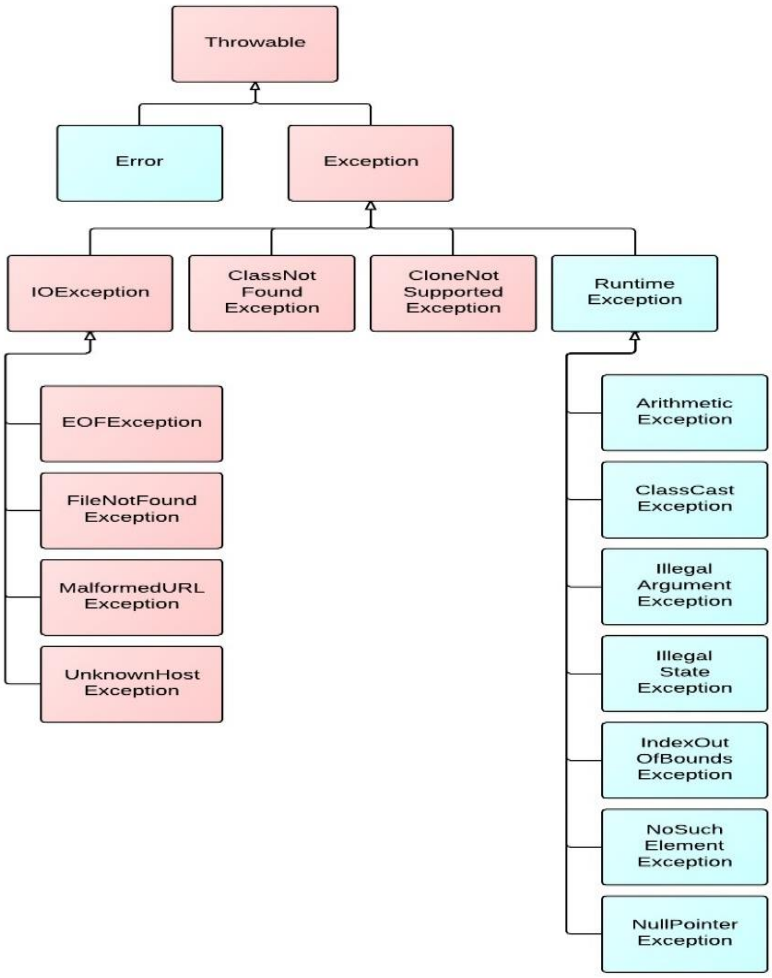
| Instrukcja laboratoryjna **2** | **Przetwarzanie / Systemy odporne na błędy** |
|---|---|
| | **Temat:** Wyjątki, testy jednostkowe, logowanie zdarzeń w systemie |
| | **Przygotował:** mgr inż. Michał Młodawski |

# 1. Wyjątki

Przez pojęcie **wyjątków** w językach obiektowych rozumiemy mechanizm kontroli przepływu służący do obsługi **zdarzeń wyjątkowych** *(w szczególności błędów)*. Wyjątek w języku Java to obiekt, który opisuje pewną sytuację błędną lub nieprawidłową – wyjątkową. Jest to obiekt odpowiedniego typu, tj. obiekt klasy **Throwable** lub jej dowolnej podklasy *(np. Exception, Error itd.)*.

Ważną częścią każdej aplikacji jest **sygnalizowanie** oraz obsługa pojawiających się błędów. Wystąpienie błędu w aplikacji zaimplementowanej w języku Java sygnalizujemy poprzez **rzucenie (wyrzucenie) wyjątku**. Obsługa błędów to tzw. **łapanie (przechwytywanie) wyjątków**.



Wyjątki dzielą się na dwa rodzaje:

- **kontrolowane** - to takie które musimy deklarować i obsługiwać,
- **niekontrolowane** - to takie które możemy obsługiwać, ale nie musimy.

Wyjątki **niekontrolowane** to instancje klas Error i **RuntimeException** oraz ich dowolnych podklas. Wyjątki **kontrolowane** to instancje klas **Throwable** i **Exception** oraz ich podklas, z wyłączeniem podklas klas **Error** i **RuntimeException**. Wyjątki mogą być rzucane (zgłaszane) przez **Wirtualną Maszynę Javy (JVM)** oraz przez programistę. Najpopularniejszym wyjątkiem rzucanym przez JVM jest **NullPointerException**. Wyjątek ten rzucany jest przy próbie odwołania się do obiektu (np. próba uruchomienia metody) z użyciem referencji, która ma wartość null. **NullPointerException** jest podklasą klasy **RuntimeException** tak więc jest wyjątkiem niekontrolowanym – nie musimy go w żaden sposób deklarować czy obsługiwać.

## 1.1.Obsługa wyjątków

Wyjątki są **wyrzucane** i **propagowane** tak długo jak długo pozostają nie przechwycone w kodzie programu. Wyjątki nie przechwycone propagują się aż do metody main(...) i jeśli także tam nie są przechwycone powodują przerwanie wykonania programu. W celu przechwycenia i obsłużenia wyjątków w języku Java korzysta się z bloku try-catch-finally.

Kod którego wyjątki chcemy przechwytywać umieszczamy w klauzuli try {...}. Wewnątrz klauzuli catch {...} umieszczamy **kod obsługi błędu** - kod ten będzie wykonany tylko i wyłącznie wówczas gdy kod programu wyrzuci wyjątek *typu* zgodnego z typem określonym w deklaracji klauzuli catch. Kod umieszczony wewnątrz klauzuli finally {...} będzie wykonany **zawsze**, niezależnie od tego czy kod programu wyrzucił wyjątek czy nie. Gdy w wyniku wykonania instrukcji w bloku try powstanie wyjątek typu Exception lub Throwable to sterowanie zostanie przekazane do kodu umieszczonego w w/w klauzulach catch. Klauzula finally jest opcjonalna, za to klauzul catch może być dowolnie wiele.

Przykład definicji bloku try-catch-finally:

```
try {
    //kod programu generujacy wyjatek
  } catch (Exception e) {
    //obsluga wyjatkow klasy Exception
  } catch (Throwable t) {
    //obsluga wyjatkow klasy Throwable
  } finally {
    //kod zawsze wykonywany
  }
```

Przykład użycia bloku try-catch-finally:

```
class TestExceptions {

    public static void main(String  args[]) {
       int d, a;
       try {
         d = 0;
         a = 42 / d;
         System.out.println("This will not be printed.");
       } catch (ArithmeticException e) {
         System.out.println("Division by zero.");
       } finally {
         System. out. println( "Finally") ;
       }
       System.out.println("Out of try/catch/finally - statement");
    }
 }
```

## 1.2.Generowanie (propagowanie) wyjątków

Wyjątki mogą być generowane **jawnie** w kodzie programu tj. przez programistę - z użyciem słowa kluczowego throw. Tak wygenerowany wyjątek może zostać obsłużony bezpośrednio przez klauzulę catch lub może zostać propagowany bezpośrednio wyżej do metody wywołującej nasz kod (naszą metodę) w której został wyrzucony wyjątek.

Przykład użycia instrukcji throw do generowania wyjątku:

```
if (o == null)
  throw new NullPointerException();
```

W istocie, instrukcja throw jest niczym innym jak sposobem specyficznego przekazywania sterowania do jakichś punktów programu (do miejsc obsługi wyjątku). Należy jednak korzystać z niej wyłącznie w celu **sygnalizowania błędów**.

Przykład ponownego wyrzucenia wyjątku:

```
try {
    ...
  } catch (Exception e) {
    ...
    throw e; //ponowne wyrzucenie wyjątku
  }
```

Przykład propagowania wyjątku:

```
// propagowanie wyjątku którego obsługą zajmie się kod wywołujący metodę
void method() throws IOException {

}
```

Przykład propagowania wyjątku:

```
class TestExceptions {

    static void myMethod(int testnum) throws Exception {
       System.out.println ("start - myMethod");
       if (testnum == 12)
          throw new Exception();
       System.out.println("end - myMethod");
       return;
    }

    public static void main(String  args[]) {
       int testnum = 12;
       try {
          System.out.println("try - first statement");
          myMethod(testnum);
          System.out.println("try - last statement");
       }
       catch ( Exception ex) {
          System.out.println("An Exception");
       }
       finally {
          System. out. println( "finally") ;
       }
       System.out.println("Out of try/catch/finally - statement");
    }
 }
```

## 1.3.Tworzenie własnych klas wyjątków

Oprócz wykorzystywania dostępnych klas wyjątków do generowania/obsługi sytuacji wyjątkowych programista ma możliwość tworzenia **własnych klas wyjątków**. Żeby stworzyć własny wyjątek należy zdefiniować odpowiednią klasę która dziedziczy funkcjonalność z klasy Exception.

Przykład definiowanie własnej klasy wyjątku

```
class SuperWyjatek extends Exception {
    ...
  }
```

Zwykle w naszej klasie wystarczy umieścić dwa konstruktory: **bezparametrowy** oraz z **jednym argumentem** typu String (komunikat o przyczynie powstania wyjątku). W konstruktorach tych należy wywołać **konstruktor** nadklasy (za pomocą odwołania super(...), w drugim przypadku z argumentem String).

Przykład użycia własnej klasy wyjątku:

```
class MyException extends Exception{
  public MyException() {
    System.out.println("Error");
  }
}

class TestExceptions {

   static void myMethod(int testnum) throws MyException {
      System.out.println ("start - myMethod");
      if (testnum == 12)
         throw new MyException();
      System.out.println("end - myMethod");
      return;
   }
   public static void main(String  args[]) {
      int testnum = 12;
      try {
         System.out.println("try - first statement");
         myMethod(testnum);
         System.out.println("try - last statement");
      }
      catch ( MyException ex) {
         System.out.println("An Exception");
      }
      finally {
         System. out. println( "finally") ;
      }
```

```
        System.out.println("Out of try/catch/finally - statement");
    }
}
```

## 2. Testowanie aplikacji

Testowanie oprogramowania jest kluczowym elementem procesu wytwarzania oprogramowania, mającym na celu zapewnienie jego jakości i zgodności z wymaganiami. Istnieje wiele rodzajów testów oprogramowania, które można zastosować w różnych etapach procesu. Oto niektóre z nich:

1. Testy jednostkowe (unit testing): Testy te skupiają się na pojedynczych modułach lub funkcjach w kodzie, sprawdzając ich poprawność i niezawodność.
2. Testy integracyjne (integration testing): Te testy mają na celu sprawdzenie współpracy między różnymi komponentami systemu, aby upewnić się, że razem działają poprawnie.
3. Testy funkcjonalne (functional testing): Testy te polegają na sprawdzeniu, czy oprogramowanie działa zgodnie z wymaganiami funkcjonalnymi i spełnia oczekiwania użytkowników.
4. Testy wydajności (performance testing): Testy te mają na celu ocenę wydajności systemu pod różnymi obciążeniami, takimi jak szybkość przetwarzania, czas odpowiedzi, zużycie zasobów i skalowalność.
5. Testy obciążeniowe (load testing): Sprawdzają, jak system radzi sobie z dużą ilością pracy, na przykład przy jednoczesnym korzystaniu przez wielu użytkowników.
6. Testy stresowe (stress testing): Te testy mają na celu sprawdzenie, jak system radzi sobie w ekstremalnych warunkach, takich jak bardzo wysokie obciążenie, awarie zasobów lub błędy w komponentach.
7. Testy użyteczności (usability testing): Testy te mają na celu ocenę, jak łatwo i intuicyjnie oprogramowanie jest w użyciu dla końcowego użytkownika.
8. Testy zgodności (compatibility testing): Te testy sprawdzają, czy oprogramowanie działa poprawnie na różnych platformach sprzętowych, systemach operacyjnych i przeglądarkach internetowych.
9. Testy bezpieczeństwa (security testing): Testy te mają na celu identyfikację luk i potencjalnych zagrożeń dla bezpieczeństwa oprogramowania, takich jak wycieki danych, podatności na ataki czy błędy w zarządzaniu uprawnieniami.
10. Testy regresji (regression testing): Testy te polegają na ponownym przetestowaniu oprogramowania po wprowadzeniu zmian, aby upewnić się, że nie wprowadzono nowych błędów i że poprawione elementy nadal działają poprawnie.
11. Testy akceptacji (acceptance testing): Ostateczne testy przeprowadzane przez klienta lub użytkownika końcowego, mające na celu sprawdzenie, czy oprogramowanie spełnia ich wymagania i jest gotowe do wdrożenia.

## 3. Testy jednostkowe

Testy jednostkowe są kluczowym elementem testowania oprogramowania i mają na celu sprawdzenie pojedynczych funkcji, klas lub modułów w kodzie. W Javie i Spring Boot, testy jednostkowe można przeprowadzić za pomocą różnych narzędzi i bibliotek, takich jak JUnit, Mockito i AssertJ.

Oto krótki przegląd, jak przeprowadzić testy jednostkowe w Javie i Spring Boot:

1. JUnit: JUnit to popularna biblioteka do testowania jednostkowego dla Javy. W Spring Boot, JUnit 5 jest domyślnie wspierany, ale można także korzystać z wcześniejszych wersji JUnit.

Aby utworzyć test jednostkowy z JUnit, należy utworzyć klasę testową, która będzie zawierać metody testowe. Każda metoda testowa powinna być oznaczona adnotacją **@Test** i skupiać się na jednym konkretnym scenariuszu testowania. Przykład:

```
package com.example.demo.lab5;
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.assertEquals;

class MyMathTest {
    @Test
    void testAddition() {
        MyMath myMath = new MyMath();
        int result = myMath.add(5, 3);
        assertEquals(8, result);
    }
}
```

1. Mockito: Mockito to biblioteka służąca do tworzenia mocków (sztucznych obiektów) dla zależności, które można wykorzystać w testach jednostkowych. Mockowanie jest szczególnie przydatne w przypadku testowania komponentów, które są zależne od zewnętrznych usług czy zasobów.

Aby użyć Mockito w Spring Boot, można dodać zależność do pliku **pom.xml** lub **build.gradle**. Przykład użycia Mockito:

```
package com.example.demo.lab5;

import org.apache.catalina.User;
import org.junit.jupiter.api.Test;
import org.mockito.Mockito;

import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.mockito.Mockito.*;

class UserServiceTest {
    @Test
    void testFindUser() {
        UserRepository userRepository = Mockito.mock(UserRepository.class);
        when(userRepository.findUserById(1)).thenReturn(new User(1, "John
```

```
Doe"));

        UserService userService = new UserService(userRepository);
        User user = (User) userService.findUserById(1);

        assertEquals("John Doe", user.getName());
        verify(userRepository, times(1)).findUserById(1);
    }
}
```

Podsumowując, testy jednostkowe w Javie i Spring Boot można przeprowadzić za pomocą różnych narzędzi i bibliotek, takich jak JUnit, Mockito i AssertJ. Wspólnie te narzędzia pozwalają na tworzenie solidnych, czytelnych i łatwo utrzymywanych testów jednostkowych, które pomagają w utrzymaniu wysokiej jakości kodu. Ważne jest, aby stosować dobre praktyki testowania, takie jak testowanie jednego scenariusza w każdym teście, używanie mocków do izolowania zależności oraz stosowanie asercji w celu zweryfikowania oczekiwanych wyników.

## 4. Testy jednostkowe z wykorzystaniem JUnit

JUnit to jeden z najpopularniejszych frameworków do testów jednostkowych w ekosystemie Java. Wersja JUnit 5 zawiera szereg innowacji, których **celem jest wspieranie nowych funkcji w Javie 8 i nowszych** , a także umożliwienie wielu różnych stylów testowania.

## 4.1.Dodanie JUnit do projektu

Konfiguracja JUnit 5.x.0 jest dość prosta; wystarczy dodać następującą zależność do naszego *pom.xml* :

```
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-engine</artifactId>
    <version>5.8.1</version>
    <scope>test</scope>
</dependency>
```

Co więcej, dostępne jest są bezpośrednie wsparcie dla uruchamiania testów jednostkowych na platformie JUnit w Eclipse, a także IntelliJ. Możemy oczywiście przeprowadzać również testy z wykorzystaniem scope Maven Test.

JUnit 5 składa się z kilku różnych modułów z trzech różnych podprojektów.

Podstawowy moduł zawiera modele programowania i rozszerzeń do pisania testów w JUnit 5, podstawowe adnotacje to::

1. @Test: Oznacza metodę jako test jednostkowy. Test Runner uruchamia metody oznaczone tą adnotacją jako testy.

2. @BeforeEach: Oznacza metodę, która zostanie wykonana przed każdym testem w klasie testowej. Może być używana do przygotowania danych testowych lub inicjalizacji zasobów.

3. @AfterEach: Oznacza metodę, która zostanie wykonana po każdym teście w klasie testowej. Może być używana do czyszczenia danych testowych lub zwalniania zasobów.

4. @BeforeAll: Oznacza metodę, która zostanie wykonana raz przed wszystkimi testami w klasie testowej. Może być używana do przygotowania danych testowych lub inicjalizacji zasobów, które są wspólne dla wszystkich testów.

5. @AfterAll: Oznacza metodę, która zostanie wykonana raz po wszystkich testach w klasie testowej. Może być używana do czyszczenia danych testowych lub zwalniania zasobów, które są wspólne dla wszystkich testów.

6. @Disabled: Oznacza test lub klasę testową, który ma być pominięty podczas uruchamiania testów. Może być używane do tymczasowego wyłączenia testów, które są w trakcie naprawy lub rozwoju.

7. @DisplayName: Pozwala na ustawienie opisowego tekstu dla testu lub klasy testowej, który będzie wyświetlany w wynikach testów zamiast nazwy metody lub klasy.

8. @Nested: Oznacza klasę wewnętrzną jako klasę testową zagnieżdżoną w innej klasie testowej. Pozwala na tworzenie grup testów związanych z tą samą funkcją lub funkcjonalnością.

9. @RepeatedTest: Pozwala na wielokrotne wykonywanie tego samego testu. Przyjmuje liczbę powtórzeń jako argument.

10. @ParameterizedTest: Pozwala na przeprowadzenie testu z różnymi zestawami danych wejściowych. Współpracuje z innymi adnotacjami, takimi jak @ValueSource, @EnumSource, @CsvSource itp., które definiują wartości wejściowe.

11. @ValueSource: Adnotacja używana z @ParameterizedTest, która dostarcza zestaw prostych wartości wejściowych, takich jak liczby, łańcuchy znaków czy wartości logiczne.

12. @EnumSource: Adnotacja używana z @ParameterizedTest do przeprowadzenia testów z wartościami wyliczeniowymi (enum).

13. @CsvSource: Adnotacja używana z @ParameterizedTest do dostarczania zestawów danych wejściowych zdefiniowanych jako wartości CSV.

14. @CsvFileSource: Adnotacja używana z @ParameterizedTest do dostarczania zestawów danych wejściowych z pliku CSV.

15. @Timeout: Adnotacja, która pozwala na ustawienie limitu czasowego dla testu. Jeśli test przekroczy ten limit, zostanie uznany za nieudany.

16. @ExtendWith: Pozwala na rozszerzenie funkcjonalności testów jednostkowych za pomocą klas implementujących interfejs Extension. Może być używana na poziomie klasy testowej lub na poziomie metody testowej.

## 4.2.Asercje w JUnit

W JUnit 5 dostępna jest klasa **org.junit.jupiter.api.Assertions**, która zawiera wiele metod asercji, które można użyć do sprawdzania różnych warunków w testach jednostkowych. Oto niektóre z nich:

1. assertEquals(expected, actual) - sprawdza, czy dwie wartości są równe.
2. assertNotEquals(unexpected, actual) - sprawdza, czy dwie wartości są różne.
3. assertTrue(condition) - sprawdza, czy podany warunek jest prawdziwy.
4. assertFalse(condition) - sprawdza, czy podany warunek jest fałszywy.
5. assertNull(value) - sprawdza, czy wartość jest null.
6. assertNotNull(value) - sprawdza, czy wartość nie jest null.
7. assertArrayEquals(expectedArray, actualArray) - sprawdza, czy dwie tablice są równe pod względem zawartości.
8. assertIterableEquals(expectedIterable, actualIterable) - sprawdza, czy dwie kolekcje są równe pod względem zawartości.
9. assertLinesMatch(expectedLines, actualLines) - sprawdza, czy listy ciągów znaków są równe lub mają te same wzorce.
10. assertThrows(expectedType, executable) - sprawdza, czy podany kod rzuca wyjątek oczekiwanego typu.
11. assertTimeout(duration, executable) - sprawdza, czy wykonanie podanego kodu nie przekracza określonego czasu.
12. assertTimeoutPreemptively(duration, executable) - sprawdza, czy wykonanie podanego kodu nie przekracza określonego czasu, ale przerywa wykonanie po przekroczeniu limitu czasowego.
13. assertAll(executable...) - sprawdza wiele warunków naraz i zgłasza wszystkie niepowodzenia (a nie tylko pierwsze napotkane). Używane głównie dla grupowania asercji.

Przykład użycia różnych asercji w JUnit 5:

```java
package com.example.demo.lab5;

import org.junit.jupiter.api.Test;

import java.time.Duration;

import static org.junit.jupiter.api.Assertions.*;

class ExampleTest {
    @Test
    void testAssertions() {
        int actual = 2 + 2;
        int expected = 4;

        assertEquals(expected, actual, "2 + 2 should equal 4");
        assertNotEquals(5, actual, "2 + 2 should not equal 5");
        assertTrue(actual == 4, "2 + 2 should be true");
        assertFalse(actual == 5, "2 + 2 should be false");
        assertNull(null, "Should be null");
        assertNotNull(actual, "Should not be null");

        int[] array1 = {1, 2, 3};
        int[] array2 = {1, 2, 3};
        assertArrayEquals(array1, array2, "Arrays should be equal");

        assertThrows(IllegalArgumentException.class, () -> {
            throw new IllegalArgumentException("Example exception");
        }, "Should throw an IllegalArgumentException");

        assertTimeout(Duration.ofMillis(100), () -> {
            Thread.sleep(50);
        }, "Should not exceed timeout of 100 milliseconds");
    }
}
```

Te metody asercji dostarczane przez JUnit 5 pozwalają na sprawdzanie różnych warunków w testach jednostkowych, tak aby można było szybko i dokładnie upewnić się, że testowany kod działa poprawnie. Asercje umożliwiają stwierdzenie, czy wartości są równe, różne, prawdziwe, fałszywe, czy też sprawdzenie, czy dane wyjątki są rzucane podczas wykonywania kodu. Dzięki temu, JUnit 5 ułatwia tworzenie i utrzymanie testów jednostkowych dla różnorodnych scenariuszy.

## 4.3.Adnotacje w JUnit

1. @Test - oznacza metodę jako test jednostkowy:

```
package com.example.demo.lab5;

import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.assertEquals;

class ExampleTest {
    @Test
    void testAddition() {
        int a = 2;
        int b = 3;
        int sum = a + b;
        assertEquals(5, sum);
    }
}
```

2. @BeforeEach - oznacza metodę, która zostanie wykonana przed każdym testem:

```
package com.example.demo.lab5;

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

import java.util.ArrayList;
import java.util.List;

import static org.junit.jupiter.api.Assertions.assertEquals;

class ExampleTest {
    private List<String> list;

    @BeforeEach
    void setUp() {
        list = new ArrayList<>();
    }

    @Test
    void testList() {
        list.add("example");
        assertEquals(1, list.size());
    }
}
```

3. @AfterEach - oznacza metodę, która zostanie wykonana po każdym teście:

```
import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

import java.util.ArrayList;
import java.util.List;

import static org.junit.jupiter.api.Assertions.assertEquals;

class ExampleTest {
    private List<String> list;

    @BeforeEach
    void setUp() {
        list = new ArrayList<>();
    }

    @AfterEach
    void tearDown() {
        list.clear();
    }

    @Test
    void testList() {
        list.add("example");
        assertEquals(1, list.size());
    }
}
```

4. @BeforeAll - oznacza metodę, która zostanie wykonana raz przed wszystkimi testami:

```
import org.junit.jupiter.api.BeforeAll;
import org.junit.jupiter.api.Test;

import java.util.ArrayList;
import java.util.List;

import static org.junit.jupiter.api.Assertions.assertTrue;

class ExampleTest {
    private static List<String> list;

    @BeforeAll
    static void setUpBeforeAll() {
        list = new ArrayList<>();
        list.add("example");
    }

    @Test
    void testList() {
        assertTrue(list.contains("example"));
```

```
    }
}
```

5. @AfterAll - oznacza metodę, która zostanie wykonana raz po wszystkich testach:

```
import org.junit.jupiter.api.AfterAll;
import org.junit.jupiter.api.BeforeAll;
import org.junit.jupiter.api.Test;

import java.util.ArrayList;
import java.util.List;

import static org.junit.jupiter.api.Assertions.assertTrue;


class ExampleTest {
    private static List<String> list;

    @BeforeAll
    static void setUpBeforeAll() {
        list = new ArrayList<>();
        list.add("example");
    }

    @AfterAll
    static void tearDownAfterAll() {
        list.clear();
    }

    @Test
    void testList() {
        assertTrue(list.contains("example"));
    }
}
```

6. @Disabled - oznacza test, który ma być pominięty podczas uruchamiania testów:

```
import org.junit.jupiter.api.Disabled;
import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.assertEquals;

class ExampleTest {
    @Test
    void enabledTest() {
        int a = 2;
        int b = 3;
        int sum = a + b;
        assertEquals(5, sum);
    }

    @Disabled
    @Test
    void disabledTest() {
        int a = 2;
        int b = 3;
        int product = a * b;
        assertEquals(6, product);
    }
}
```

7. @DisplayName - pozwala na ustawienie opisowego tekstu dla testu:

```
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.assertEquals;

class ExampleTest {
    @DisplayName("Test Addition")
    @Test
    void testAddition() {
        int a = 2;
        int b = 3;
        int sum = a + b;
        assertEquals(5, sum);
    }
}
```

8. @Nested - oznacza klasę wewnętrzną jako klasę testową zagnieżdżoną w innej klasie testowej:

```
import org.junit.jupiter.api.Nested;
import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.assertEquals;

class ExampleTest {
    @Nested
    class NestedTestClass {
        @Test
        void nestedTestMethod() {
            int a = 2;
            int b = 3;
            int sum = a + b;
            assertEquals(5, sum);
        }
    }
}
```

9. @RepeatedTest - pozwala na wielokrotne wykonywanie tego samego testu:

```
import org.junit.jupiter.api.RepeatedTest;

import static org.junit.jupiter.api.Assertions.assertEquals;

class ExampleTest {
    @RepeatedTest(3)
    void repeatedTest() {
        int a = 2;
        int b = 3;
        int sum = a + b;
        assertEquals(5, sum);
    }
}
```

10. @ParameterizedTest - pozwala na przeprowadzenie testu z różnymi zestawami danych wejściowych:

```
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.ValueSource;

import static org.junit.jupiter.api.Assertions.assertTrue;

class ExampleTest {
    @ParameterizedTest
    @ValueSource(ints = {1, 2, 3})
    void parameterizedTest(int input) {
        assertTrue(input > 0);
    }
}
```

11. @Timeout - adnotacja, która pozwala na ustawienie limitu czasowego dla testu:

```
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.Timeout;

import static org.junit.jupiter.api.Assertions.assertEquals;

class ExampleTest {
    @Timeout(5)
    @Test
    void timeoutTest() {
        int a = 2;
        int b = 3;
        int sum = a + b;
        assertEquals(5, sum);
    }
}
```

12. @EnumSource - używana z @ParameterizedTest do przeprowadzenia testów z wartościami wyliczeniowymi (enum):

```
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.EnumSource;

import static org.junit.jupiter.api.Assertions.assertNotNull;

enum MyEnum {
    FIRST, SECOND, THIRD
}

class ExampleTest {
    @ParameterizedTest
    @EnumSource(MyEnum.class)
    void enumSourceTest(MyEnum input) {
        assertNotNull(input);
    }
}
```

13. @CsvSource - używana z @ParameterizedTest do dostarczania zestawów danych wejściowych zdefiniowanych jako wartości CSV:

```
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.CsvSource;

import static org.junit.jupiter.api.Assertions.assertEquals;

class ExampleTest {
    @ParameterizedTest
    @CsvSource({"1, 3, 4", "2, 3, 5", "3, 4, 7"})
    void csvSourceTest(int a, int b, int sum) {
        assertEquals(sum, a + b);
    }
}
```

14. @CsvFileSource - używana z @ParameterizedTest do dostarczania zestawów danych wejściowych z pliku CSV:

Przykład pliku CSV (**src/test/resources/test-data.csv**):

```
1, 3, 4

2, 3, 5

3, 4, 7
```

```java
package com.example.demo.lab5;

import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.CsvFileSource;

import static org.junit.jupiter.api.Assertions.assertEquals;

class ExampleTest {
    @ParameterizedTest
    @CsvFileSource(resources = "/test-data.csv", numLinesToSkip = 0)
    void csvFileSourceTest(int a, int b, int sum) {
        assertEquals(sum, a + b);
    }
}
```

15. @MethodSource - używana z @ParameterizedTest do dostarczania zestawów danych wejściowych z metody statycznej:

```java
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.Arguments;
import org.junit.jupiter.params.provider.MethodSource;
import java.util.stream.Stream;

import static org.junit.jupiter.api.Assertions.assertEquals;

class ExampleTest {
    static Stream<Arguments> provideTestArguments() {
        return Stream.of(
                Arguments.of(1, 3, 4),
                Arguments.of(2, 3, 5),
                Arguments.of(3, 4, 7)
        );
    }

    @ParameterizedTest
    @MethodSource("provideTestArguments")
    void methodSourceTest(int a, int b, int sum) {
        assertEquals(sum, a + b);
    }
}
```

16. @Tag - pozwala na dodanie etykiet do testów, co ułatwia ich filtrowanie i grupowanie:

```
package com.example.demo.lab5;

import org.junit.jupiter.api.Tag;
import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.assertEquals;

class ExampleTest {
    @Tag("fast")
    @Test
    void fastTest() {
        int a = 2;
        int b = 3;
        int sum = a + b;
        assertEquals(5, sum);
    }

    @Tag("slow")
    @Test
    void slowTest() {
        int a = 2;
        int b = 3;
        int product = a * b;
        assertEquals(6, product);
    }
}
```

17. @TestInstance - umożliwia zmianę cyklu życia instancji testów, czyli określa, jak często tworzona jest nowa instancja klasy testowej:

```
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.TestInstance;

import static org.junit.jupiter.api.Assertions.assertEquals;

@TestInstance(TestInstance.Lifecycle.PER_CLASS)
class ExampleTest {
    private int counter = 0;

    @Test
    void firstTest() {
        counter++;
        assertEquals(1, counter);
    }

    @Test
    void secondTest() {
        counter++;
        assertEquals(2, counter);
    }
}
```

18. @TestFactory - używana do tworzenia kolekcji dynamicznych testów:

```
import org.junit.jupiter.api.DynamicTest;
import org.junit.jupiter.api.TestFactory;

import java.util.Arrays;
import java.util.Collection;

import static org.junit.jupiter.api.Assertions.assertEquals;

class ExampleTest {
    @TestFactory
    Collection<DynamicTest> dynamicTests() {
        return Arrays.asList(
                DynamicTest.dynamicTest("Test 1", () -> assertEquals(4, 2 *
2)),
                DynamicTest.dynamicTest("Test 2", () -> assertEquals(9, 3 *
3))
        );
    }
}
```

19. @RegisterExtension - pozwala na rejestrowanie rozszerzeń JUnit w sposób deklaratywny:

```
import com.example.demo.lab5.MyExtension;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.RegisterExtension;

class ExampleTest {
    @RegisterExtension
    static MyExtension myExtension = new MyExtension();

    @Test
    void exampleTest() {
        //...
    }
}
```

20. @Order - określa kolejność wykonywania testów:

```
import org.junit.jupiter.api.Order;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.TestMethodOrder;
import org.junit.jupiter.api.MethodOrderer.OrderAnnotation;

@TestMethodOrder(OrderAnnotation.class)
class ExampleTest {
    @Test
    @Order(2)
    void secondTest() {
        //...
    }

    @Test
    @Order(1)
    void firstTest() {
        //...
    }
}
```

## 5.Logowanie zdarzeń w systemie

Logowanie danych/błędów (ang. logging) to proces zbierania, przechowywania i analizowania informacji o działaniach systemu lub aplikacji. Logi zawierają różnego rodzaju dane, takie jak informacje o błędach, wyjątkach, zdarzeniach, akcjach użytkowników itp.

Logowanie danych jest wykorzystywane w celu monitorowania działania systemu lub aplikacji oraz pomocy w identyfikacji i rozwiązywaniu problemów. Dzięki logowaniu można łatwiej znaleźć przyczynę awarii lub błędów, co pozwala na szybsze ich naprawienie. Ponadto, logi pozwalają na analizę działania systemu lub aplikacji, co może pomóc w optymalizacji działania i poprawie wydajności.

Zalety logowania danych/błędów:

- Ułatwienie identyfikacji i rozwiązywania problemów
- Pomoc w analizie działania systemu lub aplikacji
- Możliwość optymalizacji działania i poprawy wydajności
- Zwiększenie bezpieczeństwa systemu lub aplikacji poprzez monitorowanie zdarzeń

Wady logowania danych/błędów:

- Wymaga dodatkowego nakładu pracy i zasobów w celu zbierania i przechowywania logów
- Duża ilość logów może zajmować dużo miejsca na dysku
- Czasami trudno jest zinterpretować logi i znaleźć przyczynę błędów lub awarii

Logowanie danych/błędów jest nieodłącznym elementem procesu wytwarzania oprogramowania, a także jego eksploatacji. W ciągu działania systemów i aplikacji mogą wystąpić różnego rodzaju problemy, takie jak awarie, błędy, nieoczekiwane zdarzenia czy problemy z wydajnością. Dlatego tak ważne jest zbieranie i przechowywanie logów, które zawierają informacje o takich sytuacjach.

Logowanie danych/błędów umożliwia identyfikację i rozwiązanie problemów, które występują w systemie lub aplikacji. Dzięki logom można łatwiej zlokalizować miejsce wystąpienia błędu lub awarii, co przyspiesza proces naprawy. Logowanie pozwala również na śledzenie akcji użytkowników, co jest przydatne w celu poprawy interfejsu użytkownika.

Kolejną zaletą logowania danych/błędów jest możliwość analizy działania systemu lub aplikacji. Logi zawierają informacje o działaniach systemu lub aplikacji, takie jak czas odpowiedzi, ilość zapytań, akcje użytkowników itp. Te informacje mogą być wykorzystane do optymalizacji działania systemu lub aplikacji oraz poprawy wydajności.

Logowanie danych/błędów ma również znaczenie w kontekście bezpieczeństwa systemu lub aplikacji. Logi pozwalają na monitorowanie zdarzeń i wykrywanie podejrzanych aktywności. Dzięki temu można szybciej zareagować na ewentualne zagrożenia i zminimalizować ich skutki.

Jednak logowanie danych/błędów ma również pewne wady. Wymaga dodatkowego nakładu pracy i zasobów w celu zbierania i przechowywania logów. Ponadto, duże ilości logów mogą zajmować dużo miejsca na dysku. Czasami trudno jest zinterpretować logi i znaleźć przyczynę błędów lub awarii.

Warto zwrócić uwagę, że logowanie danych/błędów jest kluczowym elementem w procesie rozwoju oprogramowania, wdrażania go oraz jego eksploatacji. Dlatego warto zainwestować w narzędzia umożliwiające zbieranie i analizowanie logów, co pozwoli na skuteczne zarządzanie systemami i aplikacjami, a także zminimalizowanie kosztów i czasu potrzebnego na rozwiązywanie problemów.

Podsumowując, logowanie danych/błędów jest ważnym narzędziem w monitorowaniu i analizie działania systemów lub aplikacji. Mimo że logowanie wymaga dodatkowego nakładu pracy i zasobów, to jego zalety zdecydowanie przeważają nad wadami.

W Javie oraz Spring Boot istnieje wiele narzędzi umożliwiających logowanie informacji. Oto kilka sposobów, jak można logować informacje przy użyciu Javy oraz Spring Boot:

SLF4J (Simple Logging Facade for Java) to framework logowania dla aplikacji napisanych w języku Java. Jego celem jest zapewnienie prostego, intuicyjnego API logowania dla programistów, którzy korzystają z różnych implementacji logowania, takich jak Log4j, java.util.logging czy Logback.

SLF4J działa jako warstwa pośrednicząca pomiędzy aplikacją, która loguje informacje, a konkretnej implementacji logowania. Dzięki temu, jeśli chcemy zmienić implementację logowania, wystarczy zmienić tylko odpowiednią konfigurację w pliku konfiguracyjnym, bez potrzeby modyfikacji kodu źródłowego aplikacji.

SLF4J oferuje prosty i intuicyjny interfejs programistyczny, który pozwala na logowanie informacji w różnym stopniu ważności, takim jak DEBUG, INFO, WARN, ERROR, a także na podstawie różnych kategorii lub nazw loggerów. Przykładowo, logowanie informacji o błędzie może wyglądać tak:

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class MyClass {
    private static final Logger logger =
LoggerFactory.getLogger(MyClass.class);

    public void doSomething() {
        try {
            // some code that can throw an exception
        } catch (Exception e) {
            logger.error("An error occurred while doing something", e);
        }
    }
}
```

W powyższym przykładzie logujemy informacje o wystąpieniu błędu za pomocą metody **error()** na obiekcie loggera, który został utworzony za pomocą LoggerFactory. Informacje o błędzie oraz wyjątku są przekazywane jako parametry metody.

SLF4J jest bardzo popularnym frameworkiem logowania w Javie, ze względu na swoją prostotę i elastyczność. Dzięki niemu programiści mogą skupić się na logowaniu informacji, bez potrzeby poznawania szczegółów konkretnej implementacji logowania.

Log4j2 jest popularnym narzędziem do logowania dla aplikacji Java. W przypadku aplikacji opartych na Spring Boot, konfiguracja Log4j2 może być wykonana za pomocą pliku konfiguracyjnego XML lub programowo. Poniżej znajduje się przykładowa konfiguracja Log4j2 w aplikacji Spring Boot, wykorzystując plik konfiguracyjny XML:

1. Dodaj zależności Log4j2 do pliku **pom.xml** (dla projektu Maven):

```
<dependencies>
    <!-- SLF4J API -->
    <dependency>
        <groupId>org.slf4j</groupId>
        <artifactId>slf4j-api</artifactId>
        <version>2.0.16</version>
    </dependency>

    <!-- Logback Classic (implementacja dla SLF4J) -->
    <dependency>
        <groupId>ch.qos.logback</groupId>
        <artifactId>logback-classic</artifactId>
        <version>1.5.8</version>
    </dependency>
</dependencies>
```

```
<!-- Dodaj zależności dla Log4j2 w przypadku używania springa -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-log4j2</artifactId>
</dependency>
```

**spring-boot-starter-log4j2** to zestaw startowy dla Log4j2, który zawiera wszystkie potrzebne zależności.

2. Utwórz plik konfiguracyjny Log4j2:

W katalogu **src/main/resources** utwórz plik konfiguracyjny dla Log4j2, jak **log4j2.xml**, **log4j2.yaml** lub **log4j2.json**, w zależności od preferowanego formatu. Poniżej przedstawiono przykładową konfigurację w formacie XML:

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="WARN">
    <Appenders>
        <Console name="Console" target="SYSTEM_OUT">
            <PatternLayout pattern="%d{yyyy-MM-dd HH:mm:ss.SSS} [%t] %-
5level %logger{36} - %msg%n"/>
        </Console>
    </Appenders>
    <Loggers>
        <Logger name="com.example" level="info" additivity="false">
            <AppenderRef ref="Console"/>
        </Logger>
        <Root level="error">
            <AppenderRef ref="Console"/>
        </Root>
    </Loggers>
</Configuration>
```

W powyższym przykładzie zdefiniowano appender **Console** oraz dwa loggery: jeden dla pakietu **com.example** z poziomem logowania **INFO** i drugi dla loggera Root z poziomem logowania **ERROR**.

3. Uruchom aplikację:

Po dodaniu zależności do pliku **pom.xml** i utworzeniu pliku konfiguracyjnego Log4j2, Spring Boot automatycznie skonfiguruje Log4j2 jako system logowania dla Twojej aplikacji. Teraz możesz uruchomić swoją aplikację Spring Boot i zauważyć, że logowanie jest realizowane za pomocą Log4j2 zgodnie z utworzoną konfiguracją.

4. Używaj Log4j2 w swoim kodzie:

Aby korzystać z Log4j2 w swoim kodzie, po prostu dodaj logger do swoich klas:

```
import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;
public class MyClass {
    private static final Logger logger =
LogManager.getLogger(MyClass.class);

    public void exampleMethod() {
        logger.trace("To jest log na poziomie TRACE");
        logger.debug("To jest log na poziomie DEBUG");
        logger.info("To jest log na poziomie INFO");
        logger.warn("To jest log na poziomie WARN");
        logger.error("To jest log na poziomie ERROR");
        logger.fatal("To jest log na poziomie FATAL");
    }
}
```

W Log4j2, element **<Configuration>** jest głównym elementem w pliku konfiguracyjnym i może zawierać następujące atrybuty:

1. **status**: (opcjonalne) Określa poziom logowania dla Log4j2, który ma być używany podczas inicjalizacji i diagnozowania problemów z konfiguracją. Przyjmuje wartości: TRACE, DEBUG, INFO, WARN, ERROR, FATAL i OFF. Domyślnie ustawione na "WARN".
2. **monitorInterval**: (opcjonalne) Czas w sekundach, co jaki Log4j2 będzie sprawdzać, czy plik konfiguracyjny uległ zmianie. Jeśli plik zostanie zmieniony, konfiguracja zostanie automatycznie zaktualizowana. Wartość musi być liczbą całkowitą większą od 0. Domyślnie funkcja ta jest wyłączona.
3. **packages**: (opcjonalne) Lista pakietów oddzielonych przecinkami, które zawierają niestandardowe pluginy Log4j2. W przypadku korzystania z niestandardowych komponentów, takich jak appender'y, filtr'y lub lookup'y, należy dodać ich pakiety do tego atrybutu.
4. **name**: (opcjonalne) Unikatowa nazwa dla konfiguracji. Może być użyte, jeśli chcesz zdefiniować

Appenders w Log4j2 są odpowiedzialne za dostarczanie logów do różnych celów wyjściowych, takich jak konsola, pliki, bazy danych, systemy zdalne czy usługi e-mail. Dzięki appenderom logi mogą być zapisywane i przechowywane w sposób dostosowany do potrzeb aplikacji.

W konfiguracji Log4j2 możemy zdefiniować jeden lub więcej appenderów i przypisać je do różnych loggerów. Przykłady typowych appenderów to:

1. ConsoleAppender: Wypisuje logi na standardowe wyjście (np. konsolę). Jest to użyteczne podczas debugowania aplikacji w trakcie jej tworzenia.

2. FileAppender: Zapisuje logi do pliku na dysku. Może być używany do przechowywania logów na stałe lub przeglądania ich po fakcie.
3. RollingFileAppender: Podobnie jak FileAppender, zapisuje logi do pliku, ale z dodatkową funkcją rotacji plików. Pozwala to na kontrolowanie rozmiaru plików z logami, tworzenie kopii zapasowych i utrzymanie określonej liczby archiwów.
4. SocketAppender: Wysyła logi do serwera zdalnego przez gniazdo sieciowe. Może być używane do agregowania logów z wielu aplikacji w jednym miejscu.
5. SMTPAppender: Wysyła logi przez e-mail. Jest to przydatne Jest to przydatne w przypadku powiadamiania administratorów lub deweloperów o krytycznych błędach lub ważnych zdarzeniach.
6. DatabaseAppender: Zapisuje logi do bazy danych. Może być używane do przechowywania logów w strukturze danych, co ułatwia ich analizę, wyszukiwanie i generowanie raportów.

Aby używać appenderów w konfiguracji Log4j2, należy je zdefiniować w sekcji **<Appenders>** pliku konfiguracyjnego, a następnie przypisać do odpowiednich loggerów w sekcji **<Loggers>**.

W Log4j2 logger Root (nazywany również loggerem głównym) odgrywa kluczową rolę jako ostateczny punkt przechwytywania logów. Wszystkie logi, które nie są przechwytywane przez żadnego z zdefiniowanych loggerów, są przekazywane do loggera Root.

Logger Root ma kilka istotnych cech:

1. Jest to logger domyślny, który zawsze istnieje w hierarchii loggerów, niezależnie od tego, czy jest zdefiniowany w konfiguracji, czy nie.
2. Otrzymuje logi od wszystkich loggerów, które mają ustawioną flagę **additivity** na **true** (domyślna wartość), a także logi, których poziom nie jest przechwytywany przez inne zdefiniowane logger'y.
3. Działa jako "ostatnia linia obrony" w przechwytywaniu logów, co pomaga uniknąć sytuacji, gdy logi są tracone z powodu braku odpowiedniego logger'a.

Stosowanie loggera Root ma na celu zapewnienie, że żadne logi nie zostaną utracone i że wszystkie logi zostaną przekazane do przynajmniej jednego appender'a. Możemy ustawić poziom logowania i appender'y dla loggera Root, aby kontrolować, jakie logi będą przekazywane dalej.

W praktyce logger Root jest często używany do przechwytywania logów o wyższych poziomach, takich jak ERROR lub WARN, co pozwala na szybkie wykrywanie problemów we wszystkich komponentach aplikacji.

W konfiguracji Log4j2, logger może być zdefiniowany z następującymi parametrami i argumentami:

1. **name**: (wymagane) Unikatowa nazwa loggera, zwykle reprezentująca pakiet lub klasę, dla której ma być używany logger. Na przykład: **name="com.example"**.
2. **level**: (opcjonalne) Poziom logowania, który określa, jakie logi będą przechwytywane przez logger. Przyjmuje wartości TRACE, DEBUG, INFO, WARN, ERROR i FATAL. Jeśli nie jest określony, logger dziedziczy poziom logowania po swoim przodku.
3. **additivity**: (opcjonalne) Flagę określającą, czy logi mają być przekazywane do loggerów wyższego rzędu (rodziców) w hierarchii. Domyślnie ustawiona na **true**. Jeśli ustawiona na **false**, logi są przekazywane tylko do zdefiniowanych appenderów dla tego loggera i nie są przekazywane dalej.
4. **AppenderRef**: (opcjonalne) Wskazuje, do których appenderów logi mają być przekazywane przez logger. Można dodać jeden lub więcej appenderów, które zostały zdefiniowane w sekcji **<Appenders>**.

PatternLayout w Log4j2 pozwala na tworzenie niestandardowych wzorców formatowania logów, które można dostosować do indywidualnych potrzeb. Oto niektóre z dostępnych konwerterów wzorców (ang. pattern converters), które można wykorzystać w PatternLayout:

1. **%d** - Data i czas: Wyświetla datę i godzinę w określonym formacie. Można dostosować format, podając go w nawiasach klamrowych, np. **%d{yyyy-MM-dd HH:mm:ss.SSS}**.
2. **%msg** - Wiadomość: Wyświetla wiadomość związana z logowaniem.
3. **%level** - Poziom: Wyświetla poziom logowania (np. ERROR, WARN, INFO, DEBUG, TRACE).
4. **%logger** - Logger: Wyświetla nazwę loggera, który wygenerował wpis. Można ograniczyć długość nazwy loggera, dodając liczbę w nawiasach klamrowych, np. **%logger{36}**.
5. **%t** - Wątek: Wyświetla nazwę bieżącego wątku.
6. **%F** - Plik: Wyświetla nazwę pliku źródłowego, z którego pochodzi wpis.
7. **%L** - Numer linii: Wyświetla numer linii w pliku źródłowym, z którego pochodzi wpis.
8. **%M** - Metoda: Wyświetla nazwę metody, w której został wygenerowany wpis.
9. **%C** - Klasa: Wyświetla nazwę klasy, z której pochodzi wpis. Można ograniczyć długość nazwy klasy, dodając liczbę w nawiasach klamrowych, np. **%C{1}**.
10. **%n** - Nowa linia: Wstawia znak nowej linii.
11. **%p** - Poziom (zdeprecjonowany): Wyświetla poziom logowania. Zamiast tego zaleca się używanie **%level**.
12. **%c** - Kategoria (zdeprecjonowany): Wyświetla nazwę kategorii loggera. Zamiast tego zaleca się używanie **%logger**.

13. **%x** - NDC (Nested Diagnostic Context): Wyświetla informacje związane z kontekstem diagnostycznym.
14. **%X** - MDC (Mapped Diagnostic Context): Wyświetla informacje związane z kontekstem diagnostycznym w postaci mapy. Można wybrać konkretną wartość, podając klucz w nawiasach klamrowych, np. **%X{userId}**.
15. **%r** - Czas od uruchomienia: Wyświetla czas, który upłynął od momentu uruchomienia aplikacji do momentu zarejestrowania wpisu w milisekundach.
16. **%R** - Czas relatywny: Wyświetla czas, który upłynął od momentu zarejestrowania poprzedniego wpisu w milisekundach.
17. **%marker** - Marker: Wyświetla marker, który został przypisany do wpisu, jeśli istnieje.
18. **%K** - Klucz mapy: Wyświetla klucz przekazany do metody **MapMessage** jako argument.
19. **%V** - Wartość mapy: Wyświetla wartość przekazaną do metody **MapMessage** jako argument.
20. **%enc** - Kodowanie: Koduje dane wyjściowe według podanego schematu. Na przykład, **%enc{%d %msg}{CRLF}** koduje dane wyjściowe za pomocą "CRLF" (Carriage Return Line Feed).

Możesz łączyć te konwertery wzorców, aby stworzyć niestandardowy wzorzec formatowania logów. Na przykład:

```
<PatternLayout pattern="%d{yyyy-MM-dd HH:mm:ss.SSS} [%t] %-5level
%logger{36} - %msg%n"/>
```

Log4j2 oferuje wiele poziomów logowania, które pozwalają na kontrolowanie tego, jak szczegółowe informacje są rejestrowane przez aplikację. Oto one w kolejności od najniższego do najwyższego:

1. TRACE: Ten poziom logowania jest używany do rejestrowania bardzo szczegółowych informacji o działaniu aplikacji. Jest on użyteczny w trakcie diagnozowania problemów, które są trudne do zidentyfikowania. Zwykle nie jest stosowany na środowiskach produkcyjnych, gdyż generuje dużą ilość danych.
2. DEBUG: Poziom DEBUG jest używany do rejestrowania informacji, które mogą być pomocne podczas debugowania aplikacji. Jest mniej szczegółowy niż TRACE, ale nadal może generować dużą ilość danych. W związku z tym, podobnie jak TRACE, zwykle nie jest stosowany na środowiskach produkcyjnych.
3. INFO: Poziom INFO jest używany do rejestrowania informacji o ogólnym przebiegu działania aplikacji. W przeciwieństwie do TRACE i DEBUG, poziom INFO nie generuje nadmiernej ilości danych, więc może być używany zarówno na środowiskach deweloperskich, jak i produkcyjnych.
4. WARN: Poziom WARN służy do rejestrowania sytuacji, które mogą wskazywać na potencjalne problemy w aplikacji, ale nie powodują jej awarii. Może to obejmować

niespodziewane zachowania, błędy walidacji lub nieprawidłowe konfiguracje. Poziom WARN jest odpowiedni zarówno na środowiskach deweloperskich, jak i produkcyjnych.

5. ERROR: Poziom ERROR jest używany do rejestrowania poważnych błędów, które mają wpływ na działanie aplikacji i mogą prowadzić do awarii lub nieprawidłowego działania. Poziom ERROR powinien być stosowany na wszystkich środowiskach, aby szybko wykrywać i rozwiązywać krytyczne problemy.
6. FATAL: Poziom FATAL jest używany do rejestrowania bardzo poważnych błędów, które zwykle prowadzą do awarii aplikacji lub powodują, że jej działanie jest niemożliwe. Jest to najwyższy poziom logowania pod względem ważności. Zastosowanie logowania na poziomie FATAL jest zazwyczaj ograniczone do sytuacji, w których aplikacja napotyka tak poważne problemy, że wymaga natychmiastowej uwagi i interwencji. Przykładem może być brak dostępu do kluczowych zasobów lub awaria krytycznej usługi.
7. OFF: Poziom OFF nie jest właściwie poziomem logowania, ale specjalnym ustawieniem, które całkowicie wyłącza logowanie dla danego loggera. Jeśli logger ma ustawiony poziom OFF, nie będzie rejestrować żadnych logów, niezależnie od ich ważności. Stosowanie poziomu OFF może być użyteczne w sytuacjach, gdy chcemy całkowicie wyłączyć logowanie dla określonej części aplikacji lub w przypadku optymalizacji wydajności. Należy jednak zachować ostrożność przy użyciu tego poziomu, ponieważ może to prowadzić do utraty ważnych informacji o błędach i ostrzeżeniach, które mogą być istotne dla monitorowania i diagnozowania problemów z aplikacją.

W praktyce poziom OFF powinien być używany tylko wtedy, gdy jest to absolutnie konieczne, a konsekwencje wyłączenia logowania są dobrze zrozumiane i akceptowalne. Warto również pamiętać, że poziom OFF działa na konkretnej instancji loggera, więc pozostałe logi z innych loggerów o różnych poziomach logowania nie są wpływane przez to ustawienie.