@helion.pl

Spis treści

Rozdział 2.

Poprawność i wydajność programów

Cele dydaktyczne

- Zrozumieć różnicę między trzema kategoriami błędów w programach.
- Zrozumieć efekty nie wyłapania wyjątku i poznać powody, dla których warto wyłapywać wyjątki.
- Zapoznać się z hierarchią `Exception` i zrozumieć różnicę między wyjątkami weryfikowanymi i nieweryfikowanymi.
- Nauczyć się korzystania z bloku `try-catch-finally` w celu wyłapywania i przetwarzania wyjątków.
- Zrozumieć, co to znaczy zgłosić wyjątek, i w jaki sposób zgłosić wyjątek w metodzie.
- Zrozumieć różne podejścia do testowania i nauczyć się je stosować w zależności od aktualnej sytuacji.
- Dowiedzieć się, jak pisać specjalne metody testujące inne metody i klasy.
- Zapoznać się z technikami poszukiwania błędów i programami uruchomieniowymi.
- Zapoznać się z procesem weryfikacji programów, a także stosowaniem asercji i niezmienników pętli.
- Zrozumieć znaczenie notacji złożoności obliczeniowej algorytmów i zacząć używać jej do analizy wydajności algorytmów.

Niniejszy rozdział omawia poprawność i wydajność programów. Przedstawiamy w nim metody znajdowania defektów w aplikacjach i rozwiązania, dzięki którym uda się uniknąć popełniania błędów: uważne projektowanie programu i wykorzystywanie innych członków zespołu do odnajdowania błędów logicznych, zanim znajdą się w kodzie. Podobnie jak w wielu innych życiowych sytuacjach wczesne wykrycie problemu prowadzi do najlepszych wyników.

Prześledzimy kilka rodzajów błędów, które mogą pojawić się w programach: błędy składniowe, błędy czasu wykonania i wyjątki oraz błędy logiczne. Pokażemy różne rodzaje wyjątków języka Java, różne sposoby ich obsługi, a także korzyści płynące z ich stosowania.

Rozdział do pewnego stopnia zajmuje się też testowaniem programów. Przedstawimy sposoby generowania odpowiedniego planu testów oraz różnice między testami jednostkowymi i integracyjnymi w odniesieniu do projektowania obiektowego. Zobrazujemy również zastosowanie programów testujących (ang. *test drivers*) i namiastek — specjalnych metod napisanych do testowania innych metod i klas.

Przedstawimy techniki wykrywania błędów za pomocą programów uruchomieniowych oraz sposoby generowania informacji diagnostycznych. Pokażemy, które elementy zintegrowanych środowisk programistycznych pozwalają szybciej przeprowadzić proces znajdowania błędów.

Opiszemy, w jaki sposób stosować weryfikację formalną do sprawdzania logicznej poprawności programu. Nie oczekujemy pisania programów z matematyczną dokładnością, ale mamy nadzieję, iż przedstawiane pomysły dotyczące projektowania i implementacji pozwolą zwiększyć niezawodność najważniejszych elementów programów.

Ostatnia część rozdziału została poświęcona na omówienie wydajności algorytmów i sposobów ich kategoryzacji. Przedstawimy notację związaną ze złożonością obliczeniową algorytmów, która pozwala w sposób względny porównać szybkość działania różnych algorytmów.

Poprawność i wydajność programów

2.1. Defekty i błędy w programach

Niniejszy podrozdział dotyczy błędów w programach i sposobów ich unikania. To, iż program działa wydajnie, nie ma dużego znaczenia, gdy nie wykonuje swych zadań poprawnie. Bardzo często defekty zaczynają występować dopiero po dostarczeniu produktu klientowi, co czasem ma bardzo niemiłe konsekwencje. Kilka bardziej znanych defektów oprogramowania spowodowało problemy z dostarczaniem zasilania, przeciążenie

sieci telefonicznej, utratę statku kosmicznego, a także wypadki samochodowe spowodowane nieprawidłowym działaniem komputera pokładowego.

Jednym ze sposobów sprawdzania poprawności programów jest ich intensywne testowanie. Niestety, często bardzo trudno określić, ile potrzeba testów, by uznać program za wystarczająco pewny. Trzeba pogodzić się z faktem, iż testowanie nigdy nie pozwoli uznać, że program na 100% jest pozbawiony błędów. Co gorsza, w pewnych sytuacjach nie można nawet przetestować oprogramowania we wszystkich warunkach (dotyczy to na przykład systemów naprowadzania pocisków lub systemów zabezpieczających przez stopieniem się reaktora nuklearnego).

Termin „debugowanie" służy często do określenia sytuacji, w której poszukuje się powodów generowania przez program błędnych wyników. Program uruchomieniowy to najczęściej stosowane narzędzie do znajdowania defektów (patrz podrozdział 2.6). Ponieważ społeczeństwa w coraz większym stopniu uzależniają się od systemów komputerowych, wielu profesjonalistów wierzy, iż nazywanie błędów w programach „defektami" skutkuje zbyt frywolnym podchodzeniem do ich skutków.

Uważne projektowanie i testowanie pozwala znacząco zmniejszyć liczbę błędów. Oczywiście znacznie prościej ustrzec się błędów na etapie projektowania, niż poprawiać je później w fazie testów.

Istnieją trzy podstawowe rodzaje błędów:

- błędy składniowe,
- błędy czasu wykonania i wyjątki,
- błędy logiczne.

Błędy składniowe

Błąd składniowy to pomyłka w zastosowaniu zasad gramatycznych (składniowych) języka programowania Java (na przykład zastosowanie operatora przypisania = zamiast operatora równości ==). Kompilator języka Java wykrywa większość tego rodzaju pomyłek i wymusza ich poprawienie przed udaną kompilacją programu. Poniżej przedstawiamy przykłady kilku podstawowych błędów składniowych:

- pominięcie lub złe umieszczenie nawiasów otaczających złożone polecenia,
- przeprowadzenie niewłaściwej operacji na typie podstawowym (na przykład zastosowanie dodawania dla typu `boolean` lub przypisanie wartości rzeczywistej do zmiennej typu `int`),
- wywołanie metody, która nie stanowi części definicji obiektu,
- niezadeklarowanie zmiennej przed jej użyciem,
- wprowadzenie kilku deklaracji zmiennej,
- nieprzypisanie wartości do zmiennej lokalnej przed skorzystaniem z niej,
- niezwrócenie wartości z metody, której typ wynikowy jest inny niż `void`.

Pewne błędy składniowe wynikają z błędów typograficznych (na przykład przeinaczenie nazwy zmiennej lub wpisanie nawiasu { zamiast } lub]).

Kompilator nie wykrywa wszystkich błędów typograficznych. Jeśli na przykład pominie się znak } tam, gdzie powinien się znaleźć, a zamiast tego wstawi się go kilka wierszy niżej, składnia programu będzie poprawna, choć inna od zamierzonej. Ponieważ programy poprawne syntaktycznie mogą zawierać błędy innego rodzaju, trzeba je bardzo dokładnie testować.

Błędy czasu wykonania i wyjątki

Błędy czasu wykonania, jak sama nazwa wskazuje, występują w trakcie wykonywania programu. Tego rodzaju błędy występują, gdy maszyna wirtualna Javy wykryje sytuację, o której wie, że nie jest prawidłowa. Tabela 2.1 zawiera przykłady niektórych błędów czasu wykonania. Błąd wykonania powoduje **zgłoszenie wyjątku** przez maszynę wirtualną — powstaje obiekt klasy wyjątków, który identyfikuje niepoprawną operację i doprowadza do przerwania podstawowego toku przetwarzania. Jest to w zasadzie sytuacja typu „dobra wiadomość, zła wiadomość". Dobra wiadomość to fakt wykrycia błędu. Zła wiadomość to przerwanie wykonywania głównej ścieżki programu. Wyjątki zostaną szczegółowo opisane w podrozdziałach 2.2 i 2.3, które zawierają także rady, jak uniknąć błędów. Poniżej znajdują się krótkie omówienia wraz z przykładami wyjątków wymienionych w tabeli 2.1.

Tabela 2.1. *Podklasy klasy java.lang.RuntimeException*

Wyjątek czasu wykonania	Powód lub konsekwencja
`ArithmeticException`	Dzielenie liczby całkowitej przez 0.
`ArrayIndexOutOfBoundsException`	Próba dostępu do elementu tablicy, którego indeks jest mniejszy od 0 albo większy lub równy rozmiarowi tablicy.
`IllegalArgumentException`	Próba wywołania metody z argumentem o niepoprawnym typie lub formacie.
`NumberFormatException`	Próba skonwertowania ciągu znaków, który nie zawiera liczby, na liczbę całkowitą lub rzeczywistą.
`NullPointerException`	Próba wykorzystania referencji z wartością `null` do dostępu do obiektu.
`NoSuchElementException`	Próba pobrania następnego tokenu po wydobyciu wszystkich tokenów z analizowanego tekstu.
`InputMismatchException`	Token zwrócony przez metodę `next...` klasy `Scanner` nie odpowiada wzorcowi spodziewanego typu danych.

Dzielenie przez zero

Jeśli zmienna `count` oznacza liczbę przetworzonych elementów i możliwe jest, iż nie zostaną przetworzone żadne elementy, poniższa instrukcja:

```
average = sum / count;
```

może doprowadzić do błędu dzielenia przez zero. Jeśli sum i count są zmiennymi typu int, maszyna wirtualna zgłasza wyjątek ArithmeticException. Bardzo łatwo ochronić się przed tego rodzaju dzieleniem, stosując konstrukcję if sprawdzającą, czy można poprawnie wykonać operację dzielenia przez count.

```
if (count==0)
  average = 0;
else
  average = sum / count;
```

Często wartość średnią (average) oblicza się jako liczbę rzeczywistą (typ double), więc na ogół rzutuje się wartość całkowitą sum na typ double przed dokonaniem dzielenia. W takiej sytuacji wyjątek nie zostanie zgłoszony, jeśli count jest równie 0. Zamiast tego zmienna average przyjmie jedną z wartości specjalnych: Double.POSITIVE_INFINITY, Double.NEGATIVE_INFINITY lub Double.NaN w zależności od tego, czy zawartość zmiennej sum była dodatnia, ujemna czy równa 0.

Indeks tablicy poza zakresem

Wyjątek ArrayIndexOutOfBoundsException zostaje zgłoszony przez maszynę wirtualną Javy, gdy wartość indeksu używanego do uzyskania dostępu do elementu tablicy jest mniejsza od 0 albo większa lub równa rozmiarowi tablicy. Załóżmy, że zdefiniowano tablicę scores w następujący sposób:

```
int[] scores = new int[500];
```

W zapisie scores[i] zmienna i (typu int) określa indeks tablicy. Wyjątek ArrayIndexOutOfBoundsException zostanie zgłoszony, gdy i będzie zawierać wartość mniejszą od 0 lub większą od 499.

Tego rodzaju błędów można uniknąć, bardzo uważnie sprawdzając wartości graniczne dla indeksu, który jest zmienną inkrementowaną w kolejnych iteracjach pętli. Często błędnie jako górną granicę przyjmuje się rozmiar tablicy zamiast wartość o jeden mniejszą od tego rozmiaru.

Przykład 2.1

Poniższa pętla spowoduje zgłoszenie wyjątku ArrayIndexOutOfBoundsException w ostatniej iteracji, gdy i będzie równe x.length:

```
for (int i = 0; i <= x.length; i++)
  x[i] = i * i;
```

Warunek w pętli powinien mieć postać i < x.length.

Jeśli indeks tablicy ma zostać przekazany jako argument metody, metoda ta powinna sprawdzić poprawność argumentu przed jego zastosowaniem. Poniższy przykład obrazuje sposób ochrony przed tego rodzaju błędami. W dalszej części rozdziału zostanie przedstawiona jeszcze inna technika radzenia sobie z niepoprawnymi argumentami.

Przykład 2.2

Metoda `setElementOfX()` zapamiętuje swój drugi argument (`val`) w elemencie tablicy `x` (pole danych typu `int[]`) wybranym dzięki pierwszemu argumentowi (`index`). Instrukcja `if` sprawdza przed przypisaniem, czy wartość `index` znajduje się w poprawnym przedziale. Metoda zwraca wartość typu `boolean` informującą, czy element tablicy uległ zmianie. Jeśli metoda nie sprawdzałyby poprawności indeksu, podane błędnie wartości mogłyby doprowadzić do zgłoszenia wyjątku `ArrayIndexOutOfBoundsException`.

```
/** Zapamiętuje val w elemencie tablicy x określonym argumentem index.
    @param index Indeks elementu do modyfikacji.
    @param val Wartość do zapamiętania.
    @return Zwraca true, jeśli val zostało zapisane, lub false w przeciwnym przypadku.
*/
public boolean setElementOfX(int index, int val) {
  if (index >= 0 && index < x.length) {
    x[index] = val;
    return true;
  } else {
    return false;
  }
}
```

STYL PROGRAMOWANIA

Wykorzystanie wyniku logicznego do wskazania sukcesu lub porażki

Co zyskujemy, zwracając wartość logiczną po wykonaniu metody `setElementOfX()`? Osoba używająca tej metody może wyświetlić komunikat ostrzeżenia, jeśli elementu nie udało się zapisać. Poniższa instrukcja `if` wykonuje metodę `setElementOfX()` obiektu `myXArray` i zapisuje komunikat błędu do standardowego strumienia błędów (`System.err`), jeśli nie udało się przypisać wartości elementowi:

```
if (!myXArray.setElementOfX(sub, data))
  System.err.println("***** Nie udało się przypisać " + data
                     + " do elementu " + sub + " tablicy x");
```

Wartości logiczne były stosowane do wskazania sukcesu lub porażki w wykonaniu metody przed wprowadzeniem wyjątków do języków programowania. Ponieważ wyjątki stanowią część języka Java, zaleca się ich stosowanie do wskazania sytuacji nieprawidłowych.

Przykład 2.3

Typowym sposobem informowania programu o nazwie pliku danych jest przekazywanie nazwy pliku jako parametru do metody `main()`. Jeśli korzysta się z Java SDK, wystarczy po poleceniu uruchamiającym program podać nazwę pliku. Jeżeli stosuje się zintegrowane środowisko programistyczne (IDE), można wpisać parametry w specjalnym oknie. W momencie rozpoczęcia wykonywania metody `main()` wszystkie parametry znajdują się w tablicy `args`.

Poniższa metoda `main()` oczekuje, iż pierwszym parametrem programu będzie nazwa pliku wejściowego, więc sprawdza przy użyciu instrukcji `if`, czy tablica `args` zawiera choć jeden tekst, zanim spróbuje zapamiętać `args[0]` w zmiennej `inputFileName`. Jeżeli parametr nie został podany, dla pliku przyjmowana jest domyślna nazwa `"Ksiazka.dat"`. Bez instrukcji `if` Java zgłosiłaby wyjątek `ArrayIndexOutOfBoundsException`, gdyby nie przekazano żądnych parametrów i spróbowano odczytać wartość `args[0]`.

```
public static void main(String[] args) {
  String inputFileName;

  if (args.length > 0)
    inputFileName = args[0];
  else
    inputFileName = "Ksiazka.dat";
}
```

Wyjątki NumberFormatException i InputMismatchException

Wyjątek `NumberFormatException` zostaje zgłoszony, gdy program próbuje skonwertować ciąg znaków nie reprezentujący liczby (na ogół datę) na postać liczbową. Jeśli użytkownik wpisze tekst `"2.6e"`, metoda `parseDouble()` z poniższego kodu:

```
String speedStr = JOptionPane.showInputDialog("Wpisz prędkość");
double speed = Double.parseDouble(speedStr);
```

zgłosi wyjątek `NumberFormatException`, ponieważ `"2.6e"` nie jest poprawnym tekstem reprezentującym liczbę (po `e` nie występuje wykładnik). Nie istnieje ogólny sposób zapobiegania tego rodzaju błędom, ponieważ nie jest łatwo ochronić się przed wszystkimi możliwymi błędami danych, które może popełnić użytkownik.

Podobny błąd może wystąpić, gdy do pobierania danych używa się obiektu `Scanner`. Jeśli zmienna `scIn` wskazuje na obiekt `Scanner`, polecenie:

```
double speed = scIn.nextDouble();
```

zgłosi wyjątek `InputMismatchException`, jeśli następnym odczytanym tokenem będzie `"2.6e"`.

Wskaźnik równy null

Wyjątek `NullPointerException` zostaje zgłoszony, gdy próbuje się uzyskać dostęp do obiektu, który nie istnieje — zmienna referencyjna zawiera specjalną wartość znaną jako `null`. Najprościej bronić się przed tym wyjątkiem, sprawdzając istnienie wartości `null` przed wywołaniem metody.

Przykład 2.4

Po odczytaniu przez pętlę wszystkich wierszy zmienna `name` zawiera wartość `null`. Zmienna `count` zawiera informację o liczbie wystąpień tekstu zawartego w `specialName` w odczytanym ciągu znaków. Warunek pętli `while` zapamiętuje kolejny wiersz w `name` i sprawdza istnienie wartości `null`. W ten sposób zapewnia się, iż wywołanie

`name.equals(specialName)` wystąpi tylko wtedy, gdy referencja `name` rzeczywiście wskazuje na obiekt `String`.

```
// Zlicza wystąpienia specialName.
int count = 0;
while ((name = in.readLine()) != null) {
  if (name.equals(specialName))
    count++;
}
System.out.println(specialName + " wystąpiło w tekście " + count + " razy.");
```

Błędy logiczne

Błędy logiczne to ostatnia kategoria błędów. Tego rodzaju błąd występuje wtedy, gdy programista lub analityk popełnił błąd w projekcie klasy lub metody albo niepoprawnie zaimplementował algorytm. W każdym z tych przypadków kod Javy nie spełni wymagań stawianych metodzie — będzie się wykonywał, ale uzyskiwane z niego dane nie będą prawidłowe. Jeżeli użytkownik będzie miał szczęście, błąd logiczny spowoduje powstanie błędu czasu wykonania w innej metodzie, która korzysta z wyników błędnej metody. Taka sytuacja nie musi jednak zajść. Większość błędów logicznych nie powoduje błędów składniowych ani wykonania, co znacznie utrudnia ich odnalezienie. Przykład takiego błędu znajduje się w jednej z ramek dotyczących pułapek w poprzednim rozdziale.

Czasem błędy logiczne wykrywa się na etapie testów, uważnie porównując wyniki uzyskane z programu ze spodziewanymi wynikami. Taka sytuacja zajdzie jednak tylko wtedy, gdy tester uwzględni testy związane z działaniem niepoprawnej części programu. Co więcej, tester musi znać poprawne wyniki, jakie powinien zwrócić program, aby móc je porównać z wynikami testów.

Najgorsza sytuacja występuje wtedy, gdy błędy logiczne ujawniają się w trakcie działania programu w docelowym środowisku. Istnieje wiele przykładów wiadomości dziennikarskich, w których błędy logiczne odgrywały główną rolę. Pojazd kosmiczny Mars Lander rozbił się z powodu niejednorodnego zastosowania typów (stóp i metrów). Programy księgowe wysyłały rachunki na niewyobrażalne sumy. Błędy logiczne w programach sterujących pracą bankomatów i systemów zakładów bukmacherskich spowodowały ogromne straty ich właścicieli. Wiele popularnych systemów operacyjnych zostało wydanych z błędami logicznymi umożliwiającymi hakerom stosunkowo łatwe uzyskanie niepowołanego dostępu do danych komputera. Jeden z popularnych edytorów tekstu po integracji z systemem zarządzania dokumentami potrafił „zjeść” dokument z serwera, pozostawiając na jego miejscu pusty plik. W latach 80-tych XX wieku kilku pacjentów zmarło z powodu błędów w oprogramowaniu sterującym pracą maszyn do terapii radiacyjnej ponieważ zaaplikowano im zbyt duże dawki promieniowania. W roku 2004 zdarzyły się dwa wypadki samochodowe spowodowane błędami logicznymi: jeden dotyczył błędu w oprogramowaniu systemu ABS, a drugi w oprogramowaniu systemu zapłonu. Oba błędy zwiększały ryzyko wypadku. Aby uniknąć takich katastrofalnych następstw, programiści i testerzy muszą bardzo dokładnie sprawdzać swoje wyroby pod kątem błędów logicznych.

W podrozdziale 2.5 przedstawimy, w jaki sposób zmniejszyć liczbę błędów logicznych, uważnie sprawdzając projekt programu. Omówimy również wykrywanie tego rodzaju błędów dzięki testom.

ĆWICZENIA DO PODROZDZIAŁU 2.1

SPRAWDŹ SIĘ

1. Wyjaśnij i popraw błędy składniowe w poniższej metodzie `main()`:

```
int x = 3.45;
float w = x / 2;
char ch = 'a' + 'b';
if (x = y) {
  int z = x++;
else
  int z = y++;
}
System.out.println(z + "***" + sqrt(z));
```

2. Powiedz, które z poniższych instrukcji spowodują zgłoszenie wyjątku. Jaki to będzie rodzaj wyjątku?

```
int[] x = new int[10];
x[x.length] = 5;
x[x.length - 1] = -95;
x[0] = 7;
int n = 0;
x[n] = x.length / n;
String num = "344.e5";
n = Integer.parseInt(num);
double y = Double.parseDouble(num);
```

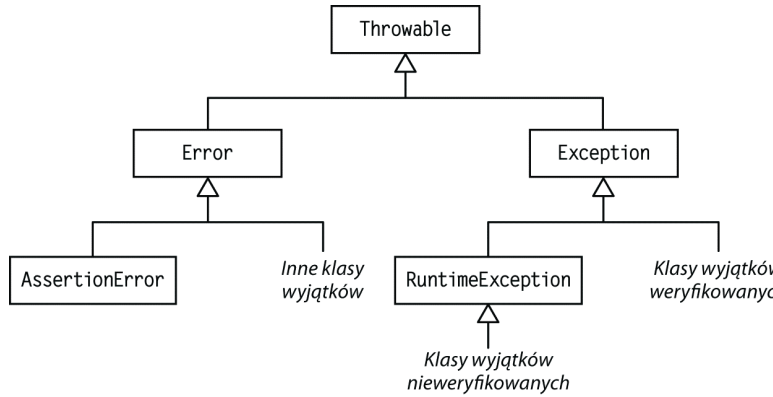
PROGRAMOWANIE

1. Napisz metodę klasy, która zwraca wartość typu `int` zapisaną w tablicowym polu danych `x`. Parametrem metody jest indeks elementu do pobrania. Dokonaj walidacji parametru, by zabezpieczyć się przed wyjątkiem `ArrayIndexOutOfBoundsException`. Zwróć wartość `Integer.MIN_VALUE`, jeśli indeks nie jest poprawny.

2.2. Hierarchia klas wyjątków

W momencie zgłaszania wyjątku tworzony jest egzemplarz jednej z klas wyjątków języka Java. Niniejszy podrozdział omawia hierarchię klas wyjątków.

Wyjątki są zdefiniowane w hierarchii klas, która u swego szczytu zawiera klasę bazową `Throwable` (patrz diagram UML z rysunku 2.1) (szczegółowe omówienie hierarchii klas znajduje się w rozdziale 3.; podrozdział 3.1 zawiera opis terminów podklasa i klasa bazowa). Diagram UML obrazuje, iż klasy `Error` i `Exception` są podklasami klasy



Rysunek 2.1. *Podsumowanie hierarchii klas wyjątków*

`Throwable`. Każda z tych klas zawiera kolejne podklasy (niektóre z nich zostały przedstawione na rysunku). W niniejszym rozdziale skupimy się przede wszystkim na klasie `Exception` i jej podklasach. Ponieważ `RuntimeException` jest podklasą klasy `Exception`, jest również podklasą klasy `Throwable` (związki podklas są przechodnie).

Klasa Throwable

Klasa `Throwable` jest klasą bazową dla wszystkich wyjątków. Metody tej klasy zostały podsumowane w tabeli 2.2. Ponieważ wszystkie inne klasy wyjątków wywodzą się z tej klasy, dziedziczą wszystkie jej metody. Innymi słowy, dowolna klasa wyjątku zawiera metody zdefiniowane w klasie `Throwable`. Jeśli `ex` jest obiektem `Exception`, wywołanie:

```
ex.printStackTrace();
```

wyświetla **stos wywołań** omówiony dokładniej w podrozdziale 2.3. Polecenie:

```
System.err.println(ex.getMessage());
```

wyświetla **szczegółowy komunikat** (błędu) opisujący wyjątek. Polecenie:

```
System.err.println(ex.toString());
```

wyświetla nazwę wyjątku oraz szczegółowy komunikat.

Tabela 2.2. *Podsumowanie najczęściej stosowanych metod klasy java.lang.Throwable*

Metoda	Działania
`String getMessage()`	Zwraca szczegółowy komunikat.
`void printStackTrace()`	Wyświetla stos wywołań na standardowym wyjściu błędów (`System.err`).
`String toString()`	Zwraca nazwę wyjątku oraz szczegółowy komunikat.

Wyjątki weryfikowane i nieweryfikowane

Istnieją dwie kategorie wyjątków: **weryfikowane** i **nieweryfikowane**. Wyjątek weryfikowany to na ogół wyjątek, który nie jest związany z błędem programistycznym i pozostaje poza kontrolą programisty. Do tej kategorii zalicza się wszystkie wyjątki

spowodowane przez błędy wejścia-wyjścia. Jeśli program próbuje uzyskać dostęp do pliku danych, który nie jest dostępny z powodu błędu użytkownika lub systemu, zostaje zgłoszony wyjątek `FileNotFoundException`. Klasa `IOException` i jej podklasy (patrz tabela 2.3) również są wyjątkami weryfikowanymi. Choć ta kategoria wyjątków na ogół pozostaje poza kontrolą programisty, musi on o niej pamiętać i obsługiwać ją w pewien sposób. W dwóch kolejnych podrozdziałach przedstawimy sposoby ich obsługi. Wszystkie weryfikowane wyjątki są podklasami klasy `Exception`, ale nie są podklasami klasy `RuntimeException`.

Tabela 2.3. *Klasa java.io.IOException i kilka jej podklas*

Klasa wyjątku	Powód zgłoszenia
`IOException`	Pewien rodzaj błędu wejścia-wyjścia.
`EOFException`	Próba odczytu danych poza końcem pliku.
`FileNotFoundException`	Nie udało się odnaleźć pliku.

Rysunek 2.2 przedstawia pełniejszy diagram hierarchii dla klasy Exception

Wyjątki nieweryfikowane oznaczają błędy, które są wynikiem pomyłki programisty lub poważnego błędu zewnętrznego, którego na ogół nie udaje się naprawić. Na przykład wyjątki `NullPointerException` lub `ArrayIndexOutOfBoundsException` są wyjątkami nieweryfikowanymi, gdyż na ogół wynikają z błędów programisty. Wszystkie te wyjątki są podklasami klasy `RuntimeException`. Choć można uniknąć niektórych z nich dzięki programowaniu defensywnemu, zapobieganie im wszystkim jest mało praktyczne, gdyż wymaga zapewnienia dla nich odpowiednich procedur obsługi. Można obsługiwać te wyjątki, ale język Java tego nie wymusza.

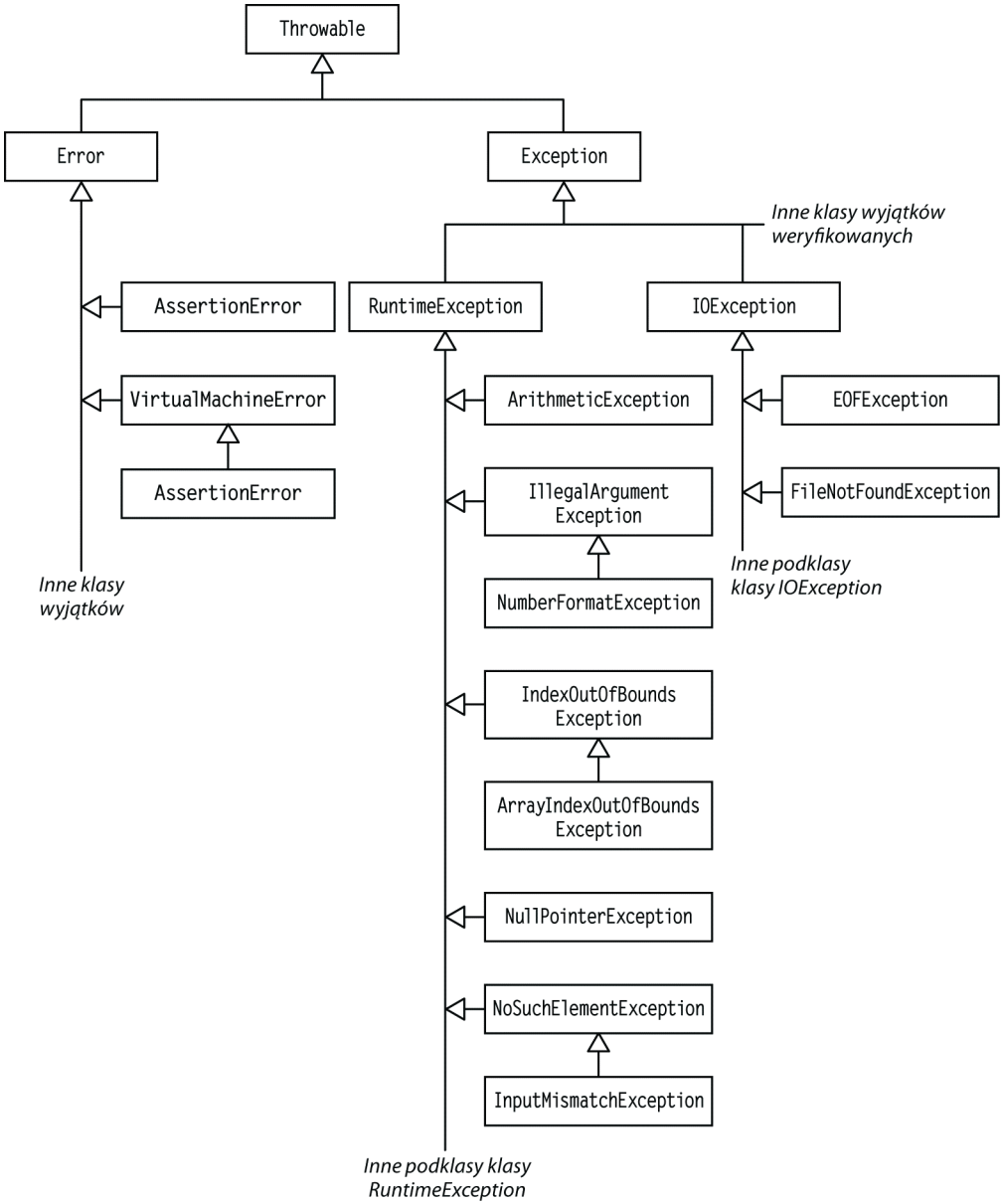
Klasa `Error` i jej podklasy reprezentują błędy związane z poważnymi sytuacjami zewnętrznymi. Przykładem takiego błędu jest wyjątek `OutOfMemoryError`, który zostaje zgłoszony, gdy nie ma wystarczającej ilości wolnej pamięci, by dokonać alokacji. Nie można przewidzieć ani obronić się przed tego rodzaju błędami. Niektórzy starają się obsługiwać te wyjątki, ale nie zalecamy takiego podejścia, gdyż próby poradzenia sobie z nimi na ogół będą nieudane. Jeśli zostaje zgłoszony wspomniany wcześniej wyjątek `OutOfMemoryError`, zapewne nie ma również dostępnej wystarczającej ilości pamięci, by wykonać kod obsługi wyjątku.

Jak można stwierdzić czy dany wyjątek jest weryfikowany czy nieweryfikowany? Wyjątki dziedziczące po klasach `RuntimeException` i `Error` są wyjątkami nieweryfikowanymi. Wszystkie pozostałe klasy wyjątków są weryfikowane.

ĆWICZENIA DO PODROZDZIAŁU 2.2

SPRAWDŹ SIĘ

1. Wyjaśnij podstawową różnicę między wyjątkami weryfikowanymi i nieweryfikowanymi. Podaj po jednym przykładzie każdego z tych wyjątków. Jakich kryteriów używa Java, by określić, czy wyjątek jest weryfikowany czy nieweryfikowany.



Rysunek 2.2. *Hierarchia wyjątków przedstawiająca wybrane wyjątki weryfikowane i nieweryfikowane*

2. Jaka jest różnica między wyjątkiem nieweryfikowanym klasy `Error` a klasy `Exception`?
3. Wymień cztery podklasy klasy `RuntimeException`.
4. Wymień dwie podklasy klasy `IOException`.

2.3. Wyłapywanie i obsługa wyjątków

W poprzednim podrozdziale wspomnieliśmy o tym, że gdy maszyna wirtualna Javy wykryje błąd wykonania, zgłasza wyjątek. Powoduje to przerwanie normalnego toku wykonywania programu, ponieważ wykonanie dalszych poleceń zapewne spowodowałoby powstanie jeszcze poważniejszych błędów. Domyślne zachowanie polega na zatrzymaniu programu i wyświetleniu przez maszynę wirtualną komunikatu błędu wskazującego typ wyjątku i miejsce jego powstania. Programista może zmienić domyślne zachowanie, umieszczając instrukcję mogącą spowodować wyjątek wewnątrz bloku `try` i zapewniając blok `catch` stanowiący procedurę obsługi wyjątku.

Wyjątki niewyłapane

Jeśli wyjątek nie zostanie wyłapany, program zatrzymuje swe działanie, a maszyna wirtualna wyświetla komunikat błędu wraz ze stosem wywołań. Jak sama nazwa wskazuje, stos wywołań zawiera sekwencję wywołań metod, zaczynając od metody, która spowodowała wyjątek, a kończąc na metodzie `main()` będącej punktem rozpoczęcia programu.

Stos wywołań z rysunku 2.3 wskazuje, iż niewyłapany wyjątek `NullPointerException` wystąpił w trakcie wykonywania klasy `ExceptionDemo`. Sam wyjątek wystąpił w metodzie `doSomethingElse()` (wiersz 18. klasy `ExceptionDemo`). Metoda ta została wywołana przez metodę `doSomething()` (wiersz 13.). Metoda `doSomething()` została wywołana przez metodę `main()` (wiersz 7.).

Rysunek 2.3. *Przykład stosu wywołań dla niewyłapanego wyjątku*

W kilku kolejnych punktach przedstawimy, w jaki sposób uniknąć domyślnego zachowania w trakcie pisania kodu metody, która może zgłosić wyjątek. Omówimy również sposoby takiego postępowania.

Sekwencja try-catch-finally

Podstawowym sposobem unikania niewyłapanych wyjątków jest napisanie sekwencji `try-catch`, która „wyłapuje" wyjątek i „obsługuje go" zamiast korzystać z domyślnego zachowania maszyny wirtualnej.

```
try {
  // Polecenie wykonujące operację na pliku.
}
```

```
catch (IOException ex) {
  ex.printStackTrace();  // Wyświetlenie stosu wywołań.
  System.exit(1);        // Zakończenie programu ze wskazaniem błędu.
}
```

Jeśli wszystkie polecenia z bloku `try` zostaną wykonane bez żadnych błędów, blok `catch` zostanie pominięty. Jeżeli wystąpi wyjątek `IOException`, nastąpi wyjście z bloku `try` i wykonanie bloku `catch`. Przykładowy blok `catch` wyświetla na konsoli (standardowy strumień błędów `System.err`) sekwencję wywołań metod, która doprowadziła do błędu (najpierw pojawia się najnowsze wywołanie metody, a następnie coraz to starsze), i kończy działanie programu.

Obsługa wyjątków w celu usunięcia błędu

Poza samym zgłoszeniem błędu wyjątki dają również możliwość usunięcia błędu. Typowym źródłem błędów są dane wpisywane przez użytkownika.

Wyobraźmy sobie, iż metoda `JOptionPane.showInputDialog()` wyświetla okno dialogowe, które daje użytkownikowi możliwość wpisania tekstu. Po wpisaniu tekstu i naciśnięciu klawisza *Enter* metoda zwraca ciąg znaków zawierający wpisaną treść. Jeżeli oczekuje się wartości całkowitoliczbowej, trzeba skonwertować ciąg znaków na liczbę. Za tego rodzaju konwersję odpowiada metoda `Integer.parseInt()`, która może zgłosić wyjątek `NumberFormatException`.

Przykład 2.5

Metoda `readInt()` (listing 2.1) zwraca wartość całkowitą, która została wpisana w oknie dialogowym przez użytkownika programu. Parametrem metody jest tekst, który ma się pojawić w oknie dialogowym jako zapytanie.

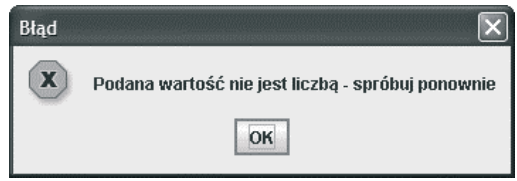
Listing 2.1. *Metoda readInt (część klasy MyInput.java) z jednym parametrem*

```
/** Metoda zwracającą liczbę całkowitą podaną przez użytkownika.
  @param prompt Tekst zapytania.
  @return Odczytana wartość jako typ int.
 */
public static int readInt(String prompt) {
  while (true) { // Wykonywanie pętli aż do momentu wpisania poprawnej wartości.
    try {
      String numStr = JOptionPane.showInputDialog(prompt);
      return Integer.parseInt(numStr);
    }
    catch (NumberFormatException ex) {
      JOptionPane.showMessageDialog(
          null,
          "Podana wartość nie jest liczbą - spróbuj ponownie",
          "Błąd", JOptionPane.ERROR_MESSAGE);
    }
  }
}
```

Warunek pętli `while (true)` zapewnia, iż sekwencja `try-catch` będzie wykonywana tak długo, aż użytkownik wpisze poprawą wartość. Polecenia:

```
String numStr = JOptionPane.showInputDialog(prompt);
return Integer.parseInt(numStr);
```

wyświetlają okno dialogowe i zwracają wartość całkowitą, jeśli `numStr` zawiera jedynie cyfry. W przeciwnym razie zostaje zgłoszony wyjątek `NumberFormatException`, który obsługuje klauzula `catch`. Blok `catch` wyświetla okno komunikatu o błędzie, wywołując metodę `JOptionPane.showMessageDialog()`. Ostatni argument wywołania metody, `JOptionPane.ERROR_MESSAGE`, powoduje wyświetlenie w oknie ikony znaku stop (patrz rysunek 2.4). Po zamknięciu okna użytkownik ma kolejną szansę na wpisanie wartości całkowitej.



Rysunek 2.4. *Błąd wywołany przez wpisanie tekstu, który nie jest wartością całkowitą*

Blok try

Składnia bloku `try` jest następująca.

```
try {
  // Kod, który może zgłosić wyjątek.
}
```

Klauzula i blok catch

Wyjątki są wyłapywane przez tak zwane klauzule `catch`. Klauzula taka przypomina metodę. Jej składnia jest następująca:

```
catch (KlasaWyjatku argumentWyjatku) {
  // Kod obsługujący wyjątek.
}
```

Kod wewnątrz nawiasów klamrowych nosi nazwę **bloku** `catch`. Klauzula `catch` (lub kilka klauzul) musi znajdować się za blokiem `try`. Kilka klauzul `catch` stosuje się wtedy, gdy chce się obsługiwać wiele klas wyjątków. Wtedy jedna klasa wyjątku odpowiada jednej klauzuli `catch`.

Wyjątek dopasowuje się do klauzuli `catch`, jeśli jego typ jest taki sam jak argument klauzuli lub stanowi podklasę tego argumentu. Jeżeli w bloku `try` wystąpi wyjątek, po kolei sprawdzane są przypisane do niego klauzule `catch`, by znaleźć dopasowanie. Gdy dopasowanie zostanie znalezione, zostaje wykonany odpowiedni blok `catch`. W przeciwnym razie wyjątek jest propagowany w górę łańcucha wywołań metod, by sprawdzić, czy któraś z nich nie występuje w bloku `try` z odpowiednią klauzulą `catch`. Jeśli znajdzie się taka klauzula, zostanie wykonany jej blok. Gdy żadna klauzula nie dopasowała się do wyjątku, taki wyjątek uważa się za niewyłapany.

Przykład 2.6

Ciało przedstawionej poniżej metody `readIntTwo()` zawiera jedynie instrukcje z bloku `try` metody `readInt()` (listing 2.1).

```
public static int readIntTwo(String prompt) {
  String numStr = JOptionPane.showInputDialog(prompt);
  return Integer.parseInt(numStr);
}
```

W tym przypadku, jeśli wyjątek `NumberFormatException` zostanie zgłoszony przez metodę `parseInt()`, metoda `readIntTwo()` nie będzie potrafiła go obsłużyć (brak klauzuli `catch`). Spowoduje to propagację wyjątku do metody, która wywołała metodę `readIntTwo()`, by tam poszukać odpowiedniej klauzuli `catch`. Jeśli metoda `readIntTwo()` została wywołana do odczytania wartości do zmiennej `age` w przedstawiony poniżej sposób:

```
try {
  // Wpisz wiek.
  age = readIntTwo("Wpisz swój wiek");
} catch (Exception ex) {
  System.err.println("W trakcie wykonywania metody readIntTwo() wystąpił błąd");
  age = DEFAULT_AGE;
}
```

klauzula `catch` obsłuży wyjątek `NumberFormatException` (ponieważ wyjątek ten jest podklasą klasy `Exception`) i przypisze zmiennej `age` wartość `DEFAULT_AGE`. Dodatkowo w oknie konsoli wyświetli stosowny komunikat. Program będzie wykonywał się dalej od następnego polecenia po końcu sekwencji `try-catch`.

Warto traktować klauzulę `catch` jak metodę, a blok `catch` jak jej treść. Termin **klauzula** `catch` dotyczy zarówno nagłówka, jak i treści, natomiast termin **blok** `catch` dotyczy tylko treści. Rozróżnienie to nie ma dużego znaczenia, dlatego w wielu innych książkach i publikacjach stosuje się je zamiennie.

Przykład 2.7

Wyjątek `EOFException` jest podklasą wyjątku `IOException`. Klauzule `catch` z poniższego przykładu muszą występować w dokładnie takiej kolejności, aby nie został zgłoszony błąd składniowy `catch is unreachable` (klauzula `catch` nigdy nie zostanie wywołana). Pierwszy blok `catch` obsługuje sytuację, w której zostały odczytane i przetworzone wszystkie dane (nie jest to sytuacja błędna). Blok `catch` informuje o tym fakcie użytkownika i kończy działanie programu. Druga klauzula `catch` obsługuje wszystkie pozostałe wyjątki wejścia-wyjścia (wskazujące sytuacje błędne) — wywołuje metodę `printStackTrace()`, by wyświetlić stos wywołań podobny do tego z rysunku 2.3, i dodatkowo kończy program, wskazując sytuację nietypową (`System.exit(1)`).

(*Uwaga:* Metoda `printStackTrace()` jest zdefiniowana w klasie `Throwable`, więc znajduje się również we wszystkich klasach po niej dziedziczących.)

```
catch(EOFException ex) {
  System.out.print("Osiągnięto koniec pliku ");
  System.out.println("- zakończono przetwarzanie");
  Syste.exit(0);
}
catch (IOException ex) {
  System.err.println("Błąd wejścia-wyjścia:");
  ex.printStackTrace();
  System.exit(1);
}
```

PUŁAPKA

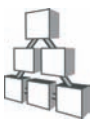
Nieosiągalny blok catch

Warto pamiętać o tym, iż do obsługi wyjątku wykorzystywany jest blok `catch` z pierwszej pasującej klauzuli `catch`. Wszystkie pozostałe bloki `catch` są pomijane. Jeśli wyjątek klauzuli `catch` jest podklasą typu wyjątku z wcześniejszej klauzuli `catch`, blok `catch` tej drugiej klauzuli nigdy nie zostanie wykonany, więc kompilator Javy wyświetla błąd składniowy `catch is unreachable`. Aby poprawić ten błąd, należy na początku umieścić klauzule `catch` z podklasami, a dopiero później klauzulę `catch` związaną z ogólnym wyjątkiem.

Blok finally

W momencie zgłoszenia wyjątku aktualny ciąg instrukcji programu zostaje zawieszony, a wykonywanie programu przenosi się do odpowiedniego bloku `catch`. Program nie może już w żaden sposób wrócić do bloku `try`. Zamiast tego przetwarzanie jest kontynuowane od pierwszej instrukcji po klauzulach `catch`, dla których został zgłoszony wyjątek.

Istnieją sytuacje, w których kontynuowanie programu po wyjątku może doprowadzić do problemów. Jeśli na przykład pewne obliczenia musiały zostać przeprowadzone przed powrotem z metody, obliczenia te trzeba by zduplikować zarówno w bloku `try`, jak i wszystkich blokach `catch`. Aby uniknąć tej niepotrzebnej duplikacji (która może doprowadzić do błędów), stosuje się blok `finally`. Kod tego bloku zostaje wykonany albo po zakończeniu bloku `try`, albo po zakończeniu odpowiedniego bloku `catch` (jeśli był wykonywany). Blok `finally` jest opcjonalny. Jego przykład znajduje się w podsumowaniu składni.



SKŁADNIA Sekwencja try-catch-finally

POSTAĆ:

```
try {
  instrukcje, które mogą spowodować zgłoszenie wyjątku
}
catch (KlasaWyjatku1 argumentWyjatku1) {
  polecenia obsługujące KlasaWyjatku1
```

```
}
catch (KlasaWyjatku2 argumentWyjatku2) {
  polecenia obsługujące KlasaWyjatku2
}
catch (KlasaWyjatkun argumentWyjatkun) {
  polecenia obsługujące KlasaWyjatkun
}
finally {
  polecenia wykonywane po zakończeniu bloku try lub jednego z bloków catch
}
```

PRZYKŁAD:

```
try {
  String sizeStr = JOptionPane.showInputDialog("Wpisz rozmiar");
  size = Integer.parseInt(sizeStr);
}
catch (NumberFormatException ex) {
  size = DEFAULT_SIZE; // Zastosowanie wartości domyślnej, jeśli błędny format liczby
całkowitej.
}
finally {
  if (size > MAX_CAPACITY)
    size = MAX_CAPACITY;
}
```

ZNACZENIE:

Instrukcje z bloku `try` zostaną wykonane w całości, chyba że wcześniej zostanie zgłoszony wyjątek. Jeśli istnieje klauzula `catch` obsługująca tego rodzaju wyjątek, zostanie wykonany jej blok. Po zakończeniu wykonywania bloku `try` lub bloku `catch` program wykona blok `finally`.

Jeśli nie istnieje klauzula `catch` przystosowana do obsługi wyjątku zgłoszonego w bloku `try`, zostaje wykonany blok `finally`, a następnie wyjątek jest przekazywany wyżej w łańcuchu wywołań, aż zostanie obsłużony przez jedną z metod wywołujących. Jeśli wyjątku nie obsłuży żadna klauzula `catch`, zostanie on przetworzony przez maszynę wirtualną Javy jako wyjątek niewyłapany.

Poinformowanie o błędzie i zakończenie programu

Istnieje wiele sytuacji, w których wyjątek zostaje zgłoszony, ale nie istnieje prosty sposób poradzenia sobie z jego obsługą i kontynuowania pracy programu. Na przykład odczyt danych z pliku lub konsoli może zaowocować zgłoszeniem wyjątku `IOException`. W takiej sytuacji klauzula `catch` powinna wyświetlić stos wywołań metod i zakończyć program. Oto kod wykonujący to zadanie:

```
catch (IOException ex) {
  ex.printStackTrace();
  System.exit(1);
}
```

Wywołanie metody `System.exit(1)` powoduje zakończenie wykonywania programu ze wskazaniem sytuacji błędnej.

PUŁAPKA Ignorowanie wyjątków

Wyjątki zostały zaprojektowane po to, by programista dowiadywał się o powstałych błędach i miał możliwość zareagowania na nie. Niektórzy programiści nie lubią korzystać z tej możliwości, więc piszą poniższy kod:

```
catch (Exception e) {}
```

Choć taka klauzula jest poprawna składniowo i eliminuje wiele nieprzyjemnych komunikatów o błędach, w większości przypadków jest bardzo złym rozwiązaniem. Program kontynuuje wykonywanie po sekwencji `try-catch` bez żadnej informacji o tym, iż wystąpił problem. Polecenie, które zgłosiło wyjątek, z pewnością nie wykonało się poprawnie. Instrukcje znajdujące się po nim nie wykonały się w ogóle. Program będzie zawierał ukryte defekty, z których użytkownicy nie będą zadowoleni. W pewnych sytuacjach może mieć to nieprzyjemne konsekwencje.

STYL PROGRAMOWANIA

Korzystanie z wyjątków w celu ułatwienia analizy kodu

W językach programowania, które nie zawierają wyjątków, programista musi mieszać kod odpowiadający za sprawdzanie sytuacji nietypowych (które na ogół zdarzają się niezmiernie rzadko) z kodem wykonywanym regularnie. Na ogół prowadzi to do powstania mało czytelnego kodu:

```
krok A
if (krok A udany) {
  krok B
  if (Krok B udany) {
    krok C
  } else {
    zgłoś błąd w kroku B
    wyczyść po kroku A
  }
} else {
  zgłoś błąd w kroku A
}
```

Po zastosowaniu wyjątków powyższy kod jest znacznie bardziej czytelny:

```
try {
  krok A
  krok B
  krok C
} catch (wyjątek wskazujący niepowodzenie w kroku B) {
  zgłoś błąd w kroku B
  wyczyść po kroku A
} catch (wyjątek wskazujący niepowodzenie w kroku A) {
  zgłoś błąd w kroku A
}
```

ĆWICZENIA DO PODROZDZIAŁU 2.3

SPRAWDŹ SIĘ

1. Załóżmy, iż metoda `main()` wywołuje metodę `first()` w 10. wierszu klasy `MyApp`, metoda `first()` wywołuje metodę `second()` w 10. wierszu klasy `Other`, a metoda `second()` wywołuje metodę `parseInt()` w wierszu 20. klasy `Other`. Wywołania te doprowadziły do zgłoszenia wyjątku `NumberFormatException` w wierszu 430. klasy `Integer`. Przedstaw stos wywołań.
2. Załóżmy, iż mamy klauzule `catch` dla wyjątków `Exception`, `NumberFormatException` i `RuntimeException` po bloku `try`. Pokaż poprawną sekwencję tych klauzul `catch`.

PROGRAMOWANIE

1. Dla poniższego bloku `try`:

```
try {
  numStr = in.readLine();
  num = Integer.parseInt(numStr);
  average = total / num;
}
```

napisz sekwencję `try-catch-finally` z klauzulami `catch` dla wyjątków `ArithmeticException`, `NumberFormatException` i `IOException`. Dla wyjątku `ArithmeticException` należy ustawić zmienną `average` na 0, wyświetlić stosowy komunikat błędu i stos wywołań, a następnie zakończyć program. Po zakończeniu bloku `try` lub bloku `catch` dla `ArithmeticException` należy wyświetlić tekst „That's all folks”, korzystając z bloku `finally`.

2.4. Zgłaszanie wyjątków

Poprzedni podrozdział opisuje wyłapywanie i obsługę wyjątków za pomocą sekwencji `try-catch`. Jako alternatywę dla wyłapania wyjątku w metodzie niższego poziomu można umożliwić jej wyłapanie i obsługę w metodzie wyższego poziomu. Zadanie to można wykonać na dwa sposoby:

1. Deklaruje się, iż metoda niższego poziomu może zgłosić wyjątek weryfikowany, dodając klauzulę `throws` w nagłówku metody.
2. Samemu zgłasza się wyjątek w metodzie niższego poziomu, używając instrukcji `throw`, gdy tylko zostanie wykryta nieprawidłowa sytuacja.

Klauzula throws

Kolejny przykład ilustruje użycie klauzuli `throws` do zadeklarowania, iż metoda może zgłosić pewien rodzaj wyjątku weryfikowanego. Jest to przydatne rozwiązanie, jeśli metoda wyższego poziomu zawiera klauzulę `catch` dla tego rodzaju wyjątku. Jeśli nie

zastosuje się klauzuli throws, trzeba zduplikować blok catch w metodzie niższego poziomu, by uniknąć podczas kompilacji błędu unreported exception (nie zgłoszony wyjątek).

Przykład 2.8

Metoda readData() odczytuje dwa ciągi znaków z obiektu BufferedReader o nazwie console powiązanego z System.in (wejście konsoli systemowej) i zapisuje je w polach danych firstName i lastName. Każde wywołanie metody readLine() może spowodować zgłoszenie wyjątku weryfikowanego IOException, więc metoda readData() nie skompiluje się bez zastosowania klauzuli throws. Jeśli się ją pominie, kompilator zgłosi błąd składniowy: unreported exception: java.io.IOException; must be caught or declared to be thrown (nie zgłoszony wyjątek: java.io.IOException; należy go wyłapać lub zadeklarować jego zgłaszanie).

```
public void readData() throws IOException {
  BufferedReader console = new BufferedReader(
    new InputStreamReader(System.in));
  System.out.print("Wpisz imię: ");
  firstName = console.readLine();
  System.out.print("Wpisz nazwisko: ");
  lastName = console.readLine();
}
```

Jeśli metoda readData() jest wywoływana przez metodę setNewPerson(), metoda ta musi zawierać blok catch, który obsługuje wyjątek IOException.

```
public void setNewPerson() {
  try {
    readData();
    // Przetwarzanie odczytanych danych.
    . . .
  }
  catch (IOException iOEx) {
    System.err.println(" Wywołanie readLine() z metody readData() zwróciło błąd");
    iOEx.printStackTrace();
    System.exit(1);
  }
}
```

Jeśli metoda może zwrócić więcej niż jeden typ wyjątku, należy podać wszystkie typy po klauzuli throws, oddzielając je przecinkami. Gdy pominie się choć jeden typ wyjątku weryfikowanego, kompilator zgłosi błąd. Kompilator sprawdza również, czy wszystkie typy wymienione po klauzuli throws są klasami wyjątków.

STYL PROGRAMOWANIA

Wykorzystanie znacznika @throws z Javadoc do oznaczania wyjątków nieweryfikowanych

Wymienienie wyjątków nieweryfikowanych w klauzuli throws jest poprawne składniowo, ale nie uważa się tego za dobre rozwiązanie. Zamiast tego warto wyjątki nieweryfikowane wymienić w dokumentacji metody, stosując znacznik @throws. W ten sposób informuje się innych programistów, iż taki wyjątek może zostać zgłoszony.

Instrukcja throw

Instrukcję `throw` stosuje się na ogół w metodach niższego poziomu, by wskazać zaistnienie błędu. Jej wykonanie powoduje natychmiastowe zaprzestanie wykonywania metody niskiego poziomu i rozpoczęcie przez maszynę wirtualną poszukiwań procedury obsługi wyjątku w opisany wcześniej sposób. Takie rozwiązanie stosuje się przede wszystkim wtedy, jeśli wyjątek jest nieweryfikowany i istnieje duże prawdopodobieństwo wyłapania wyjątku w metodzie wyższego poziomu. Jeśli zgłaszany wyjątek jest typu weryfikowanego, musi zostać zadeklarowany w klauzuli `throws` zgłaszającej go metody.

Przykład 2.9

Metoda `addOrChangeEntry()` interfejsu `PhoneDirectory` przyjmuje dwa parametry typu `String`: `name` i `number`. Parametr `number` powinien reprezentować poprawny numer telefonu. Chcemy go sprawdzić, by się przekonać, czy rzeczywiście został wpisany we właściwym formacie. Jeśli założymy, iż istnieje metoda `isPhoneNumberFormat()`, która weryfikuje poprawność numeru telefonu, możemy napisać metodę `addOrChangeEntry()` w sposób następujący:

```
public String addOrChangeEntry(String name, String number) {
  if (!isPhoneNumberFormat(number)) {
    throw new IllegalArgumentException
      ("Błędny numer telefonu: " + number);
  }
  // Dodanie lub zmiana numeru telefonu.
  . . .
}
```

Instrukcja `throw` tworzy i zgłasza wyjątek `IllegalArgumentException`, który może zostać obsłużony przez metody wyższego poziomu lub maszynę wirtualną, jeśli nie zostanie wyłapany. Argument konstruktora (`"Błędny numer telefonu: " + number`) dla nowego wyjątku to komunikat opisujący powód powstania błędu.

Jeśli wywołamy tę metodę, używając poniższej sekwencji `try-catch`:

```
try {
  addOrChangeEntry(myName, myNumber);
} catch (IllegalArgumentException ex) {
  System.err.println(ex.getMessage());
}
```

gdy zmienna `myNumber` zawiera tekst `"1xx1"` (niepoprawny numer telefonu), na wyjściu konsoli pojawi się następujący tekst:

```
Błędny numer telefonu: 1xx1
```

SKŁADNIA Instrukcja throw

POSTAĆ:

```
throw new KlasaWyjatku();
throw new KlasaWyjatku(szczegolowyKomunikat);
```

PRZYKŁAD:

```
throw new FileNotFoundException("Nie odnaleziono pliku " + fileSource);
```

ZNACZENIE:

Zostaje utworzony i zgłoszony nowy wyjątek typu *KlasaWyjatku*. Opcjonalny parametr tekstowy *szczegolowyKomunikat* służy do określenia komunikatu błędu powiązanego z wyjątkiem. Jeśli metoda wyższego poziomu wyłapująca wyjątek zawiera następująca klauzulę `catch`:

```
} catch (KlasaWyjatku ex) {
  System.err.println(ex.getMessage());
  System.exit(1);
}
```

komunikat *szczegolowyKomunikat* zostanie wyświetlony na standardowym wyjściu błędów przed zakończeniem programu.

Przykład 2.10

Listing 2.2 przedstawia drugą metodę `readInt()`, która przyjmuje trzy parametry. Podobnie jak w metodzie `readInt()` z przykładu 2.5 pierwszym parametrem jest tekst zapytania. Drugi i trzeci parametr określają zakres dopuszczalnych wartości całkowitych. Metoda zwraca pierwszą wartość całkowitą wpisaną przez użytkownika, która mieści się w podanym zakresie.

Listing 2.2. *Metoda readInt() z pliku MyInput.java z trzema parametrami*

```
/** Metoda zwraca wartość całkowitą w podanym zakresie.
  pocz: minN <= maxN.
  @param prompt Wyświetlany komunikat.
  @param minN Początek zakresu.
  @param maxN Koniec zakresu.
  @throws IllegalArgumentException
  @return Pierwsza liczba, która znajdzie się w zakresie.
 */
public static int readInt(String prompt, int minN, int maxN) {
  if (minN > maxN) {
    throw new IllegalArgumentException(
        "W readInt() minN " + minN
        + " nie jest <= maxN " + maxN);
  }
  // Argumenty są poprawne, odczytanie wartości.
  boolean inRange = false; // Załóż brak poprawnej wartości.
  int n = 0;
  while (!inRange) { // Powtarzaj aż do uzyskania poprawnej wartości.
    try {
      String line = JOptionPane.showInputDialog(
          prompt + "\nWpisz liczbę całkowitą między "
          + minN + " i " + maxN);
      n = Integer.parseInt(line);
      inRange = (minN <= n && n <= maxN);
    }
    catch (NumberFormatException ex) {
      JOptionPane.showMessageDialog(
          null,
```

```
            "Podana wartość nie jest liczbą całkowitą - spróbuj ponownie",
            "Błąd", JOptionPane.ERROR_MESSAGE);
      }
    } // Koniec pętli.
    return n; // Wartość n znajduje się w zakresie.
  }
```

Instrukcja `if` sprawdza, czy wartości graniczne nie definiują pustego zakresu (`minN > maxN`). Jeśli tak, poniższa instrukcja:

```
throw new IllegalArgumentException(
  "W readInt() minN " + minN + " nie jest <= maxN " + maxN);
```

zgłasza wyjątek `IllegalArgumentException`, tworząc egzemplarz tej klasy. Komunikat przekazywany do konstruktora określa powód zgłoszenia wyjątku. Komunikat zostanie wyświetlony przez metodę `printStackTrace()` lub zwrócony przez metody `getMessage()` i `toString()`.

Jeśli zakres nie jest pusty, dochodzi do wykonania pętli `while`. Warunek pętli (`!inRange`) jest prawdziwy tak długo, aż użytkownik nie wpisze wartości całkowitej w poprawnym zakresie. Blok `try` wyświetla okno dialogowe z zapytaniem zawierającym informację na temat poprawnego zakresu. Instrukcja:

```
inRange = (minN <= n && n <= maxN);
```

ustawia zmienną `inRange` na wartość `true`, jeśli wartość przypisana do `n` znajduje się w zakresie. Jeśli tak się stanie, dochodzi do zakończenia pętli i zwrócenia wartości `n`. Jeżeli jednak użytkownik nie wpisze wartości liczbowej, blok `catch` wyświetli stosowny komunikat. Jeśli użytkownik nie podał liczby lub liczba ta nie znajduje się w wymaganym zakresie, zmienna `inRange` pozostanie ustawiona na wartości `false`. W ten sposób warunek `!inRange` będzie prawdziwy i nastąpi wykonanie kolejnej iteracji pętli.

STYL PROGRAMOWANIA

Powody zgłaszania wyjątków

Niektórzy zapewne zastanawiają się, jakie mogą być powody celowego zgłaszania wyjątków. Jeśli wyjątek nie zostanie wyłapany przez jedną z metod wyższego rzędu, spowoduje zakończenie działania programu. Warto zauważyć, iż w przykładach z niniejszego podrozdziału kontynuacja pracy programu z pustym zakresem (metoda `readInt()`) lub niepoprawnym numerem telefonu (metoda `addOrChangeEntry()`) nie miałoby wielkiego sensu. Pętla z metody `readInt()` wykonywałaby się w nieskończoność, gdyby zakres był pusty. Ponieważ wartości graniczne zostały przekazane przez metodę wyższego poziomu, pobieranie nowych wartości w `readInt()` nie byłoby wskazane. Stosując wyjątek i wyłapując go w miejscu zdefiniowania wartości granicznych, programista ma możliwość zastosowania innych wartości granicznych w kolejnym wywołaniu metody.

Wyłapywanie wyjątków w programie książki telefonicznej

Analiza przypadku książki telefonicznej z podrozdziału 1.5 ilustruje obsługę wyjątków w kontekście większego programu. Metody `loadData()` i `save()` zawierają sekwencje `try-catch` obsługujące błędy przetwarzania plików. Listing 2.3 przedstawia sekwencję `try-catch` metody `loadData()`, która wczytuje zawartość pliku do tablicy spisu telefonów. Blok `try` zawiera kilka instrukcji mogących zgłosić błąd `IOException`. Poniższa instrukcja:

```
BufferedReader in = new BufferedReader(new FileReader(sourceName));
```

zgłasza wyjątek `FileNotFoundException`, jeśli nie zostanie odnaleziony plik o nazwie przechowywanej w zmiennej `sourceName`. Pierwszy blok `catch` obsługuje ten wyjątek, po prostu nic nie robiąc, ponieważ brak pliku oznacza brak możliwości wczytania danych.

Każde wywołanie metody `readLine()` lub wywołanie metody `close()` może spowodować zgłoszenie wyjątku `IOException` obsługiwanego przez drugi blok `catch`. Ten drugi blok wyświetla stos wywołań metod i kończy wykonywanie programu, wskazując zaistnienie sytuacji nietypowej (`System.exit(1)`).

Warto pamiętać, iż klauzula `catch` dla wyjątku `FileNotFoundException` musi poprzedzać klauzulę dla wyjątku `IOException`, ponieważ pierwszy wyjątek stanowi podklasę drugiego wyjątku. Gdyby klauzule zamienić miejscami, kompilator zgłosiłby błąd składniowy nieosiągalnej klauzuli `catch`.

Listing 2.3. *Metoda loadData() z sekwencją try-catch (plik ArrayBasedPD.java)*

```
/** Metoda odczytuje dane z pliku.
     pocz: Miejsce przechowywania wpisów zostało utworzone i jest puste.
       Jeśli plik istnieje, w kolejnych wierszach zawiera na przemian
       nazwiska i numery telefonów osób.
     końc: Dane z pliku zostały wczytane do tablicy z książką telefoniczną.
     @param sourceName Nazwa pliku danych.
 */
public void loadData(String sourceName) {
  // Zapamiętaj nazwę pliku danych.
  this.sourceName = sourceName;
  try {
    // Utwórz obiekt BufferedReader dla pliku.
    BufferedReader in = new BufferedReader(
        new FileReader(sourceName));
    String name;
    String number;

    // Odczytaj nazwisko i numer telefonu, a następnie dodaj nowy wpis.
    while ( (name = in.readLine()) != null) {
      // Odczytaj nazwisko i numer z kolejnych wierszy.
      if ( (number = in.readLine()) == null) {
        break; // Nie udało się odczytać numeru - koniec pętli.
      }
      // Dodanie wpisu dla pobranego nazwiska i numeru.
      add(name, number);
```

```
      }
      // Zamknięcie pliku.
      in.close();
    }
    catch (FileNotFoundException ex) {
      // Nic nie rób - nie ma danych do załadowania.
      return;
    }
    catch (IOException ex) {
      System.err.println("Załadowanie książki telefonicznej nie powiodło się.");
      ex.printStackTrace();
      System.exit(1);
    }
  }
```

Metoda `processCommands()` z interfejsu użytkownika dla programu książki telefonicznej z podrozdziału 1.7 również zawiera sekwencję `try-catch` (listing 1.7), która obsługuje wyjątek `InputMismatchException` spowodowany wywołaniem metody `Scanner.nextInt()` przedstawionej na listingu 2.4. Innymi możliwymi źródłami wyjątków są metody `doAddChangeEntry()`, `doLookupEntry()` i `doRemoveEntry()`. Zgłoszenie wspomnianego wyjątku przez jedną z nich zostanie wyłapane przez klauzulę `catch` metody `processCommands()`. Ponieważ jednak metody te korzystają z metody `Scanner.nextLine()` zamiast z metody `Scanner.nextInt()`, wystąpienie wyjątku `InputMismatchException` jest bardzo mało prawdopodobne. Warto zauważyć, iż przypadek `default` instrukcji `switch` wyświetla komunikat ostrzeżenia, jeśli podana wartość znajdowała się poza dopuszczalnym zakresem.

Listing 2.4. *Sekwencja try-catch metody processCommands (plik PDConsoleUI.java)*

```
try {
  choice = scIn.nextInt(); // Odczytaj wybór.
  scIn.nextLine(); // Pomiń znak nowego wiersza.
  switch (choice) {
    case 0: doAddChangeEntry(); break;
    case 1: doLookupEntry(); break;
    case 2: doRemoveEntry(); break;
    case 3:
    case 4: doSave(); break;
    default: System.out.println("*** Niepoprawny wybór: "
                        + choice
                        + " - spróbuj ponownie!");
  }
} catch (InputMismatchException ex) {
  System.out.println("*** Niepoprawne dane - "
                     + "wpisz liczbę całkowitą między 0 a "
                     + (commands.length-1));
  scIn.nextLine(); // Pomiń błędne wejście.
  choice = -1;
}
```

STYL PROGRAMOWANIA

Zgłaszanie kontra wyłapywanie wyjątków

Zawsze można uniknąć obsługi wyjątków, deklarując, iż zostaną zgłoszone. Można też je zgłaszać, by to metoda znajdująca się wyżej w hierarchii zapewniła ich obsługę. Ogólnie jednak lepiej obsłużyć wyjątek na miejscu, zamiast przekazywać go gdzieś wyżej. Daje to możliwość obejścia błędu i kontynuowania działania aktualnej metody. Takie podejście zostało zastosowane w dla wyjątku `NumberFormatException` w obu metodach `readInt()` (patrz listingi 2.1 i 2.2). Jeśli błędu nie można naprawić i istnieje szansa, iż podobny błąd może wystąpić w innym miejscu w łańcuchu wywołań, warto przekazać wyjątek wyżej, zamiast duplikować kod obsługi wyjątków w kilku metodach. Zalecamy stosowanie się do następujących wskazówek:

- Jeśli wyjątek można naprawić w aktualnej metodzie, należy obsłużyć go w tej metodzie.
- Jeśli wyjątek weryfikowany najprawdopodobniej będzie wyłapywany w metodzie wyższego poziomu, należy w deklaracji metody użyć klauzuli `throws` i znacznika `@throws` w dokumentacji metody.

Jeśli wyjątek nieweryfikowany będzie wyłapywany w metodzie wyższego poziomu, należy użyć znacznika `@throws` w dokumentacji metody, aby wskazać innym programistom taką ewentualność. Dla wyjątków nieweryfikowanych nie ma potrzeby stosowania klauzuli `throws`.

ĆWICZENIA DO PODROZDZIAŁU 2.4

SPRAWDŹ SIĘ

1. Wyjaśnij różnicę między klauzulą `throws` a instrukcją `throw`.
2. Kiedy lepiej zadeklarować wyjątek niż obsługiwać go w metodzie?
3. Kiedy lepiej zgłaszać wyjątek niż obsługiwać go w metodzie?
4. Jakiego rodzaju wyjątki należy umieszczać w klauzuli `throws`?
5. Dla następujących sytuacji wskaż, czy lepiej wyłapać wyjątek, czy zadeklarować wyjątek lub zgłosić wyjątek w metodzie niższego poziomu. Odpowiedź uzasadnij. Napisz kod, który musi pojawić się w metodzie, by wykonać to zadanie.

   a) Metoda niższego poziomu zawiera wywołanie metody `readLine()`. Wywołująca tę metodę metoda wyższego poziomu zawiera klauzulę `catch` dla wyjątku `IOException`.

   b) Metoda zawiera wywołanie metody `readLine()`, która umożliwia wpisanie wartości przekazywanej jako argument do metody niższego poziomu. Argumentem metody niższego poziomu musi być wartość dodatnia.

   c) Metoda niższego poziomu zawiera wywołanie metody `readLine()`. Wywołująca tę metodę metoda wyższego poziomu nie zawiera klauzuli `catch` dla wyjątku `IOException`.

d) Metoda niższego poziomu odczytuje ciąg znaków i zamienia go na typ `int`. Metoda wyższego poziomu zawiera klauzulę `catch` dla wyjątku `NumberFormatException`.

e) Metoda niskiego poziomu wykrywa błąd, którego nie da się naprawić (jest to wyjątek nieweryfikowany).

PROGRAMOWANIE

1. Istnieje następująca instrukcja `throw`:

```
throw new FileNotFoundException("Nie odnaleziono pliku " + fileSource);
```

Napisz klauzulę `catch` z metody wyższego poziomu, która obsługuje ten wyjątek.

2. Poniżej znajduje się zmodyfikowana wersja metody `setElementOfX()` z przykładu 2.2. Sprawdza ona, czy parametry `index` i `val` znajdują się we właściwych granicach przez dostępem do tablicy `x`. Zmodyfikuj kod w taki sposób, by zgłaszał wyjątek, jeśli `val` znajduje się poza zakresem (maszyna wirtualna zgłosi wyjątek, jeśli `index` znajduje się poza zakresem, ale szczegółowy komunikat będzie zawierał tylko wartość `index`). Przekaż stosowny komunikat do nowego obiektu wyjątku. Zmodyfikowana metoda powinna być typu `void`, ponieważ nie istnieje już powód, dla którego trzeba by zwracać zmienną logiczną wskazującą sytuację nietypową. Przedstaw blok `catch` dla metody wyższego poziomu obsługujący wykonany wyjątek.

```
public boolean setElementOfX(int index, int val) {
  if (index >= 0 && index < x.length
      && val >= MIN_VAL && val <= MAX_VAL) {
    x[index] = val;
    return true;
  } else {
    return false;
  }
}
```

2.5. Testowanie programów

Po usunięciu wszystkich błędów składniowych i błędów czasu wykonania program będzie się wykonywał bez przeszkód. Nie oznacza to jednak, iż program nie zawiera błędów logicznych. Ponieważ błędy logiczne bardzo rzadko doprowadzają do wyświetlenia komunikatu o błędzie, często pozostają niezauważone.

Błędy logiczne trudno wykryć i odizolować. Jeśli błąd logiczny występuje we fragmencie programu, który wykonuje się zawsze, każde uruchomienie programu będzie związane z wygenerowaniem nieprawidłowych wyników. Choć nie brzmi to najlepiej, w rzeczywistości jest dobrą sytuacją, ponieważ im błąd częściej występuje, tym łatwiej jest go wykryć. Gdy wyliczana wartość za każdym razem jest błędna, znalezienie błędu logicznego nie powinno zająć zbyt wiele czasu. Jeśli jednak wartość ta stanowi

część większych obliczeń i nie jest wyświetlana, bardzo trudno będzie wyśledzić błąd i znaleźć powodujący go fragment kodu.

Najgorzej, gdy błąd logiczny występuje w stosunkowo mało przejrzystym kodzie, który na dodatek jest wykonywany tylko od czasu do czasu. Jeśli zbiór danych testowych nie przećwiczy tej sekcji kodu, błąd nie ujawni się w trakcie typowych testów programu. Oznacza to, że produkt programistyczny zostanie dostarczony użytkownikom z ukrytymi defektami. Wtedy naprawdę trudno odnaleźć przyczynę problemu i ją usunąć.

Strukturalne ścieżki przejścia

Większość błędów logicznych pojawia się w fazie projektowania i jest wynikiem niepoprawnego algorytmu. Niektóre błędy logiczne są wynikiem błędów typograficznych wprowadzonych na etapie pisania kodu, które nie doprowadziły do błędów składniowych ani wykonania. Jedną z form testów, która nie wymaga uruchamiania programu, jest bardzo uważne sprawdzenie algorytmu przed jego implementacją oraz późniejsze jego sprawdzenie już po napisaniu kodu. Na ogół zadanie to wykonuje się, symulując działanie każdego kroku algorytmu i porównując je z wynikami teoretycznymi.

Ręczne śledzenie działania algorytmu lub programu często komplikuje fakt, iż projektant na ogół przewiduje, co powinien zrobić dany krok. Ponieważ zna zakładane działanie każdego kroku, wymaga znacznie większej dyscypliny, by poprawnie zasymulować wszystkie kroki. Z tego powodu programiści powinny pracować w grupach. Projektant wyjaśnia działanie algorytmu innym członkom zespołu i symuluje jego działanie, gdy inni programiści przyglądają się jego pracy (jest to tak zwana **strukturalna ścieżka przejścia**). Doświadczenia zdobyte przy wielu rzeczywistych projektach wykazują, iż strukturalne ścieżki przejścia pozwalają wykryć wiele defektów już na wczesnym etapie prac.

Poziomy i rodzaje testów

Testowanie to proces sprawdzania programu (lub jego fragmentu) w pewnych kontrolowanych warunkach i kontrolowanie, czy wyniki są zgodne z oczekiwanymi. Celem testów jest wykrycie defektów po poprawieniu wszystkich błędów składniowych (program kompiluje się bez przeszkód). Im dokładniejsze będą testy, tym większe prawdopodobieństwo, iż zostaną wykryte defekty. Niemniej nawet nieskończona ilość testów nie może zagwarantować, iż pozbędziemy się wszystkich błędów w bardziej złożonych programach. Niektóre testy wymagają, by sprawdzić wszystkie możliwe dane wejściowe i stany każdej metody, co na ogół jest zadaniem bardzo trudnym lub wręcz niemożliwym. Z tego powodu wiele programów komercyjnych po swojej premierze musi być łatanych za pomocą odpowiednich poprawek. Wersja *n* zazwyczaj poprawia błędy zauważone w wersji *n*–1.

Testy wykonuje się przede wszystkim na następujących poziomach:

- **Testy jednostkowe** dotyczą sprawdzania możliwe najmniejszej logicznie części programu. W programowaniu obiektowym oznacza to na ogół testowanie klasy lub metody. Złożoność metody określa, czy należy testować ją osobno czy raczej jako część klasy.

- **Testy integracyjne** dotyczą sprawdzania interakcji między jednostkami. Jeśli jednostką jest metoda, testy integracyjne mają związek ze współdziałaniem metod klasy. Na ogół jednak testy tego rodzaju uwzględniają interakcje między wieloma klasami.
- **Testy systemowe** to testy całego programu przeprowadzane w kontekście, w jakim będzie stosowany. Program stanowi zazwyczaj część zbioru innych programów i sprzętu nazywaną systemem. Czasem program działa poprawnie tylko do momentu uruchomienia lub zainstalowania w systemie innego programu. Należy wykryć takie sytuacje.
- **Testy akceptacyjne** to testy systemowe zaprojektowane do wykazania, iż program spełnia wymagania funkcjonalne. Często oznacza to użycie systemu w rzeczywistym środowisku lub najbliżej niego.

Istnieją dwa typy testów:

- **Testowanie jako czarna skrzynka** polega na sprawdzaniu elementu (metody, klasy, programu) na podstawie jego interfejsu i wymagań funkcjonalnych. Często test ten nosi również nazwę **testu funkcjonalnego**. W trakcie testowania metody stosuje się różne parametry wejściowe w dopuszczalnych zakresach, a wyniki działania porównuje się z wynikami teoretycznymi. Dodatkowo sprawdza się, czy metoda poprawnie reaguje na wartości spoza zakresu (na przykład zgłasza wyjątek lub stosuje wartość domyślną). W przypadku metod elementami wejściowymi mogą być nie tylko jej parametry, ale również wartości pól danych i zmienne globalne, z których korzystają metody.
- **Testowanie jako szklana skrzynka** polega na sprawdzaniu elementu (metody, klasy, programu) na podstawie znajomości jego wewnętrznej struktury. Często test ten nosi również nazwę **testu pokrycia** lub **testu otwartej skrzynki**. Celem jest przećwiczenie jak największej liczby ścieżek przejścia przez element. Istnieją różne stopnie pokrycia. Najprostsze to **pokrycie instrukcji**, które zapewnia wykonanie każdej instrukcji przynajmniej raz. **Pokrycie gałęzi** zapewnia, iż każdy wybór w rozgałęzieniach (instrukcjach `if`, instrukcjach `switch` i pętlach) zostanie wykonany. Na przykład instrukcja `if` jest testowana dla wartości, które spowodują uzyskanie w warunku wartości `true` i `false`. Dla kilku nie zagnieżdżonych instrukcji `if` wystarczy wykonać dwa przypadki testowe: jeden powodujący uzyskanie w warunkach wartości `true` i jeden powodujący uzyskanie wartości `false`. **Pokrycie ścieżek** sprawdza wszystkie możliwe ścieżki przejścia przez metodę. Jeśli istnieje n instrukcji `if`, pokrycie ścieżek będzie wymagało wykonana 2^n testów, jeśli instrukcje `if` nie są zagnieżdżone (każdy warunek ma dwie możliwe wartości, co daje łącznie 2^n ścieżek).

Przykład 2.11

Metoda `testMethod()` posiada zagnieżdżoną instrukcję `if` i wyświetla jeden z czterech komunikatów, od „ścieżka 1” do „ścieżka 4”, w zależności od wybranej ścieżki. Przekazane do niej argumenty określają, która ścieżka zostaje wybrana.

```
public void testMethod(char a, char b) {
  if (a < 'M') {
    if (b < 'X') {
      System.out.println("ścieżka 1");
    } else {
      System.out.println("ścieżka 2");
    }
  } else {
    if (b < 'C') {
      System.out.println("ścieżka 3");
    } else {
      System.out.println("ścieżka 4");
    }
  }
}
```

Aby przetestować metodę, trzeba przekazać do niej argumenty o takich wartościach, by zostały wykonane wszystkie ścieżki. Tabela 2.4 przedstawia możliwe wartości i odpowiadające im ścieżki.

Tabela 2.4. *Testowanie wszystkich ścieżek metody testMethod()*

a	b	Komunikat
'A'	'A'	ścieżka 1
'A'	'Z'	ścieżka 2
'Z'	'A'	ścieżka 3
'Z'	'Z'	ścieżka 4

Wybrane wartości testowe dla `a` i `b` w tabeli 2.4 to pierwsza i ostatnia wielkie litery alfabetu. Przy bardziej intensywnych testach należałoby sprawdzić, co się stanie, jeśli przekaże się wartości pomiędzy A i Z. Na przykład, co się stanie, jeśli argument `a` zmieni wartość z L na M? Wybraliśmy je, ponieważ warunek (`a < 'M'`) ma dla nich różne wartości.

Co się stanie, jeśli argumenty `a` i `b` nie będą zawierały wielkich liter? Jeśli na przykład `a` i `b` będą zawierały cyfry (przykładowo `'2'`), zostanie wyświetlony komunikat „ścieżka 1”, ponieważ znaki cyfr poprzedzają znaki wielkich liter (patrz tabela A.2 z dodatku A). Jeśli `a` i `b` będą zawierały małe litery, zostanie wyświetlony komunikat „ścieżka 4” (dlaczego?). Jeśli `a` będzie cyfrą, a `b` małą literą, zostanie wyświetlony komunikat „ścieżka 2” (dlaczego?). Jak nietrudno zauważyć, liczba testów wymaganych do sprawdzenia nawet tak prostej metody jak `testMethod()` potrafi być naprawdę pokaźna.

Przygotowanie do testów

Choć testy przeprowadza się na ogół po napisaniu kodu jednostek, plan testów powinien zostać stworzony na początku fazy projektowania. Elementy planu powinny zawierać między innymi decyzje co do sposobu testowania, momentu rozpoczęcia testów, osoby odpowiedzialnej za testy i rodzaju używanych danych testowych. Jeśli plan testów

zostanie opracowany na etapie projektowania, testy można przeprowadzać równolegle z projektowaniem i pisaniem kodu. Im szybciej zostanie odnaleziony błąd, tym łatwiejsze i tańsze będzie jego poprawienie.

Inną zaletą wczesnego określenia planu testów jest to, że programiści mogą przygotować się do testów w trakcie pisania kodu. Dobry programista praktykuje **programowanie defensywne** i dołącza kod wykrywający nieoczekiwane lub niepoprawne wartości. Jeśli na przykład metoda posiada następujący warunek początkowy:

```
pocz: Wartość n większa od zera.
```

zapewne należałoby dodać następującą instrukcję `if`:

```
if (n <= 0)
  throw new IllegalArgumentException("n <= 0: " + n);
```

na początku metody. W ten sposób metoda zgłosi wyjątek, gdy przekazany do niej argument nie będzie zawierał poprawnej wartości.

Jeśli wartość danych odczytywana z klawiatury powinna zawierać się w zakresie od 0 do 40, programista defensywny użyje trójparametrowej metody `readInt()` (patrz listing 2.2):

```
hours = MyInput.readInt("Wpisz liczbę przepracowanych godzin", 0, 40);
```

Drugi i trzeci argument metody określa dolną i górną granicę dopuszczalnych wartości.

Wskazówki dotyczące testowania programów

Przez większość czasu testuje się programy zawierające zbiory klas, z których każda posiada kilka metod. Oto kilka wskazówek związanych z pisaniem takich metod:

1. Jeśli metoda implementuje interfejs, specyfikacja interfejsu powinna dokumentować parametry wejściowe i oczekiwane wyniki.
2. Dokładnie dokumentuj wszystkie parametry metody i atrybuty klasy, korzystając z komentarzy. Dodatkowo opisz zadania metody, stosując się do konwencji Javadoc opisanej w podrozdziale 1.4.
3. Pozostaw ślad wykonywania, wyświetlając nazwę metody w momencie jej wywołania.
4. Wyświetlaj wartości wszystkich parametrów wejściowych w momencie wejścia do metody. Wyświetl również wartości wszystkich atrybutów klasy, z których korzysta metoda. Sprawdź, czy te wartości mają sens.
5. Wyświetl wynikowe wartości metody przed zakończeniem metody. Dodatkowo wyświetl wartości wszystkich atrybutów klasy, które zostały zmienione w metodzie. Dokonując obliczeń na kartce, sprawdź, czy wartości zostały wyliczone poprawnie.

Plan testowania warto opracowywać w momencie pisania modułu, a nie dopiero po jego ukończeniu. Dołącz polecenia wyjściowe wymagane przez kroki od 2. do 4. do oryginalnego kodu metody. Jeśli jest się usatysfakcjonowanym działaniem metody, można

usunąć polecenia testowe. Jednym ze sposobów wykonania tego zadania jest umieszczenie ich w bloku `if (TESTING)`:

```
if (TESTING) {
  // Kod, który chce się usunąć.
  . . .
}
```

Stałą `TESTING` definiuje się wtedy na początku klasy jako wartość `true`, by włączyć testowanie:

```
private static final boolean TESTING = true;
```

Aby wyłączyć testowanie, ustawia się ją na wartość `false`:

```
private static final boolean TESTING = false;
```

Jeśli okaże się to konieczne, można określić różne wartości dla różnych testów.

Przygotowanie danych testowych

Dane testowe warto określić na etapie analizy i projektowania. Dane powinny być pogrupowane według poziomów testów: jednostkowe, integracyjne i systemowe. W testach czarnej skrzynki koncentrujemy się na związkach między wejściem a wyjściem jednostek. Powinny pojawić się testy sprawdzające wszystkie oczekiwane dane wejściowe, jak i dane nieoczekiwane. Dodatkowo plan testów powinien zawierać opis zachowania jednostki i wyjścia dla każdego zestawu danych wejściowych.

W testach szklanej skrzynki koncentrujemy się na sprawdzeniu wszystkich alternatywnych ścieżek przejścia przez kod. Dane testowe powinny zapewniać sprawdzenie wszystkich warunków instrukcji `if` (w celu uzyskania wartości `true` i `false`). Dla zagnieżdżonych instrukcji `if` należy sprawdzić wszystkie możliwe kombinacje wartości `true` i `false`. Dla instrukcji `switch` powinny zostać sprawdzone wszystkie przypadki oraz niektóre wartości poza tymi przypadkami.

Dla pętli należy sprawdzić, czy wynik będzie poprawny, jeśli nastąpi natychmiastowe opuszczenie pętli (zero iteracji). Dodatkowo należy sprawdzić poprawność wyniku dla jednej iteracji i maksymalnej liczby iteracji. Warto również się upewnić, iż w każdej sytuacji pętla się zakończy.

Testowanie warunków granicznych

Gdy ręcznie śledzi się wykonanie algorytmu lub przeprowadza testy szklanej skrzynki, trzeba sprawdzić wszystkie możliwe przejścia przez algorytm. Należy również sprawdzić przypadki szczególne nazywane **warunkami granicznymi**, aby się upewnić, iż algorytm działa dla tych przypadków równie dobrze jak dla sytuacji bardziej typowych. Jeśli na przykład sprawdza się metodę poszukującą konkretnego elementu w tablicy, kod może zawierać pętlę wyszukiwania podobną do poniższej:

```
for (int i = 0; i < x.length; i++) {
  if (x[i] == target)
    return i;
}
```

Testowanie warunków granicznych dla tej metody oznacza upewnienie się, iż metoda działa dla wszystkich przypadków wymienionych na poniższej liście. Pierwsze cztery przypadki testują warunki graniczne pętli i są wymagane przez testy szklanej skrzynki. Osoba używająca testów czarnej skrzynki również powinna skorzystać z tych przypadków. Kolejny przypadek (cel w środku tablicy) nie jest warunkiem granicznym, ale typową sytuacją, która powinna zostać sprawdzona. Pozostałe przypadki to warunki graniczne tablicy związane z oboma rodzajami testów.

- Poszukiwany element jest pierwszym elementem tablicy (`x[0] == target` równe `true`).
- Poszukiwany element jest ostatnim elementem tablicy (`x[x.length - 1] == target` równe `true`).
- Poszukiwanego elementu nie ma w tablicy (`x[i] == target` zawsze równe `false`).
- Istnieje wiele wystąpień poszukiwanego elementu tablicy (`x[i] == target` równe `true` dla więcej niż jednej wartości `i`).
- Poszukiwany element znajduje się gdzieś w połowie tablicy.
- Tablica ma tylko jeden element.
- Tablica jest pusta.

Aby przeprowadzić testy, można napisać metodę `main()`, która tworzy przeszukiwaną tablicę. Najprościej wykonać taką tablicę, używając listy inicjalizującej. Listing 2.5 przedstawia metodę `main()`, która testuje wszystkie wymienione wcześniej przypadki. W metodzie `search()` przeszukiwana tablica jest pierwszym parametrem natomiast szukany element drugim argumentem.

Listing 2.5. *Plik ArraySearch.java*

```
/** Zawiera metodę statyczną wyszukiwania wartości w tablicy. */
public class ArraySearch {

  /** Przeszukuje tablicę w poszukiwaniu pierwszego wystąpienia wartości.
      @param x Tablica do przeszukania..
      @param target Szukana wartość.
      @return Indeks pierwszego wystąpienia wartości;
           przy jej braku zwrócenie -1.
   */
  public static int search(int[] x, int target) {
    for (int i = 0; i < x.length; i++) {
      if (x[i] == target)
        return i;
    }

    // celu nie znaleziono
    return -1;
```

```
  }

  /** Metoda z testami.
      @param args Argumenty wiersza poleceń (nieużywane).
   */
  public static void main(String[] args) {
    int[] x = {5, 12, 15, 4, 8, 12, 7}; // Tablica do przeszukania.

    // Szukana wartość znajduje się w pierwszym elemencie.
    verify(x, 5, 0);
    // Szukana wartość znajduje się w ostatnim elemencie.
    verify(x, 7, 6);
    // Szukanej wartości nie ma w tablicy.
    verify(x, -5, -1);
    // Test wielu wystąpień wartości w tablicy.
    verify(x, 12, 1);
    // Szukana wartość znajduje się w środku tablicy.
    verify(x, 4, 3);

    // Test dla tablicy jednoelementowej.
    x = new int[1];
    x[0] = 10;
    verify(x, 10, 0);
    verify(x, -10, -1);

    // Test dla tablicy pustej.
    x = new int[0];
    verify(x, 10, -1);
  }

  /** Wywołuje metodę wyszukiwania z podanymi parametrami
      i weryfikuje spodziewane wyniki.
      @param x Tablica do przeszukania.
      @param target Wartość do odnalezienia.
      @param expected Spodziewany wynik.
   */
  private static void verify(int[] x, int target, int expected) {
    int actual = search(x, target);
    System.out.print("search(x, " + target + ") wynosi "
                     + actual + ", oczekiwano " + expected);
    if (actual == expected)
      System.out.println(":  zaliczono");
    else
      System.out.println(":  ****oblano");
  }
}
```

Metoda `verify()` jako argumenty przyjmuje przeszukiwaną tablicę, poszukiwaną wartość oraz oczekiwany wynik. Następnie wywołuje metodę `search()` i wyświetla wynik rzeczywisty oraz oczekiwany.

```
int actual = search(x, target);
System.out.print("search(x, " + target + ") wynosi "
                 + actual + ", oczekiwano " + expected);
```

Dalej wyświetla informację o zaliczeniu lub oblaniu testu w zależności od tego, czy wartość oczekiwana jest równa wartości rzeczywistej.

```
if (actual == expected)
  System.out.println(":  zaliczono");
else
  System.out.println(":  ****oblano");
```

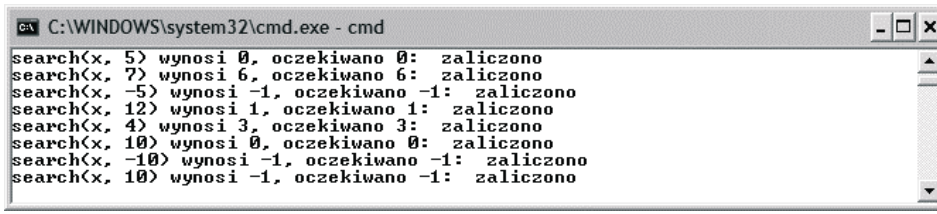
Metoda `verify()` zostaje wywołana dla każdego przypadku testowego. Na przykład pierwsze wywołanie:

```
verify(x, 5, 0);
```

poszukuje w tablicy `x` wartości 5, która znajduje się w `x[0]`. Z tego powodu oczekiwany wynik (trzeci argument) wynosi 0. Powinien zostać wyświetlony następujący wiersz:

```
search (x, 5) wynosi 0, oczekiwano 0: zaliczono
```

Rysunek 2.5 przedstawia wyniki testów. Aby sprawdzić, czy metoda `search()` przeszła test, wystarczy poszukać słowa `zaliczono` na końcu każdego wiersza. Przy większych testach do sprawdzania poprawności używa się osobnego programu.



Rysunek 2.5. *Testy metody search()*

Kto wykonuje testy?

Na ogół testy wykonuje programista, inny członek zespołu programistycznego, który nie pisał sprawdzanego modułu, oraz końcowy użytkownik produktu. Jest niezmiernie ważne, by nie polegać wyłącznie na programistach, którzy pisali testowany moduł, ponieważ programiści często nie zauważają własnych pomyłek. Jeśli nie spostrzegli możliwości powstania błędu w trakcie projektowania, prawdopodobnie nie zauważą go również w trakcie testów. Niektóre firmy zatrudniają nawet specjalne grupy testerów, których jedynym zadaniem jest znajdowanie defektów w kodzie napisanym przez innych programistów.

Uwzględnienie przyszłych użytkowników w testach pozwala odpowiedzieć na pytanie, czy będą oni rozumieli zapytania wyświetlane przez program. Ponieważ użytkownicy na ogół nie znają się na programowaniu, częściej popełniają błędy we wprowadzanych danych niż programiści lub testerzy.

Firmy często posiadają zespoły zapewniania jakości, które sprawdzają, czy proces testowania przebiega poprawnie. Członkowie takich zespołów nie przeprowadzają testów osobiście, ale w niezależny sposób przyglądają się planom testów i ich przeprowadzaniu, a także weryfikują uzyskane wyniki.

Tradycyjnie programista, który napisał jednostkę, wykonuje testy jednostkowe, natomiast osobny zespół testowy zajmuje się testami integracyjnymi i systemowymi. Niemniej coraz więcej organizacji stwierdza, iż wykonywanie testów jednostkowych przez innych programistów jest bardziej efektywne. Istnieje pewna metodologia (nazywana programowaniem ekstremalnym), w której programiści pracują parami; jeden pisze kod, a drugi pisze testy. Tester obserwuje również, jak drugi programista tworzy kod. Co jakiś czas dochodzi między programistami do zamiany ról, aby każdy z nich był odpowiedzialny zarówno za pisanie kodu, jak i testy. Jednostki powinny być niewielkie, by dawały się szybko pisać i testować.

Namiastki

Choć chce się przeprowadzać testy jednostkowe najszybciej, jak jest to możliwe, często pojawiają się problemy przy testowaniu metod lub klas, które współdziałają z metodami innych klas. Problem polega na tym, iż nie wszystkie metody i klasy są implementowane w tym samym czasie. Jeśli metoda klasy `A` wywołuje metodę zdefiniowaną w klasie `B` (która jeszcze nie jest napisana), testy jednostkowe dla klasy `A` nie mogą zostać wykonane bez pomocy metody zastępczej z klasy `B`. Metoda zastępująca rzeczywistą metodę, która jeszcze nie została zaimplementowana, nosi nazwę **namiastki** lub po prostu **metody zastępczej**. Namiastka ma taki sam nagłówek jak metoda, którą zastępuje, ale wewnątrz zawiera jedynie kod wyświetlający komunikat na temat wywołania metody.

Przykład 2.12

Poniższa metoda jest namiastką metody `save()`. Namiastka umożliwia wywoływanie w testach metody `save()` nawet wtedy, gdy nie została ona jeszcze zaimplementowana.

```
/** Zapis książki telefonicznej.
    pocz: Książka telefoniczna jest wypełniona danymi.
    końc: Zawartość książki telefonicznej zapisana w pliku w postaci
        par nazwisko-numer w kolejnych wierszach,
        pole modified zostaje zmienione na wartość false.
 */
public void save() {
  System.out.println("Wywołano namiastkę metody save().");
  modified = false;
}
```

Poza wyświetleniem komunikatu identyfikacyjnego namiastka możne również wyświetlić wartości parametrów wejściowych i ustawić zmienne, których wartości można łatwo przewidzieć. Jeśli metoda powinna zmieniać stan pola danych, jej namiastka również może dokonywać podobnej zmiany (w przykładzie namiastka ustawia pole `modified` na wartość `false`). Jeśli program kliencki wywołuje jedną lub więcej namiastek, komunikaty wyświetlane w każdej namiastce w momencie jej wywołania pozwalają programiście poznać kolejność wykonywania metod, a tym samym sprawdzić poprawność przepływu sterowania.

Programy testujące

Kolejnym narzędziem do testowania metod jest program testujący. Program ten deklaruje wszystkie niezbędne obiekty i zmienne, przypisuje wartości wejściowe metodom (zgodnie z ich warunkami początkowymi), wywołuje metodę i wyświetla wyniki uzyskane dzięki wykonaniu metod. Ponieważ każda klasa może zawierać metodę `main()`, warto umieścić taką metodę, by służyła jako program testujący dla pozostałych metod klasy. Gdy uruchamia się program napisany w Javie, wykonywanie zaczyna się od metody `main()` wskazanej klasy — metody `main()` pozostałych klas są ignorowane. Metoda `main()` z listingu 2.5 jest programem testującym metodę `search()`.

Testowanie klasy

W podrozdziale 1.5 określiliśmy klasę `DirectoryEntry`. Napisanie kodu tej klasy pozostawiliśmy Czytelnikowi do wykonania jako ćwiczenie. Poniżej przedstawiamy program testujący stanowiący metodę `main()` tej klasy. Program testujący zawiera kilka przypadków testowych.

Wykonanie metody `main()` rozpoczyna się od utworzenia dwóch obiektów `DirectoryEntry` i wyświetlenia ich zawartości (pokazana zostaje zawartość pól `name` i `number`). Przypadki testowe służą do sprawdzenia, czy konstruktor i metody `getName()` i `getNumber()` działają zgodnie z oczekiwaniami.

```
public static void main(String[] args) {
  // Utworzenie kilku wpisów.
  DirectoryEntry tom =
      new DirectoryEntry("Tom", "023-555-4567");
  DirectoryEntry dick =
      new DirectoryEntry("Dick", "021-555-9876");

  // Wyświetlenie zawartości wpisów.
  System.out.println("tom -- nazwisko lub imię:" + tom.getName()
                     + " numer: " + tom.getNumber());
  System.out.println("dick -- nazwisko lub imię:" + dick.getName()
                     + " numer: " + dick.getNumber());
```

Następnie testujemy metodę `equals()`. Ponieważ oba obiekty mają inną zawartość pól `name`, oczekujemy, iż nie uzyskamy równości.

```
// Sprawdzamy, czy są równe.
if (tom.equals(dick))
  System.out.println
      ("BŁĄD -- Tom i Dick są równi");
else
  System.out.println
      ("SUKCES -- Tom i Dick są różni");

if (dick.equals(tom))
  System.out.println
      ("BŁĄD -- Tom i Dick są równi");
else
  System.out.println
      ("SUKCES -- Tom i Dick są różni");
```

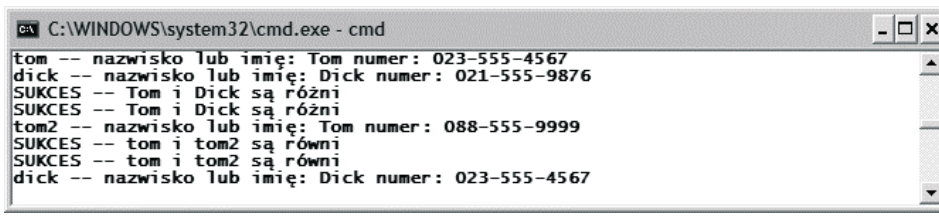
Tworzymy trzeci obiekt `DirectoryEntry` z takim samym polem `name` jak jeden z wcześniejszych obiektów. Oczekujemy uzyskania równości, gdyż pola mają identyczną zawartość.

```
DirectoryEntry tom2 =
    new DirectoryEntry("Tom", "088-555-9999");
System.out.println("tom2 -- nazwisko lub imię:" + tom2.getName()
                   + " numer: " + tom2.getNumber());
// Sprawdzenie, czy oba obiekty są równe.
if (tom.equals(tom2))
  System.out.println
      ("SUKCES -- tom i tom2 są równi");
else
  System.out.println
      ("BŁĄD -- tom i tom2 są różni");

if (tom2.equals(tom))
  System.out.println
      ("SUKCES -- tom i tom2 są równi");
else
  System.out.println
      ("BŁĄD -- tom i tom2 są różni");
```

Na końcu testujemy metodę `setNumber()`, zmieniając numer jednego z obiektów i wyświetlając jego zawartość.

```
dick.setNumber(tom.getNumber());
    // Dick i Tom powinni mieć ten sam numer.
    System.out.println("dick -- nazwisko lub imię:" + dick.getName()
                       + " numer: " + dick.getNumber());
}
```

Rysunek 2.6 przedstawia wyniki testowania klasy `DirectoryEntry`.



Rysunek 2.6. *Test klasy DirectoryEntry*

Wykorzystanie systemu testującego

Program testowy dla klasy `DirectoryEntry` polegał na wizualnej weryfikacji danych wyjściowych. W pewnych przypadkach użytkownik musiał znać spodziewany wynik, natomiast w innych sam program wskazywał, czy test został zaliczony. Można by dodać kod weryfikujący oczekiwane wyniki wszystkich testów, ale znacząco wydłużyłoby to kod.

Program testowy zawierał kilka **przypadków testowych**. Przypadek testowy to pojedynczy test. Zbiór przypadków testowych nosi nazwę **pakietu testów**. Program, który

wykonuje serie testów i wyświetla związane z nimi wyniki, nosi często nazwę **systemu uruchamiania z testowaniem**. Program testowy dla klasy `DirectoryEntry` był zarówno systemem uruchamiania z testowaniem, jak i pakietem testów.

System testujący (**szkielet testowania**) to produkt, który ułatwia pisanie przypadków testowych, organizację przypadków w pakiety testów, wykonywanie pakietów testów i raportowanie wyników. Systemem testującym często stosowanym w projektach pisanych w języku Java jest `JUnit` — bezpłatny produkt typu *open-source*, który może być stosowany jako niezależny moduł. System można pobrać z witryny *junit.org*. Co więcej, system jest dołączany do popularnych zintegrowanych środowisk programistycznych (Eclipse, NetBeans i JBuilder). Poniżej przedstawiamy pakiet testów dla klasy `DirectoryEntry` utworzony za pomocą szkieletu JUnit. Klasa pakietu testów musi rozszerzać klasę `TestCase` (zdefiniowaną w JUnit). Metoda `setUp()` służy do deklaracji obiektów testowych.

```
import junit.framework.*;

public class TestDirectoryEntry extends TestCase {

  private DirectoryEntry tom;
  private DirectoryEntry dick;
  private DirectoryEntry tom2;

  public void setUp() {
    tom = new DirectoryEntry("Tom", "023-456-7890");
    dick = new DirectoryEntry("Dick", "012-656-5490");
    tom2 = new DirectoryEntry("Tom", "011-222-3333");
  }
```

Następnie tworzymy ciąg przypadków testowych, pisząc metody o nazwach `test`*`Xxxxx`*`()`. Na przykład, w celu sprawdzenia, czy obiekt `tom` został wykonany poprawnie, piszemy metodę `testTomCreate()` o następującej treści:

```
public void testTomCreate() {
  assertEquals(tom.getName(), "Tom");
  assertEquals(tom.getNumber(), "023-456-7890");
}
```

Metoda `assertEquals()` zgłosi błąd, jeśli jej argumenty nie będą sobie równe. W ten sposób dowiemy się, iż wyniki metod `getName()` lub `getNumber()` nie są poprawne. Istnieje wiele różnych metod typu `assert`*`Xxxxx`*`()` w JUnit. Nie należy jednak mylić tych metod z instrukcją `assert` Javy, która zostanie omówiona w podrozdziale 2.7.

Metodę `DirectoryEntry.equals()` można przetestować w następujący sposób:

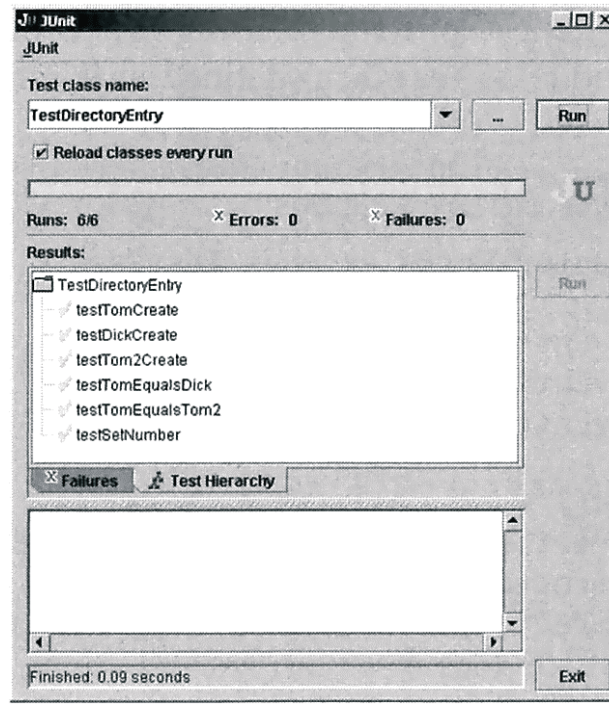
```
public void testTomEqualsDick() {
  assertFalse(tom.equals(dick));
  assertFalse(dick.equals(tom));
}

public void testTomEqualsTom2() {
  assertTrue(tom.equals(tom2));
  assertTrue(tom2.equals(tom));
}
```

Zakładamy, iż `tom` nie jest równe `dick`, więc stosujemy metodę `assertFalse()`. Ponieważ zakładamy, iż `tom` i `tom2` są równe, używamy `assertTrue()` w drugim przypadku.

Na końcu piszemy przypadek testowy dla metody `setNumber()`.

```
public void testSetNumber() {
  dick.setNumber(tom.getNumber());
  assertEquals(tom.getNumber(), dick.getNumber());
}
```

Rysunek 2.7 przedstawia wykonanie pakietu testów pod kontrolą systemu uruchamiania z testowaniem. Bez problemu można stwierdzić, iż wszystkie testy zostały wykonane pomyślnie. Jeśli któryś z testów nie zostałby wykonany poprawnie, pojawiłaby się informacja o nazwie tego testu i numerze wiersza, w którym wystąpiła metoda `assertXxxxx()` powodująca błąd.



Rysunek 2.7. *Test JUnit dla klasy DirectoryEntry*

Testy regresyjne

Plan testów, pakiet testów (namiastki i programy testujące) i wyniki powinny zostać zapamiętane. Gdy tylko dokona się zmian w metodzie, klasie lub programie, powinno się ponownie wykonać testy, by sprawdzić, czy modyfikacje nie spowodowały pojawienia się niechcianych problemów. Ponowne wykonywanie testów w celu sprawdzenia, czy zmiany nie niosą ze sobą nowych problemów, nosi nazwę **testu regresyjnego**.

Testy integracyjne

Kolejnym aspektem testowania systemu są tak zwane testy integracyjne. Test integracyjny pozwala stwierdzić, czy poszczególne komponenty, które zostały sprawdzone w odosobnieniu, będą poprawnie działały z innymi komponentami. W tym kontekście słowo **komponent** oznacza metodę, klasę lub zbiór klas.

Każda kolejna faza testów integracyjnych dotyczy coraz to większych komponentów — przechodzi się od sprawdzania kilku jednostek poprzez klasy aż do całego systemu. Po ukończeniu dwóch jednostek testy integracyjne muszą odpowiedzieć, czy te jednostki będą ze sobą poprawnie współpracować. Po wykonaniu całego systemu testy integracyjne dają odpowiedź na pytanie, czy system jest zgodny z innymi systemami w środowisku, w którym ma być stosowany.

Testy integracyjne zazwyczaj nakierowane są na przypadki użycia. Bazując na informacjach uzyskanych z diagramów sekwencji UML, tester stara się dokonać sprawdzenia programu pod kątem poprawnej interakcji z użytkownikiem i urządzeniami zewnętrznymi.

ĆWICZENIA DO PODROZDZIAŁU 2.5

SPRAWDŹ SIĘ

1. Dlaczego stosowanie strukturalnej ścieżki przejścia przy ręcznej analizie algorytmu jest dobrym pomysłem?
2. Wymień dwa warunki graniczne, które powinny zostać sprawdzone w trakcie testowania metody `readInt()` (patrz listing 2.2).
3. Wyjaśnij dlaczego metoda niespełniająca swojej deklaracji z interfejsu nie zostałaby wykryta w trakcie testów szklanej skrzynki.
4. Przedstaw zbiory danych testowych dla metody `readInt()` z trzema parametrami (listing 2.2), używając:
   a) Testowania szklanej skrzynki.
   b) Testowania czarnej skrzynki.
5. W których fazach testów zostaną przeprowadzone opisane poniżej testy?
   a) Sprawdzenie, czy metoda działa poprawnie dla wszystkich warunków granicznych.
   b) Sprawdzenie, czy klasa `A` może używać klasy `B` jako komponentu.
   c) Sprawdzenie, czy program książki telefonicznej i edytor tekstu mogą być uruchomione jednocześnie i działać poprawnie na jednym komputerze klasy PC.
   d) Sprawdzenie, czy metoda `search()` potrafi przeszukać tablicę zwróconą przez metodę `buildArray()`.

e) Sprawdzenie, czy klasa z tablicowym polem danych może skorzystać z metody statycznej `search()` klasy `ArraySearch`.

PROGRAMOWANIE

1. Napisz program testujący metodę `readInt()`, używając w tym celu danych testowych uzyskanych dzięki odpowiedzi na pytanie 2. z sekcji „Sprawdź się".
2. Napisz namiastkę metody, która mogłaby być stosowana zamiast metody `readInt()`.
3. Napisz metodę `search()` z czterema parametrami: przeszukiwaną tablicą, szukanym elementem, początkowym indeksem wyszukiwania i końcowym indeksem wyszukiwania. Dwa ostatnie parametry określają część tablicy, w której ma nastąpić wyszukiwanie. Metoda powinna zwracać wyjątek w przypadku podania niepoprawnych argumentów. Napisz program testujący tę metodę.

2.6. Usuwanie błędów z programów

W tym podrozdziale zajmiemy się **usuwaniem błędów** z programów (ang. *debugging*) zarówno za pomocą programu uruchomieniowego (debugera), jak i bez niego. Usuwanie błędów to jedno z głównych zadań wykonywanych przez programistów w fazie testów. Testy pozwalają się dowiedzieć, czy istnieje błąd; usuwanie błędów oznacza określenie przyczyny błędu logicznego lub czasu wykonania, a następnie jego poprawienie bez wprowadzania kolejnych błędów. Jeśli stosowało się do sugestii związanych z testowaniem wymienionych w poprzednich podrozdziałach, powinno się być dobrze przygotowanym do usuwania błędów.

Usuwanie błędów przypomina pracę detektywa. Trzeba uważnie śledzić informacje wyświetlane przez program (od samego początku), by sprawdzić, czy to, co się otrzymuje, jest tym, co chciało się otrzymać. Jeśli na przykład wynik zwrócony przez metodę nie jest poprawny, choć przekazane do niej argumenty były prawidłowe, wiadomo, iż problem znajduje się we wnętrzu metody. Można przejść krok po kroku przez wszystkie instrukcje metody, by dokładnie zlokalizować źródło błędów i je poprawić. Jeśli w ten sposób nie udało się odnaleźć przyczyny problemu, potrzeba więcej informacji. Jednym ze sposobów ich zdobycia jest umieszczenie dodatkowych poleceń wyświetlających dane statystyczne. Jeżeli metoda zawiera pętlę, warto wyświetlić wartości zmiennych sterujących pętlą w kolejnych jej iteracjach.

Przykład 2.13

Pętla z listingu 2.6 wydaje się nie przerywać swego działania, gdy użytkownik wpisuje specjalny ciąg znaków ("***"). Pętla kończy się po wpisaniu 10 wyrazów (lub kliknięciu przycisku *Cancel*), ale zwrócony tekst zawiera ciąg znaków, który miał kończyć pętlę.

Listing 2.6. *Metoda getSentence()*

```
/** Zwraca poszczególne słowa wpisane przez użytkownika.
    Aby przerwać wpisywanie, użytkownik może kliknąć przycisk Cancel
    lub wpisać ciąg ***.
    @return Ciąg znaków z maksymalnie 10 wyrazami.
*/

public static String getSentence() {
  String sentence = "";
  int count = 0;
  String word =
      JOptionPane.showInputDialog("Wpisz wyraz lub ***, by zakończyć.");
  while (word != "***" && word != null && count < 10) {
    // Dodaj słowo do sentencji.
    sentence = sentence + word + " ";
    count++;
    word =
        JOptionPane.showInputDialog("Wpisz wyraz lub ***, by zakończyć.");
  }
  return sentence;
}
```

Aby określić źródło problemu, warto wstawić polecenia diagnostyczne wyświetlające wartości zmiennych `word` i `count`, aby się upewnić, iż zmienna `word` jest ustawiana na wartość tekstu przerywającego ("***"). Można wstawić poniższy wiersz:

```
System.out.println("!!! Następny wyraz: " + word + ", wartość count wynosi " +
count);
```

jako pierwszą instrukcję ciała pętli. Jeśli jako trzeci wyraz został wpisany specjalny ciąg, pojawi się następujący tekst:

```
!!! Następny wyraz: ***, wartość count wynosi 2
```

W ten sposób mamy pewność, iż zmienna `word` rzeczywiście zawiera specjalny ciąg, ale ciało pętli wykonuje się nadal. Oznacza to, iż warunek pętli zawiera błąd. Poprawny nagłówek pętli jest następujący:

```
while (word != null && !word.equals("***") && count < 10) {
```

Błąd polegał na tym, iż `word != "***"` porównywało **adres** ciągu znaków zapisanego w `word` z **adresem** stałej tekstowej `"***"`. Nie dochodziło do zamierzonego porównania **zawartości**. Adresy ciągów znaków zawsze będą różne, choć ich zawartość może być identyczna. Do porównania zawartości trzeba zastosować metodę `equals`. Co więcej, najpierw należy sprawdzić, czy `word != null`, by uniknąć zgłoszenia wyjątku `NullPointerException`, gdyby zmienna `word` rzeczywiście zawierała wartość `null`.

Wykorzystanie programu uruchomieniowego

Jeśli korzysta się ze zintegrowanego środowiska programistycznego (IDE), zapewne zawiera ono program uruchomieniowy (debuger). Program uruchomieniowy umożliwia wykonywanie aplikacji krok po kroku zamiast w jednym etapie. Po wykonaniu każdego

kroku debuger zatrzymuje działanie aplikacji, by można było się przyjrzeć zawartości zmiennych. Program uruchomieniowy umożliwia przyjrzenie się wszystkim zmiennym bez potrzeby wstawiania poleceń diagnostycznych. Po zapoznaniu się z zawartością zmiennych przechodzi się do wykonania kolejnego kroku aplikacji.

Wykonywany krok może być tak mały jak pojedyncza instrukcja (wykonywanie **pojedynczymi krokami**), by móc poznać efekt działania instrukcji. Inne rozwiązanie polega na ustawieniu w programie punktów **kontrolnych**, które wstrzymują działanie aplikacji w ściśle określonych miejscach. Debuger wykonuje wszystkie polecenia między kolejnymi punktami kontrolnymi. Jeśli na przykład chce się poznać efekt działania pętli, ale nie chce się przechodzić przez każdą jej iterację, można ustawić punkty kontrolne tuż przed i tuż za pętlą.

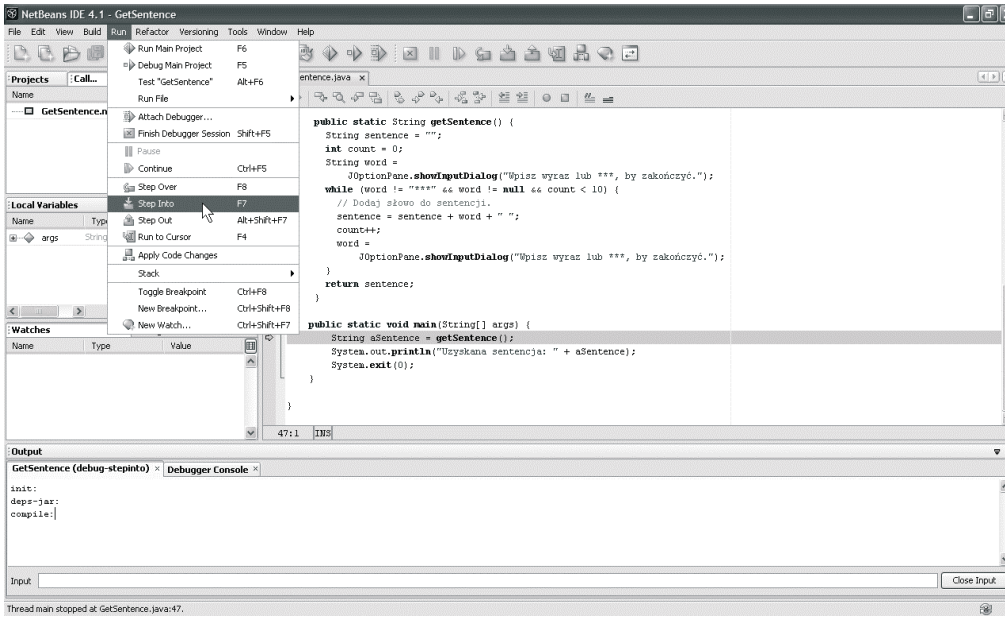
Gdy program znajduje się w stanie wstrzymania, a następne polecenie zawiera wywołanie metody, ma się dwie możliwości: wejście do wnętrza metody (**krok do** metody) lub wykonanie wszystkich poleceń metody jako grupy i zatrzymanie się w momencie zwracania przez nią wartości (**krok nad** metodą).

Szczegóły działania debugera uzależnione są od stosowanego środowiska programistycznego. Ogólny sposób jego stosowania jest jednak zawsze bardzo podobny. Gdy się go zrozumie, bez przeszkód będzie można używać dowolnego debugera. W tym podrozdziale przedstawimy, w jaki sposób korzystać z debugera znajdującego się w środowisku programistycznym NetBeans dostarczanym przez firmę Sun wraz z SDK Javy.

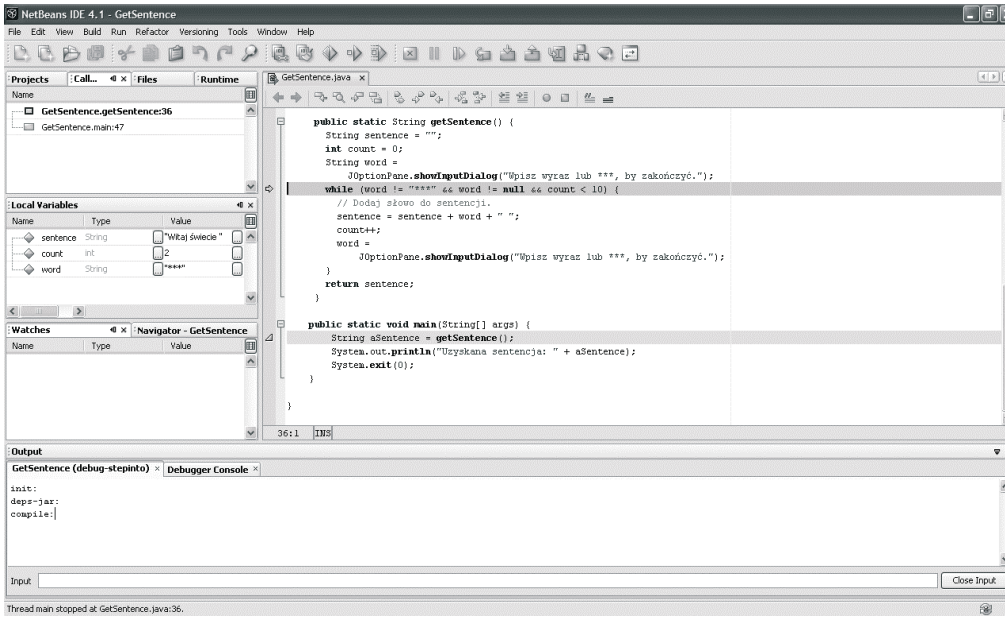
Rysunek 2.8 przedstawia początek pracy z debugerem — wskaźnik wykonywania znajduje się na początku klasy `GetSentence`. Edytor kodu źródłowego wyświetla testowany kod. Menu *Run* zawiera polecenia związane z wykonywaniem kodu. Wybrany element, *Step Into* (krok do), to typowa technika uruchamiania wykonywania krokowego w opisany wcześniej sposób. Okno (na przykład okno *Local Variables*, po lewej mniej więcej na środku) przedstawia na ogół wartości pól danych i zmiennych lokalnych. Metoda `main()` w aktualnym miejscu wykonywania programu posiada tylko jedną zmienną lokalną — tablicę obiektów klasy `String` o nazwie `args`, która jest pusta. Strzałka obok podświetlonego wiersza w oknie kodu źródłowego wskazuje następny krok do wykonania (wywołanie metody `getSentence()`). Wybierz z menu polecenie *Run/Step Into*, aby wykonać poszczególne instrukcje metody.

Rysunek 2.9 przedstawia edytor i okno zmiennych lokalnych (*Local Variables*) po wpisaniu tekstów „Witaj", „świecie" i „***". Zawartość zmiennej `sentence` to `"Witaj świecie"`, wartość zmiennej `count` wynosi 2 natomiast zmienna `word` zawiera tekst `"***"`. Podświetlone jest następne polecenie do wykonania. Jest to nagłówek pętli, który sprawdza warunek jej powtórzenia. Choć oczekujemy, iż warunek będzie równy `false`, okazuje się, że jest równy `true` (dlaczego?) i pętla wykonuje się nadal, dodając „***" do ciągu wyrazów w zmiennej `sentence`.

Zilustrujmy teraz zastosowanie punktów kontrolnych. Rysunek 2.10 przedstawia okno edytora kodu źródłowego z dodanymi dwoma punktami kontrolnymi przed rozpoczęciem wykonywania kodu: jeden punkt znajduje się w wierszu nagłówka pętli, a drugi po zakończeniu pętli. W środowisku NetBeans punkty kontrolne dodaje się, klikając na pionowym pasku na lewo od wiersza, w którym ma znaleźć się punkt kontrolny.

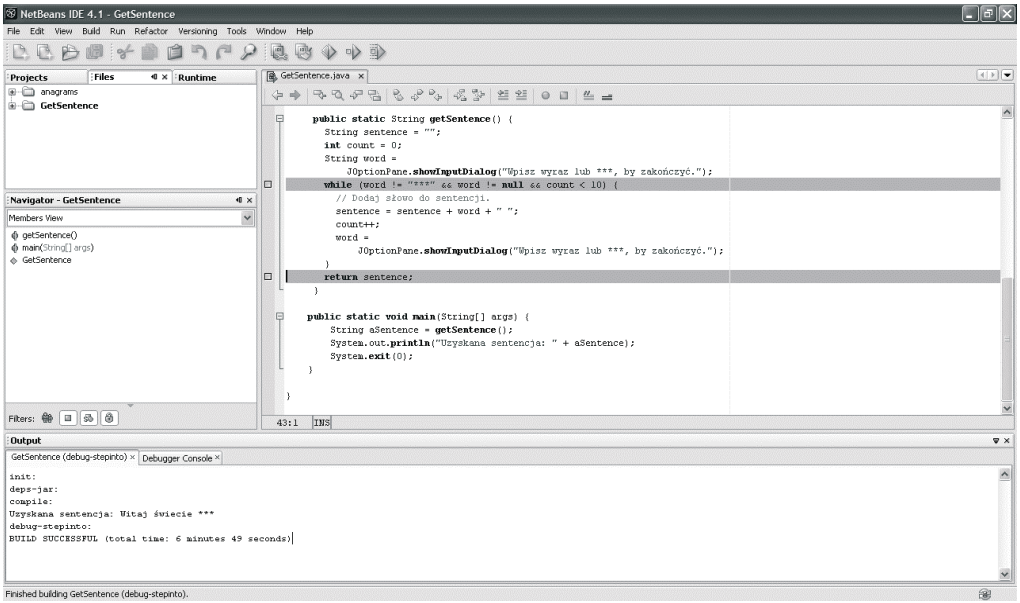


Rysunek 2.8. *Wykorzystanie programu uruchomieniowego środowiska NetBeans*

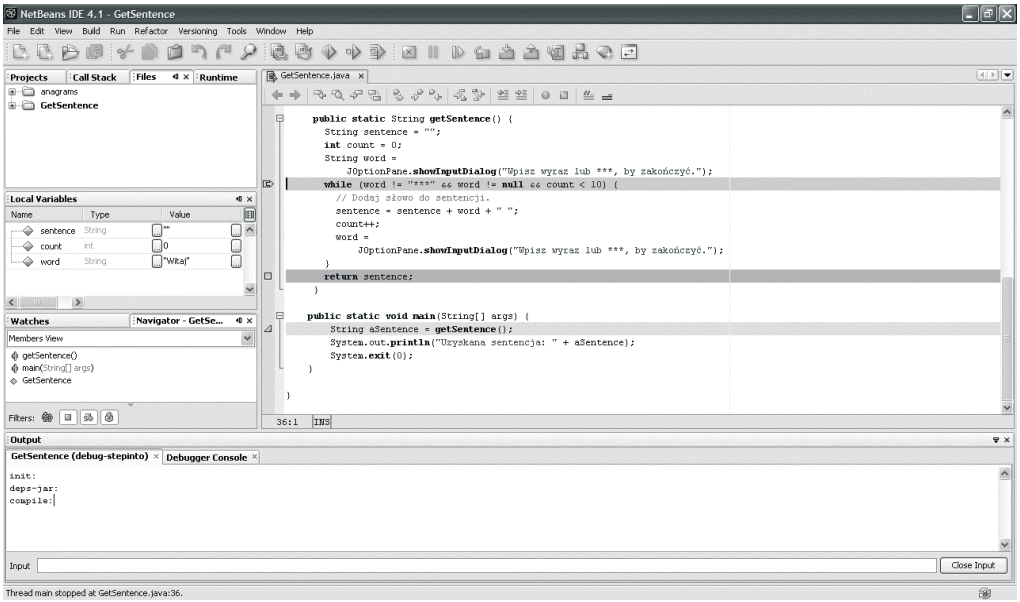


Rysunek 2.9. *Okna edytora kodu źródłowego i debugera*

Małe kwadraciki oznaczają punkty kontrolne. Ponowne kliknięcie kwadracika powoduje wyłączenie punktu kontrolnego. Wybraliśmy najpierw polecenie *Step Into*, a następnie polecenie *Continue*, aby program wyświetlił okno dialogowe (wpisaliśmy w nim tekst „Witaj") i zatrzymał się na pierwszym punkcie kontrolnym (rysunek 2.11). Jeśli



Rysunek 2.10. *Wybranie dwóch punktów kontrolnych*



Rysunek 2.11. *Zatrzymanie programu na punkcie kontrolnym*

ponownie wybierzemy polecenie *Continue*, program wyświetli kolejne okno dialogowe i ponownie zatrzyma się na punkcie kontrolnym związanym z nagłówkiem pętli. Ponowne wybranie polecenia *Continue* i kliknięcie przycisku *Cancal* w oknie dialogowym spowoduje powtórne zatrzymanie się na nagłówku pętli. Kolejne wybranie polecenia *Continue* spowoduje zakończenie pętli i zatrzymanie programu na drugim punkcie kontrolnym.

ĆWICZENIA DO PODROZDZIAŁU 2.6

SPRAWDŹ SIĘ

1. Poniższa metoda wydaje się nie działać poprawnie. Wyjaśnij, gdzie dodałbyś instrukcje wyświetlające dane diagnostyczne i podaj przykłady tych instrukcji. Napisz program testujący działanie metody.

```
/** Znajduje największą wartość w tablicy elementów od x[start] do x[last].
    pocz: first <= last
    @param x Tablica, dla której znajdowany jest największy element.
    @param start Pierwszy indeks zakresu.
    @param last Ostatni indeks zakresu.
    @return Największa wartość od x[start] do x[last].
*/
public int findMax(int[] x, int start, int last) {
  if (start > last)
    throw new IllegalArgumentException("Pusty zakres");
  int maxSoFar = 0;
  for (int i = start; i < last; i++) {
    if (x[i] > maxSoFar)

maxSoFar = i;
  }
  return maxSoFar;
}
```

2. Wyjaśnij różnicę między „krokiem nad" a „krokiem do" w trakcie sprawdzania wywołań metod.

3. Wyjaśnij, dlaczego punkty kontrolne dla metody `getSentence()` zostały umieszczone we wskazanych miejscach?

4. Jak zmieniłoby się wykonanie metody `getSentence()`, gdyby punkt kontrolny został umieszczony tuż przed pętlą zamiast na nagłówku pętli?

PROGRAMOWANIE

1. Po znalezieniu błędu w ćwiczeniu 1. z sekcji **Sprawdź się** podaj poprawną wersję metody. Pozostaw polecenia diagnostyczne, ale wykonuj je tylko wtedy, gdy zmienna globalna `TESTING` zostanie ustawiona na wartość `true`.

2.7. Rozważania na temat programów: asercje i niezmienniki pętli

W podrozdziale 2.5 zostały omówione pewne aspekty testowania programów i systemów. W niniejszym podrozdziale przedstawimy pewne formalne pojęcia, które mają za zadanie pomóc programiście dowieść, iż program lub metoda są poprawne. Skupimy się na asercjach i niezmiennikach pętli, by zapoznać się z rozumowaniem na temat

działania algorytmów. Naszym celem jest jedynie przedstawienie tych pojęć i pokazanie, co można zrobić za ich pomocą. Choć udowodnienie poprawności programu byłoby rzeczą niezwykle pożądaną, na ogół nie można tego zrobić. W praktyce weryfikacja formalna służy jako narzędzie do dokumentowania i rozważań na temat algorytmu — nie jest ścisłym dowodem poprawności. Formalna weryfikacja programów stanowi poważny element badań wielu grup naukowców.

Asercje

Ważnym elementem weryfikacji formalnej jest dokumentowanie programu z wykorzystaniem **asercji**: stwierdzeń logicznych, które zawsze powinny być prawdziwe w trakcie poprawnego wykonywania programu. Asercje często pisze się jako komentarze mówiące, co w danym punkcie programu powinno być prawdą. Warunki początkowe i końcowe również są asercjami. Warunek początkowy to asercja na temat stanu programu (przede wszystkim parametrów wejściowych) przed wykonaniem metody. Warunek końcowy to asercja na temat stanu programu po zakończeniu metody.

Przykład 2.14

Zmodyfikowaliśmy przedstawioną wcześniej metodę `search()`, dodając komentarz asercyjny po zakończeniu pętli.

```
public static int search(int[] x, int target) {
  for (int i = 0; i < x.length; i++) {
    if (x[i] == target)
      return i;
  }

  // asercja: Sprawdzono wszystkie elementy, ale celu nie znaleziono.
  return -1;
}
```

Asercja określa, co powinno być prawdą w danym miejscu programu: sprawdzono wszystkie elementy, ale celu nie znaleziono, ponieważ gdyby cel został znaleziony, wykonałaby się instrukcja `return` we wnętrzu pętli.

Niezmienniki pętli

Wspomnieliśmy wcześniej, iż pętle są bardzo częstym źródłem błędów w programach. Zazwyczaj trudno stwierdzić, czy pętla wykonała się spodziewaną liczbę razy lub czy dokonała spodziewanych modyfikacji w zmiennych programu. Programiści mogą skorzystać ze specjalnego rodzaju asercji, nazywanego **niezmiennikiem pętli** (lub **inwariantem pętli**), aby móc stwierdzić, czy pętla działa zgodnie z oczekiwaniami. Niezmiennik pętli jest instrukcją (wyrażeniem logicznym), który jest prawdziwy przed wykonaniem ciała pętli, na początku każdej iteracji pętli i tuż po zakończeniu pętli. Instrukcja nosi nazwę inwariantu lub niezmiennika, ponieważ zawsze pozostaje prawdziwa w trakcie działania pętli — wykonywanie pętli **nie zmienia jej wartości**. Jeśli

potrafimy określić niezmiennik dla procesu, dla którego ma zostać napisana pętla, i zweryfikować, iż wykonywanie pętli w Javie nie zmienia inwariantu, możemy mieć niemal pewni, iż pętla została napisana poprawnie.

Proponowany niezmiennik dla pętli z metody `search()`, która znajduje pierwsze wystąpienie szukanej wartości, może być taki, że jeśli `i` jest indeksem elementu tablicy sprawdzanego w pętli:

```
dla wszystkich k, takich że 0 <= k < i, x[k] != target
```

Tłumacząc to stwierdzenie na język potoczny, można powiedzieć: „dla wszystkich nieujemnych wartości `k` mniejszych od `i` `k`-ty element `x` nie jest równy poszukiwanej wartości". Sprawdźmy, czy jest to niezmiennik.

- **Niezmiennik musi być prawdziwy przed wykonaniem pętli**. Przed wykonaniem pętli `i` jest równe 0. Nie istnieje nieujemna liczba całkowita mniejsza od 0, więc nie istnieją wartości `k`, a tym samym żadne `x[k]` nie może być równe szukanej wartości.
- **Niezmiennik musi być prawdziwy na początku każdej iteracji pętli**. W pętli porównujemy `x[i]` z `target` i wychodzimy z pętli, jeśli szukana wartość została odnaleziona. Jeśli `x[k]` byłoby równe `target` dla wartości `k < i`, oznaczałoby to, iż szukana wartość została już odnaleziona, opuściliśmy pętlę i nie mogliśmy wykonać kolejnej iteracji. W ten sposób inwariant musi być równy `true`.
- **Niezmiennik musi być prawdziwy zaraz po opuszczeniu pętli**. Wychodzimy z pętli po sprawdzeniu wszystkich elementów, więc `x[k] != target` dla wszystkich `k < x.length`. W ten sposób niezmiennik musi być równy `true`.

Skoro mamy niezmiennik, musimy w języku Java napisać pętlę, która go zachowuje. Oczywiście mamy już taką pętlę, ale udawajmy, że nic o niej nie wiemy.

```
int i = 0;
// niezmiennik: dla wszystkich k, takich że 0 <= k < i, x[k] != target
while (i < x.length) {
  if (x[i] == target)
    return i; // target znalezione dla indeksu i
  i++; // przejście do następnego elementu
}

// asercja: dla wszystkich k, takich że 0 <= k < i, x[k] != target oraz i >= x.length
return -1; // nie odnaleziono docelowej wartości
```

Wykażmy, iż pętla zachowuje niezmiennik. Zmienna `i` ma początkowo wartość 0, więc niezmiennik jest prawdziwy przed rozpoczęciem pętli. Przed każdą iteracją pętli inwariant musi być prawdziwy dla aktualnej wartości `i`, ponieważ w przeciwnym wypadku wykonalibyśmy instrukcję `return` z wartością `k` mniejszą od `i`. Wyjście z pętli ma miejsce, gdy warunek jej działania przestaje być prawdziwy (`i` jest równe `x.length`). Co więcej, takie wyjście z pętli nastąpi tylko wtedy, gdy docelowa wartość nie została odnaleziona w żadnym z elementów tablicy. Oznacza to, iż niezmiennik nadal jest prawdziwy. Ponieważ po zakończeniu pętli niezmiennik jest prawdziwy, ale warunek pętli staje się fałszywy, możemy połączyć oba elementy, by utworzyć asercję po wykonaniu pętli.

Przedstawiony wywód zastosowania niezmiennika pętli służy na ogół jako wskazówka umożliwiająca napisanie pętli, która zachowuje inwariant. Badania informatyczne wykazały, iż pętla będzie poprawna, jeśli napisze się asercję będącą warunkiem początkowym pętli i asercję będącą jej warunkiem końcowym (jest to asercja łącząca niezmiennik pętli z negacją warunku działania pętli). Trzeba wtedy wykazać, iż asercja końcowa wynika z asercji początkowej. W ten sposób uzyskuje się dowód poprawności pętli.

ROZWIĄZANIA PROJEKTOWE

Niezmienniki pętli jako narzędzia projektowe

Można napisać niezmiennik pętli jako specyfikację pętli, a następnie skorzystać z tej specyfikacji do określenia polecenia inicjalizującego pętlę, warunku powtarzania pętli i ciała pętli. Poniższy niezmiennik opisuje pętlę sumującą *n* elementów:

```
// niezmiennik: sum to suma wszystkich odczytanych do tej pory danych, a count zawiera
// liczbę odczytanych do tej pory danych, która jest mniejsza lub równa n.
```

Dzięki temu niezmiennikowi można stwierdzić, iż:

- polecenia inicjalizujące pętlę mają postać:

```
sum = 0.0;
count = 0;
```

- sprawdzenie powtarzania pętli ma postać:

```
count < n
```

- ciało pętli ma postać:

```
next = MyInput.readDouble("Wpisz kolejną wartość");
sum = sum + next;
count = count + 1;
```

Dzięki tym wszystkim informacjom napisanie pętli sumującej staje się dziecinnie łatwe (patrz drugie ćwiczenie programistyczne).

Rozszerzenie tematu — instrukcja assert języka Java

Asercje można również pisać z zastosowaniem instrukcji `assert` języka Java. Instrukcja ta została wprowadzona w wersji 1.4 tego języka. Środowisko kompilatora zawiera specjalny przełącznik, który włącza asercje. Asercje można włączać dla poszczególnych metod lub całego programu. Omówienie instrukcji `assert` wykracza poza ramy niniejszej książki — przedstawimy jedynie jej składnię.

SKŁADNIA Instrukcja assert

POSTAĆ:

```
assert wyrażenie logiczne;
```

PRZYKŁAD:

```
assert x == 5;
```

ZNACZENIE:
Jeśli *wyrażenie logiczne* zwróci wartość `false`, zostanie zgłoszony wyjątek `AssertionError`. Wyjątek ten jest podklasą klasy `Error`, czyli jest wyjątkiem nieweryfikowanym, którego projektanci języka Java zalecają nie wyłapywać.

ĆWICZENIA DO PODROZDZIAŁU 2.7

SPRAWDŹ SIĘ

1. Napisz niezmiennik pętli i asercję dla pętli `while` z metody `readInt()` (listing 2.2).

PROGRAMOWANIE

1. Napisz metodę, która zwraca liczbę cyfr w dowolnej liczbie (`number`). Rozwiązanie powinno zawierać pętlę `while`, dla której prawdziwy jest następujący niezmiennik pętli:

   ```
   // niezmiennik:
   // count >= 0, a number był dzielony przez 10 count razy
   ```

   Po zakończeniu pętli powinna zostać spełniona następująca asercja:

   ```
   // asercja: count zawiera liczbę cyfr przekazanej wartości
   ```

2. Napisz fragment programu implementującego pętlę, której niezmiennik został opisany we wcześniejszej ramce „Niezmienniki pętli jako narzędzia projektowe”.

2.8. Wydajność algorytmów

Korzystając z nowoczesnych komputerów, niełatwo oszacować ilość czasu potrzebną do wykonania programu. Gdy wykona się polecenie:

```
java MojProgram
```

(lub kliknie przycisk *Run* w środowisku programistycznym), system operacyjny najpierw uruchamia maszynę wirtualną Javy. Maszyna wirtualna wczytuje plik *.class* dla `MojProgram` oraz pliki innych klas, z których korzysta program. Dopiero na końcu dochodzi do wykonania właściwego programu (jeśli pliki klas jeszcze nie istnieją, środowisko programistyczne dokona ich kompilacji ze źródeł przed uruchomieniem programu). Większość czasu potrzebnego na uruchomienie i wykonanie programu zostanie poświęconego dwóm pierwszym krokom. Jeśli uruchomi się program po raz drugi tuż po pierwszym uruchomieniu, inicjalizacja może zająć mniej czasu. Wynika to z faktu, iż systemy operacyjne stosują pamięć podręczną do przechowywania ostatnio wczytanych plików. Gdy program jest naprawdę złożony, rzeczywisty czas jego działania będzie w sposób znaczący przewyższał czas potrzebny na załadowanie maszyny wirtualnej i plików *.class*.

Ponieważ bardzo trudno uzyskać precyzyjny pomiar wydajności algorytmu lub programu, staramy się aproksymować efekt zmiany liczby elementów wejściowych na czas działania algorytmu. W ten sposób łatwo stwierdzić, w jaki sposób rośnie czas wykonania algorytmu dla coraz to większej ilości danych, a tym samym porównać dwa algorytmy.

Dla wielu problemów istnieją bardzo proste algorytmy, które mają jedną podstawową wadę — są mało wydajne. Choć komputery stają się coraz szybsze i mają coraz to więcej pamięci, istnieją algorytmy, których krzywa wzrostu jest tak ostra, iż żaden komputer, niezależnie od tego jak szybki i jak dobrze wyposażony w pamięć, nie będzie sobie mógł poradzić z przetwarzaniem większej ilości danych. Nawet jeśli niegdysiejszy bardzo złożony problem można teraz rozwiązać przy użyciu największych i najszybszych superkomputerów, dodanie do niego tylko kilku kolejnych parametrów wejściowych powoduje ponowne problemy z jego wyliczeniem. Z tego powodu trzeba znać chociażby względną wydajność różnych algorytmów. Dzięki trzem kolejnym przykładom prześledzimy, w jaki sposób poznać względną wydajność algorytmu.

Przykład 2.15

Rozważmy następującą metodę, która poszukuje określonej wartości w tablicy:

```
public static int search(int[] x, int target) {
  for (int i = 0; i < x.length; i++) {
    if (x[i] == target)
      return i;
  }

  // celu nie znaleziono
  return -1;
}
```

Jeśli szukana wartość nie znajduje się w tablicy, pętla for wykona się `x.length` razy. Jeśli szukana wartość istnieje w tablicy, może znajdować się gdziekolwiek. Jeśli by rozważyć średnią dla wielu miejsc, w których może się znajdować szukana wartość, pętla wykona się `x.length/2` razy. Oznacza to, iż czas działania jest wprost proporcjonalnie uzależniony od liczby elementów. Jeśli rozmiar tablicy wzrośnie dwukrotnie, czas potrzebny na jej przeszukiwanie również wzrośnie dwukrotnie (nie wliczamy w to początkowego narzutu).

Przykład 2.16

Rozważmy inny problem. Chcemy się dowiedzieć, czy dwie tablice nie mają wspólnych elementów. Możemy wykorzystać metodę `search()` do przeszukania jednej tablicy na podstawie wartości z innej tablicy.

```
/** Sprawdzenie, czy dwie tablice nie mają wspólnych elementów.
    @param x Jedna tablica.
    @param y Druga tablica.
    @return Wartość true, jeśli tablice nie mają wspólnych elementów.
*/
public static boolean areDifferent(int[] x, int[] y) {
  for (int i = 0; i < x.length; i++) {
    if (search(y, x[i]) != -1)
```

```
      return false;
    }
    return true;
  }
```

Ciało pętli zostanie wykonane `x.length` razy. Niestety, pętla korzysta z metody `search()`, która zawiera pętlę wykonywaną `y.length` razy dla każdego wywołania metody. W ten sposób łączny czas wykonania jest proporcjonalny do iloczynu `x.length` i `y.length`.

Przykład 2.17

Rozważmy problem sprawdzenia, czy każdy element tablicy jest unikatowy. Zadanie to wykonuje poniższa metoda:

```
/** Sprawdza, czy wszystkie wartości tablicy są unikatowe.
    @param x Sprawdzana tablica.
    @return Wartość true, jeśli wszystkie elementy tablicy są unikatowe.
*/
public static boolean areUnique(int[] x) {
  for (int i = 0; i < x.length; i++) {
    for (int j = 0; j < x.length; j++) {
      if (i != j && x[i] == x[j])
        return false;
    }
  }
  return true;
}
```

Jeśli wszystkie wartości są unikatowe, zewnętrzna pętla `for` wykona się `x.length` razy. Wewnętrzna pętla dla takiego przypadku również wykona się `x.length` razy. Oznacza to, iż ciało wewnętrznej pętli wykona się łącznie $(\texttt{x.length})^2$ razy.

Przykład 2.18

Metoda przedstawiona w przykładzie 2.17 jest mało wydajna. Wykonuje dwa razy więcej sprawdzeń niż jest to konieczne. Można ją zmodyfikować w następujący sposób:

```
/** Sprawdza, czy wszystkie wartości tablicy są unikatowe.
    @param x Sprawdzana tablica.
    @return Wartość true, jeśli wszystkie elementy tablicy są unikatowe.
*/
public static boolean areUnique(int[] x) {
  for (int i = 0; i < x.length; i++) {
    for (int j = i + i; j < x.length; j++) {
      if (x[i] == x[j])
        return false;
    }
  }
  return true;
}
```

Za pierwszym razem wewnętrzna pętla wykona `x.length-1` iteracji. Za drugim razem wewnętrzna pętla wykona `x.length-2` iteracji itd. Za ostatnim razem wykona tylko jedną iterację. Łączna liczba wykonań będzie więc równa:

```
x.length-1 + x.length-2 + ... + 2 + 1
```

Ciąg 1+2+3+…+(n–1) jest bardzo dobrze znanym ciągiem, którego wartość wynosi

$$n \times \frac{(n-1)}{2}$$

co daje `x.length`×(`x.length`–1)/2 lub 0,5×(`x.length`)^2–0,5×`x.length`.

Notacja dużego O

Ten rodzaj analizy jest obecnie ważniejszy w trakcie prac nad wydajnym oprogramowaniem niż mierzenie czasu wykonania programu w milisekundach na konkretnym komputerze. Zrozumienie, w jaki sposób czas wykonania (i wymagania pamięciowe) rośnie jako funkcja zwiększającej się liczby danych wejściowych, pozwala porównywać różne algorytmy. Naukowcy opracowali użyteczną terminologię i notację ułatwiającą opisywanie związku między liczbą elementów wejściowych a czasem wykonania. Jeśli na przykład czas wykonania zwiększa się dwukrotnie w momencie dwukrotnego zwiększenia liczby danych, oznacza to, iż algorytm ma liniową złożoność obliczeniową. Mówi się wtedy, iż złożoność obliczeniowa jest rzędu n. Jeśli natomiast czas działania zwiększa się czterokrotnie po dwukrotnym zwiększeniu liczby elementów, oznacza to algorytm o kwadratowej złożoności obliczeniowej. Mówi się wtedy, iż złożoność obliczeniowa jest rzędu n^2.

Przedstawiliśmy wcześniej cztery metody: w jednej z nich czas wykonania był związany z `x.length`, w innej był związany z `x.length` razy `y.length`, w jeszcze innej był związany z (`x.length`)^2, a w ostatniej był związany z (`x.length`)^2 i `x.length`. Informatycy używają notacji **O**(n) do oznaczenia pierwszego przypadku, **O**(n×m) do oznaczenia drugiego przypadku, **O**(n^2) do oznaczenia trzeciego i czwartego przypadku. Element n to `x.length`, a element m to `y.length`. Symbol **O** (pisany w wielu książkach z zastosowaniem różnych stylów) warto rozumieć jako „rząd wielkości”. Tego rodzaju notacja nosi nazwę **notacji dużego O** (ang. *big O notation*).

Najprostszym sposobem określenia dużego **O** dla algorytmu lub programu jest przyjrzenie się pętlom i ich ewentualnemu zagnieżdżeniu. Zakładając, iż ciało pętli zawiera tylko proste instrukcje, prosta pętla ma złożoność **O**(n), zagnieżdżona pętla złożoność **O**(n^2), a zagnieżdżona pętla w zagnieżdżonej pętli złożoność **O**(n^3). Warto również sprawdzić, ile razy zostanie wykonana dana pętla.

Rozważmy następujący kod:

```
for (i = 1; i < x.length; i *= 2) {
  // Wykonanie pewnych działań na x[i].
}
```

Ciało pętli zostanie wykonane k–1 razy, natomiast `i` przyjmie wartości: 1, 2, 4, 8, 16, 32, …, 2^k aż do momentu, gdy 2^k stanie się większe od `x.length`. Ponieważ $2^{k-1} \leq$ `x.length` $< 2^k$ i $\log_2 2^k$ wynosi k, wiemy, że $k-1 \leq \log_2($`x.length`$) < k$. Oznacza to, iż złożoność obliczeniowa tej pętli to **O**(log n). Funkcja logarytmu rośnie bardzo powoli. Logarytm o podstawie 2 dla wartości milion to w przybliżeniu około 20. W trakcie określania złożoności algorytmów bardzo często korzysta się z logarytmów o podstawie 2.

Definicja formalna notacji dużego O

Rozważmy program o następującej strukturze:

```
for (int i = 0; i < n; i++) {
  for (int j = 0; j < n; j++) {
    prosta instrukcja
  }
}
for (int k = 0; i < n; k++) {
 prosta instrukcja 1
 prosta instrukcja 2
 prosta instrukcja 3
 prosta instrukcja 4
 prosta instrukcja 5
}
 prosta instrukcja 6
 prosta instrukcja 7
 ...
 prosta instrukcja 30
```

Załóżmy, iż wykonanie każdej prostej instrukcji zajmuje czas jednostkowy, natomiast czas obsługi pętli `for` jest pomijalnie mały. Zagnieżdżona pętla wykonuje się *`prosta instrukcja`* n^2 razy. Pięć *`prosta instrukcja`* wykonuje się n razy. Na końcu wykonuje się 25 *`prosta instrukcja`*. Nietrudno z tego wywnioskować, iż poniższe wyrażenie:

$$T(n) = n^2+5n+25$$

wyraża związek między czasem przetwarzania a n (liczbą elementów danych przetwarzanych w pętli). T(n) reprezentuje czas przetwarzania jako funkcję n.

Formalnie w kategoriach T(n) notacja dużego O:

$$T(n) = \mathbf{O}(f(n))$$

oznacza, iż istnieją dwie stałe, n_0 i c większe od zera, oraz funkcja, f(n), która dla wszystkich $n > n_0$ spełnia zależność $cf(n) \geq T(n)$. Innymi słowy, gdy n staje się wystarczająco duże (większe od n_0), istnieje pewna stała c, dla której czas przetwarzania zawsze będzie mniejszy lub równy $cf(n)$, więc $cf(n)$ staje się górną granicą wydajności. Wydajność nigdy nie będzie gorsza od $cf(n)$, ale może być lepsza.

Jeśli jesteśmy w stanie określić, w jaki sposób wartość f(n) zwiększa się dla kolejnych n, będziemy wiedzieć, w jaki sposób rośnie czas przetwarzania dla kolejnych n. Często szybkość wzrostu f(n) określa najszybciej rosnąca składowa funkcji f(n) (ta o największym wykładniku). W przedstawionym przykładzie jest to n^2. Oznacza to, iż przedstawiony algorytm ma złożoność obliczeniową **O**(n^2) zamiast **O**(n^2+5n+25). Ogólnie

bezpiecznie jest zignorować wszystkie stałe i pominąć składowe niższych rzędów w trakcie określania rządu wielkości dla algorytmu.

Przykład 2.19

Dla $T(n) = n^2+5n+25$ chcemy wykazać, iż jest to rzeczywiście złożoność $\mathbf{O}(n^2)$. Innymi słowy, chcemy pokazać, iż istnieją stałe n_0 i c takie, że dla wszystkich $n > n_0$, $cn^2 > n^2+5n+25$.

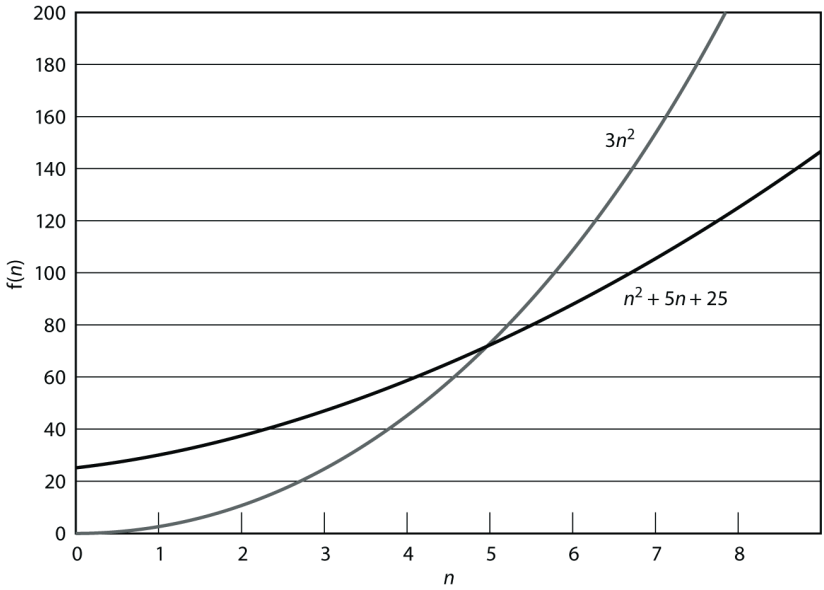
Musimy rozwiązać równość:

$$cn^2 = n^2+5n+25$$

Jeśli zamienimy n na n_0 i rozwiążemy równość względem c, otrzymamy:

$$c = 1+5/n_0+25/n_0^2$$

Dla n_0 równego 5 uzyskamy wartość c równą 3. Z tego powodu $3n^2 > n^2+5n+25$ dla wszystkich n większych od 5, co przedstawia rysunek 2.12.



Rysunek 2.12. *$3n^2$ kontra $n^2+5n+25$*

Przykład 2.20

Rozważmy następującą pętlę:

```
for (int i = 0; i < n - 1; i++) {
  for (int j = i + 1; j < n; j++) {
    trzy proste instrukcje
  }
}
```

Przy pierwszej iteracji pętli zewnętrznej pętla wewnętrzna wykona się n–1 razy. Dla kolejnej iteracji wykona się n–2 razy, a przy ostatniej iteracji tylko raz. Pętla zewnętrzna wykona się n–1 razy. Otrzymujemy następujące wyrażenie dla T(n).

$$3(n-1)+3(n-2)+\ldots+3$$

Trójkę można wyciągnąć przed nawias:

$$3(n-1+n-2+\ldots+1)$$

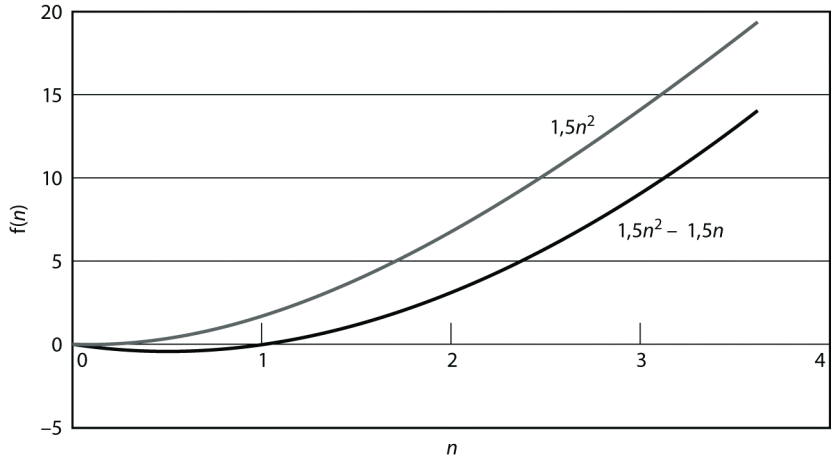
Suma 1+2+…+n–1 (w nawiasie) wynosi:

$$\frac{n\times(n-1)}{2}$$

Oznacza to, iż końcowe T(n) ma wartość:

$$T(n) = 1{,}5n^2-1{,}5n$$

Wielomian ma wartość zero dla n równego 1. Dla wartości większych od 1, $1{,}5n^2$ zawsze jest większe od $1{,}5n^2-1{,}5n$. Oznacza to, iż możemy zastosować 1 jako n_0 i 1,5 jako c, aby wykazać, iż dla T(n) uzyskujemy $\mathbf{O}(n^2)$ (patrz rysunek 2.13).



Rysunek 2.13. *$1{,}5n^2$ kontra $1{,}5n^2-1{,}5n$*

Jeśli T(n) jest wielomianem stopnia d (najwyższy wykładnik), wtedy złożoność obliczeniowa wynosi $\mathbf{O}(n^d)$. Przedstawienie dowodu matematycznego dowodzącego słuszności tego twierdzenia wykracza poza ramy niniejszej książki. Dowód intuicyjny został przedstawiony w dwóch wcześniejszych przykładach. Jeśli pozostałe elementy wielomianu mają dodatnie współczynniki, należy znaleźć wartość n, dla której pierwszy element jest równy wszystkim pozostałym elementom. Gdy n stanie się większe od tej wartości, element n^d zawsze będzie większy.

Wyrażenie **O**(1) służy do określenia stałej złożoności obliczeniowej. W takim przypadku działanie algorytmu nie jest uzależnione od liczby elementów wejściowych. Wszystkie proste kroki reprezentują złożoność **O**(1). Każda skończona liczba kroków **O**(1) nadal ma złożoność **O**(1).

Podsumowanie notacji

W niniejszym podrozdziale zostały wprowadzone symbole T(n), f(n) i **O**(f(n)). Tabela 2.5 podsumowuje ich znaczenie.

Tabela 2.5. *Symbole używane w określaniu wydajności algorytmów*

T(n)	Czas, jaki zajmuje wykonanie metody lub programu jako funkcja liczby elementów wejściowych (n). Nie zawsze można go określić w sposób dokładny.
f(n)	Dowolna funkcja parametru n. Ogólnie f(n) oznacza prostszą funkcję niż T(n), na przykład n^2 zamiast $1{,}5n^2–1{,}5n$.
O(f(n))	Rząd wielkości. **O**(f(n)) to zbiór funkcji, które rosną nie szybciej niż f(n). Mówimy, iż T(n) = **O**(f(n)), aby podkreślić, iż wzrost T(n) jest powiązany ze wzrostem f(n).

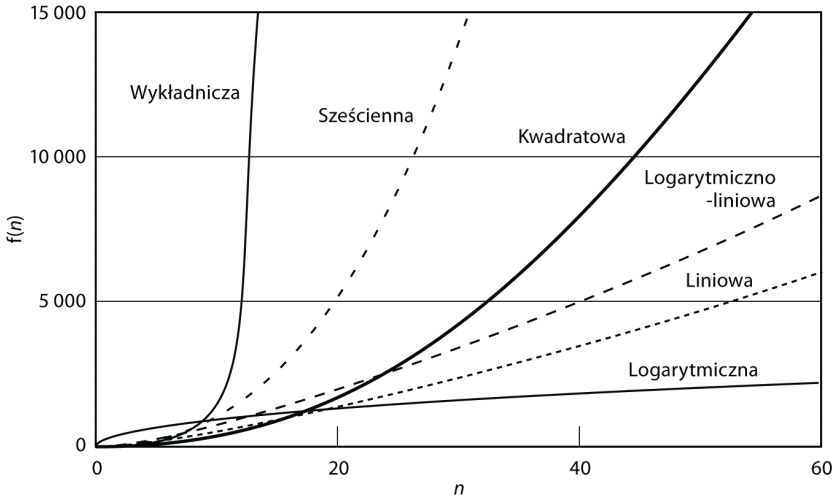
Porównanie wydajności

W niniejszej książce przedstawimy wiele algorytmów i opiszemy, jak rośnie ich czas wykonania oraz wymagania pamięciowe wraz ze zwiększaniem liczby elementów. Złożoność obliczeniową będziemy przedstawiać w notacji dużego O. Istnieje kilka podstawowych typów złożoności obliczeniowej. Zostały one podsumowane w tabeli 2.6.

Tabela 2.6. *Typowe złożoności obliczeniowe*

Duże O	Nazwa złożoności
O(1)	stała
O(log n)	logarytmiczna
O(n)	liniowa
O(n log n)	logarytmiczno-liniowa
O(n^2)	kwadratowa
O(n^3)	sześcienna
O(2^n)	wykładnicza 2^n
O(n!)	silnia

Rysunek 2.14 przedstawia szybkość wzrostu funkcji logarytmicznej, liniowej, logarytmiczno-liniowej, kwadratowej, sześciennej i wykładniczej. Warto zauważyć, iż dla małych wartości n funkcja wykładnicza przyjmuje mniejsze wartości niż wszystkie pozostałe. Dopiero dla n większego od 20 funkcja liniowa staje się mniejsza od funkcji kwadratowej. Wykres wskazuje dwie istotne kwestie. Dla małej liczby elementów mało wydajne algorytmy potrafią być tak naprawdę bardziej wydajne. Jeśli się wie, iż przetwarzanie zawsze będzie dotyczyło niewielkiej ilości danych, algorytm **O**(n^2) może okazać się bardziej pożądany od algorytmu **O**(n log n) o dużym współczynniku stałym.



Rysunek 2.14. *Funkcje odpowiadające różnym złożonościom obliczeniowym*

Z drugiej strony algorytmy, które dla niewielkich zbiorów danych działają bardzo szybko (algorytmy wykładnicze), dla większej liczby danych potrafią być niezwykle kosztowne.

Suche liczby z rysunku 2.14 potrafią być mylące. Głównym powodem jest fakt, iż notacja dużego **O** ignoruje wszystkie stałe. Algorytm o złożoności logarytmicznej może okazać się znacznie trudniejszy do zaprogramowania (i wymagać więcej czasu na analizę każdego elementu) niż algorytm o złożoności liniowej. Na przykład, dla $n = 25$ na rysunku 2.14 czas przetwarzania dla złożoności logarytmicznej wynosi 1800, natomiast dla złożoności liniowej wynosi 2500. Porównanie tego rodzaju ma niewielkie znaczenie. Dla tego stosunkowo niewielkiego zbioru danych algorytm logarytmiczny może zająć więcej czasu niż algorytm liniowy. Ważna jest zmiana ich złożoności w przypadku zmiany liczby elementów wejściowych.

Przykład 2.21

Prześledźmy zmianę złożoności, gdy dwukrotnie zwiększy się liczbę elementów (powiedzmy z 50 do 100). Wyniki przedstawia tabela 2.7. Trzecia kolumna przedstawia stosunek czasu przetwarzania między dwoma grupami elementów. W ten sposób łatwo stwierdzić, że dla złożoności $\mathbf{O}(n \log n)$ przetworzenie 100 elementów potrwa 2,35 razy dłużej niż przetworzenie 50 elementów.

Algorytmy o złożoności wykładniczej

Algorytmy o złożoności wykładniczej posiadają praktyczny górny limit rozmiaru problemu, do którego rozwiązania mogą zostać użyte (nawet przy zastosowaniu najszybszych komputerów). Jeśli na przykład wykonanie algorytmu $\mathbf{O}(2^n)$ zajmuje godzinę dla 100 danych, dodanie jeszcze jednego elementu spowoduje wydłużenie obliczeń o kolejną godzinę, dodanie pięciu kolejnych elementów wydłuży czas pracy algorytmu do

Tabela 2.7. *Efekt zwiększenia liczby elementów dla różnych złożoności*

O(f(n))	f(50)	f(100)	f(100)/f(50)
O(1)	1	1	1
O(log *n*)	5,64	6,64	1,18
O(*n*)	50	100	2
O(*n* log *n*)	282	664	2,35
O(n^2)	2500	10 000	4
O(n^3)	12 500	100 000	8
O(2^n)	$1{,}126\times10^{15}$	$1{,}27\times10^{30}$	$1{,}126\times10^{15}$
O(*n*!)	$3{,}0\times10^{64}$	$9{,}3\times10^{157}$	$3{,}1\times10^{93}$

32 godzin (ponad jeden dzień!), a dodanie 14 elementów spowoduje wykonywanie algorytmu przez 16 384 godziny, co daje prawie dwa lata!

Ten związek jest podstawą algorytmów kryptograficznych. Niektóre algorytmy kryptograficzne można złamać w czasie $\mathbf{O}(2^n)$, gdzie n jest długością klucza. Klucz o długości 40 bitów można złamać, używając nowoczesnych komputerów, ale klucz od długości 100 bitów nie zostanie złamany, ponieważ wymaga to około 10^{18} (miliard miliardów) więcej czasu niż dla algorytmu 40-bitowego.

ĆWICZENIA DO PODROZDZIAŁU 2.8

SPRAWDŹ SIĘ

1. Określ, ile razy zostanie wyświetlony tekst z ciała wewnętrznej pętli dla każdego z wymienionych fragmentów. Określ, czy algorytm ma złożoność $\mathbf{O}(n)$ czy raczej $\mathbf{O}(n^2)$.

a)
```
for (int i = 0; i < n; i++)
  for (int j = 0; j < n; j++)
    System.out.println(i + "  " + j);
```

b)
```
for (int i = 0; i < n; i++)
  for (int j = 0; j < 2; j++)
    System.out.println(i + "  " + j);
```

c)
```
for (int i = 0; i < n; i++)
  for (int j = n - 1; j >= i; j--)
    System.out.println(i + "  " + j);
```

d)
```
for (int i = 1; i < n; i++)
  for (int j = 0; j < i; j++)
    if (j % i == 0)
      System.out.println(i + "  " + j);
```

2. Dla poniższego T(*n*) znajdź takie wartości n_0 i *c*, aby cn^3 było większe od T(*n*) dla wszystkich *n* większych od n_0:

$T(n) = n^3 - 5n^2 + 20n - 10$

3. W jaki sposób zmieni się wydajność, jeśli liczba elementów zwiększy się z 2000 do 4000 dla następujących złożoności. Odpowiedz na to samo pytanie, gdy liczba elementów zwiększy się z 4000 do 8000. Wykonaj tabelę podobną do tabeli 2.7.

   a) $\mathbf{O}(\log n)$

   b) $\mathbf{O}(n)$

   c) $\mathbf{O}(n \log n)$

   d) $\mathbf{O}(n^2)$

   e) $\mathbf{O}(n^3)$

4. Zgodnie z rysunkiem 2.14, jakie są czasy przetwarzania dla $n = 20$ i $n = 40$ dla przedstawionych złożoności?

PROGRAMOWANIE

1. Napisz program, który porównuje wartości y1 i y2 uzyskane dzięki poniższym wyrażeniom dla wartości n od 0 do 100 z krokiem 10. Czy wyniki okazały się zaskakujące?

```
y1 = 100 * n + 10
y2 = 5 * n * n + 2
```

Podsumowanie rozdziału

- Istnieją trzy rodzaje błędów w programach. Błędy składniowe, które uniemożliwiają poprawną kompilację programu, są najprostsze do naprawienia. Wynikają na ogół z pomyłki typograficznej lub niezrozumienia składni języka. Nie wszystkie błędy typograficzne powodują zgłoszenie błędu składniowego.
- Błędy czasu wykonania to, jak sama nazwa wskazuje, błędy objawiające się w trakcie pracy programów (na ogół są to wyjątki). Można zmniejszyć liczbę tego rodzaju błędów, korzystając z instrukcji `if` do testowania poprawności zmiennych przed ich użyciem, jeśli mogłoby to spowodować zgłoszenie wyjątku.
- Błędy logiczne występują wtedy, gdy program nie zwraca poprawnych wyników. Czasem błędy logiczne łatwo odnaleźć, gdyż program za każdym razem generuje złe wyniki. Znacznie trudniej odnaleźć błędy logiczne, które zdarzają się niezmiernie rzadko i niełatwo je reprodukować.
- Wszystkie wyjątki z hierarchii klasy `Exception` wywodzą się z głównej klasy bazowej o nazwie `Throwable`. Klasa ta zawiera metody dotyczące ustawiania i raportowania stanu programu w momencie wystąpienia wyjątku. Najczęściej stosowane są metody `getMessage()` i `toString()`, które zwracają szczegółowe informacje na temat powodu powstania wyjątku. Metoda `printStackTrace()` wyświetla wyjątek oraz sekwencję wywołań metod, która doprowadziła do jego zgłoszenia.

- Domyślnie maszyna wirtualna wyświetla komunikat błędu i stos wywołań metod, a następnie zamyka program, jeśli wyjątek nie został wyłapany. Aby samemu wyłapać i obsłużyć wyjątek, należy zastosować sekwencję `try-catch-finally`. Unika się wtedy domyślnego zachowania i można spróbować naprawić błąd.
- Istnieją dwa rodzaje wyjątków: weryfikowane i nieweryfikowane. Wyjątki weryfikowane dotyczą na ogół błędnej sytuacji, która wystąpiła na zewnątrz aplikacji. Wyjątki nieweryfikowane wynikają na ogół z błędu programisty lub nietypowego splotu zdarzeń.
- Metoda, która może zgłosić wyjątek weryfikowany, musi albo go wyłapać, albo zadeklarować, iż może go zgłosić, stosując klauzulę `throws`. Jeśli wyjątek zostaje przekazany wyżej, musi zostać wyłapany przez jedną z metod wyższego poziomu. Metody nie muszą wyłapywać wyjątków nieweryfikowanych ani deklarować w klauzuli `throws`, że będą takie zgłaszać.
- Instrukcja `throw` służy do zgłaszania wyjątków, gdy wykryje się sytuację nietypową w trakcie wykonywania metody. Należy wyłapać wyjątek w innym miejscu programu. W przeciwnym razie zostanie on przetworzony przez maszynę wirtualną i potraktowany jako wyjątek niewyłapany.
- Testowanie programów przeprowadza się na kilku poziomach, zaczynając od najmniejszej części, którą da się przetestować — jednostki. Jednostka może być metodą lub klasą, w zależności od złożoności.
- Gdy jednostki zostaną już przetestowane samodzielnie, należy przystąpić do testowania ich wzajemnych powiązań. Są to tak zwane testy integracyjne.
- Gdy udało się poskładać cały program, należy przetestować go jako całość. Są to tak zwane testy systemowe.
- Na końcu dokonuje się testów operacyjnych demonstrujących funkcjonalność programu. Są to testy akceptacyjne.
- Testy czarnej skrzynki (nazywanej też skrzynką zamkniętą) sprawdzają element (jednostkę lub system) na podstawie wymagań funkcjonalnych bez korzystania z wiedzy na temat jego wewnętrznej struktury.
- Testy szklanej skrzynki (nazywanej też skrzynką otwartą) sprawdzają element, wykorzystując wiedzę na temat jego wewnętrznej struktury. Jednym z celów tego rodzaju testów jest wykonanie testów pokrycia. Polegają one albo na przetestowaniu każdej instrukcji przynajmniej raz, albo na wykonaniu choć raz każdego rozgałęzienia (instrukcji `if` i `switch` oraz pętli), by sprawdzić wszystkie dostępne ścieżki programu.
- Programy testujące i namiastki to narzędzia użyteczne w trakcie testowania. Program testujący sprawdza metodę lub klasę, wykonując określone testy. Namiastka istnieje zamiast rzeczywistej metody, z której korzysta testowana jednostka. Umożliwia to testowanie pomimo tego, iż rzeczywista metoda, z której korzysta jednostka, nie została jeszcze napisana.
- Przedstawiliśmy proces poszukiwania błędów, a także sposób, w jaki wykorzystać program uruchomieniowy do zdobycia informacji na temat stanu programu.

- Omówiliśmy weryfikację formalną programów oraz sposób, w jaki asercje i niezmienniki pętli pomagają zaprojektować i zweryfikować programy.
- Informatycy są często zainteresowani porównywaniem wydajności różnych algorytmów. Przedstawiliśmy notację dużego O oraz sposoby jej określania dzięki analizie wykonań pętli. Z notacji będziemy korzystali w kolejnych rozdziałach, by opisać względną wydajność różnych struktur danych i operujących na nich algorytmów.

Konstrukcje języka Java wprowadzone w rozdziale

`catch`	`throws`
`finally`	`try`
`throw`	

Klasy interfejsu programistycznego Javy wprowadzone w rozdziale

`Throwable`	`IllegalArgumentException`
`IOException`	`NumberFormatException`
`EOFException`	`InputMismatchException`
`NullPointerException`	`FileNotFoundException`
`NoSuchElementException`	`ArithmeticException`
`RuntimeException`	`ArrayIndexOutOfBoundsException`

Własne interfejsy i klasy zdefiniowane w rozdziale

`class ArraySearch`	`class MyInput`
`class DirectoryEntry`	`class TestDirectoryEntry`

Krótki test

1. Jakie są trzy główne kategorie błędów omawiane w niniejszym rozdziale?
2. Jakie jest zadanie strukturalnej ścieżki przejścia w algorytmie?
3. Testy _____ wymagają użycia danych testowych, które sprawdzają każdą instrukcję modułu.
4. Testy _____ skupiają się na sprawdzeniu charakterystyki funkcjonalnej modułu.
5. _____ pozwalają sprawdzić, czy program zawiera błędy. _____ pozwala odnaleźć _____ błędu i pomaga go _____.
6. Zakładając, iż w klauzulach `catch` występujących po bloku `try` znajdują się wymienione wyjątki, ułóż je w takiej kolejności, by nie spowodować zgłoszenia błędu składniowego przez kompilator.

```
Exception
IOException
FileNotFoundException
```

7. Wskaż, który z wymienionych elementów może być fałszywy: niezmiennik pętli, warunek pętli `while`, asercja.

8. Napisz niezmiennik pętli dla następującego fragmentu kodu:

```
product = 1;
counter = 2;
while (counter < 5) {
  product *= counter;
  counter++;
}
```

9. Określ złożoność obliczeniową w notacji dużego O dla algorytmu, którego czas działania można wyliczyć dzięki funkcji $T(n) = 3n^4–2n^2+100n+37$.

10. Jeśli pętla przetwarza *n* elementów i liczba elementów została zmieniona z 1024 na 2048, w jaki sposób wpłynie to na czas wykonania pętli $\mathbf{O}(n^2)$? Jak wpłynęłoby to na pętlę $\mathbf{O}(\log n)$? A co z pętlą typu $\mathbf{O}(n \log n)$?

Odpowiedzi do krótkiego testu

1. Błędy składniowe, błędy czasu wykonania i błędy logiczne.

2. Zwiększa prawdopodobieństwo znalezienia w programie błędów logicznych.

3. Testy **szklanej skrzynki** wymagają użycia danych testowych, które sprawdzają każdą instrukcję modułu.

4. Testy **czarnej skrzynki** skupiają się na sprawdzeniu charakterystyki funkcjonalnej modułu.

5. **Testy** pozwalają sprawdzić, czy program zawiera błędy. **Poszukiwanie błędu** pozwala odnaleźć **przyczynę** błędu i pomaga go **naprawić**.

6. Wyjątki w klauzulach muszą być ułożone w kolejności `FileNotFoundException`, `IOException` i `Exception`.

7. Warunek pętli `while`.

8. Niezmiennik: `product` zawiera iloczyn wszystkich dodatnich liczb całkowitych mniejszych od `counter`, natomiast `counter` przyjmuje wartości od 2 do 5 włącznie.

9. $\mathbf{O}(n^4)$

10. Czas działania zwiększa się czterokrotnie dla $\mathbf{O}(n^2)$. Dla pętli $\mathbf{O}(\log n)$ zwiększa się o współczynnik 1,1 (11/10). Dla $\mathbf{O}(n \log n)$ zwiększa się o współczynnik 2,2 (2048×11/1024×10).

Pytania sprawdzające

1. Opisz technikę zapobiegania błędowi czasu wykonania zgłaszanemu, gdy użytkownik wpisze zamiast oczekiwanej liczby inny tekst.

2. Opisz różnicę miedzy namiastką a programem testującym metody lub klasy.

3. Omów różnicę między klauzulą `throws` i instrukcją `throw`. Kiedy korzysta się z każdej z nich? Co trzeba zrobić, gdy zdecyduje się zgłosić wyjątek zamiast go wyłapać? Podaj dwie sytuacje, w których warto przekazać wyjątek wyżej zamiast podejmować się jego obsługi.

4. Wskaż, jaki wyjątek może zgłosić każdy z wymienionych poniżej błędów. Podaj, czy będzie to wyjątek weryfikowany czy nieweryfikowany.

a) Próba utworzenia obiektu `BufferedReader` dla pliku, który nie istnieje.

b) Próba wywołana metody dla zmiennej, która nie został zainicjalizowana.

c) Użycie wartości –1 jako indeksu tablicy.

d) Wywołanie metody `nextToken()` klasy `StringTokenizer` bez sprawdzenia metodą `hasMoreTokens()`, czy są jeszcze jakieś dane do pobrania.

5. Omów pokrótce plan testów dla programu książki telefonicznej z rozdziału 1. Załóż zastosowanie testów integracyjnych.

6. Wskaż, na którym etapie testów (jednostkowym, systemowym, integracyjnym) zostaną wykryte wymienione niżej błędy:

a) Indeks tablicy poza prawidłowymi granicami.

b) Zgłoszenie wyjątku `FileNotFoundException`.

c) W pewnych przypadkach zostaje wyliczona niepoprawna wartość podatku.

7. Które z poniższych stwierdzeń nie są prawdziwe?

a) Niezmienniki pętli stosuje się do weryfikacji pętli.

b) Niezmienniki pętli stosuje się do projektowania pętli.

c) Niezmiennik pętli zawsze jest asercją.

d) Asercja zawsze jest niezmiennikiem pętli.

8. Napisz metodę, która zlicza, ile sąsiadujących elementów nie jest na miejscu (zakłada się porządek rosnący). Dodaj niezmienniki pętli i inne asercje, by zweryfikować poprawność pętli.

9. Określ złożoności obliczeniową w notacji dużego O dla następujących pętli:

a)
```
for (int i = 1; i <=n; i++)
  for (int j = 1; j <= n; j++)
    for (int k = n; k >= 1; k--) {
      sum = i + j + k;
      System.out.println(sum);
    }
```

b)
```
for (int i = 0; i < n; i++)
  for (int j = 0; j < i * i; j++)
    System.out.println(j);
```

c)
```
for (int i = n; i >= 0 i -= 2)
  System.out.println(i);
```

d)
```
for (int i = 0; i < n; i++)
  for (int j = i; j > 0; j /= 2)
    System.out.println(j);
```

Projekty programistyczne

1. Napisz program określający średnią liczbę lokalizacji w tablicy, które zostały sprawdzone w celu odnalezienia poszukiwanego elementu. Tablica zawiera 100 nieposortowanych liczb całkowitych, a wyszukiwanie odbywa się w sposób sekwencyjny. Program powinien również obliczyć średnią liczbę sprawdzonych elementów, jeśli poszukiwana wartość nie znajdowała się w tablicy. Obliczenia średnich powinny bazować na wartościach uzyskanych w trakcie znajdowania 50 różnych wartości.
2. Powtórz pierwszy projekt, ale niech tablica zawiera elementy posortowane rosnąco. Zmodyfikuj algorytm w taki sposób, by przerywał swe działanie, gdy tylko zauważy, iż wartości tablicy stają się większe od poszukiwanej wartości.
3. Napisz program, który umożliwia sprawdzenie wpływu zmiany rozmiaru tablicy i porządku danych w tablicy na działanie ulubionego algorytmu sortowania. Dokonaj testów dla tablicy o trzech rozmiarach (100, 1 000 i 10 000 elementów) i trzech różnych ułożeniach elementów (w porządku rosnącym, w porządku malejącym i losowo). Wykonanie zadania powinno zwrócić dziewięć wyników. Do utworzenia tablicy z losowymi wartościami można posłużyć się metodą `Math.random()`. Jeśli nie ma się ulubionego algorytmu sortowania, można skorzystać z metody `Arrays.sort()` z pakietu `java.util`.
4. Napisz zbiór namiastek metod dla klasy `ArrayBasedPD` (podrozdział 1.5), które mogłyby zostać użyte w trakcie testowania klas `PDGUI` i `PDConsoleUI` (podrozdział 1.8).
5. Zaprojektuj i napisz testy szklanej skrzynki dla klasy, która wylicza funkcje sinus i kosinus w wyspecjalizowany sposób. Klasa ma być częścią systemu wbudowanego działającego na procesorze bez arytmetyki zmiennoprzecinkowej i klasy `Math` języka Java. Testowaną klasę przedstawia listing 2.9. Należy przetestować metody `sin()` i `cos()`, zakładając, iż metody `sin0to45()` i `sin45to90()` zostały już sprawdzone. Należy zaprojektować zbiór danych testowych, które pozwolą sprawdzić wszystkie instrukcje `if`. W tym celu warto przyjrzeć się warunkom granicznym i wybrać wartości znajdujące się:
   - dokładnie na granicy,
   - blisko granicy,
   - między granicami.

Listing 2.9. *Plik SinCos.java (poza elementami programu testującego)*

```
/** Klasa oblicza sinus i kosinus dla kąta wyrażonego w stopniach.
    Wynikiem jest liczba całkowita reprezentująca sinus lub kosinus
    pomnożony przez 10 tysięcy. Na przykład wynik 7071 reprezentuje
    wartość 7071e-4 lub 0.7071.
```

```
 */
public class SinCos {
  /** Oblicza sinus kąta w stopniach.
      @param x Kąt w stopniach.
      @return Sinus kąta.
   */
  public static int sin(int x) {
    if (x < 0) {
      x = -x;
    }
    x = x % 360;
    if (0 <= x && x <= 45) {
      return sin0to45(x);
    }
    else if (45 <= x && x <= 90) {
      return sin45to90(x);
    }
    else if (90 <= x && x <= 180) {
      return sin(180 - x);
    }
    else {
      return -sin(x - 180);
    }
  }

  /** Oblicza kosinus kąta w stopniach.
      @param x Kąt w stopniach.
      @return Kosinus podanego kąta.
   */
  public static int cos(int x) {
    return sin(x + 90);
  }

  /** Oblicza sinus kąta w stopniach między 0 a 45.
      pocz: 0 <= x < 45
      @param x Kąt.
      @return Sinus kąta x.
   */
  private static int sin0to45(int x) {
    // W rzeczywistym programie metoda ta używałaby
    // aproksymacji wielomianowej zoptymalizowanej dla
    // stosowanego zakresu wartości wyjściowych.

    // Kod wyliczający sin(x) dla x w zakresie od 0 do 45 stopni.

  }

  /** Oblicza sinus kąta w stopniach między 45 a 90.
      pocz: 45 <= x <= 90
      @param x Kąt.
      @return Sinus kąta x.
   */
  private static int sin45to90(int x) {
    // W rzeczywistym programie metoda ta używałaby
    // aproksymacji wielomianowej zoptymalizowanej dla
    // stosowanego zakresu wartości wyjściowych.
```

```
        // Kod wyliczający sin(x) dla x w zakresie od 45 do 90 stopni.

    }
}
```

6. Opracuj plan testów i program testujący dla drugiego projektu programistycznego z rozdziału 1.
7. Uaktualnij plan testów i program testujący dla zmodyfikowanej wersji projektu kolekcji płyt DVD opisanej w czwartym projekcie programistycznym z rozdziału 1.
8. Wykonaj plan testów i program testujący dla piątego projektu programistycznego z rozdziału 1.
9. Wykonaj plan testów i program testujący dla szóstego projektu programistycznego z rozdziału 1.