Instrukcja laboratoryjna **1**	**Przetwarzanie / Systemy odporne na błędy**
	Temat: Wprowadzenie do systemów odpornych na błędy
	Przygotował: mgr inż. Michał Młodawski

1.Co to są systemy odporne na błędy?

Systemy odporne na błędy stanowią fundamentalny aspekt nowoczesnej inżynierii oprogramowania i architektury systemów informatycznych. W dzisiejszym świecie, gdzie technologia przenika niemal każdy aspekt naszego życia, niezawodność i ciągłość działania systemów komputerowych stają się kluczowe. Wyobraźmy sobie konsekwencje awarii systemu bankowego, kontroli ruchu lotniczego czy infrastruktury energetycznej - mogłyby one prowadzić do poważnych strat finansowych, zakłóceń w transporcie, a nawet zagrożenia życia ludzkiego. Dlatego też projektowanie i implementacja systemów odpornych na błędy nie jest już tylko opcją, ale koniecznością w wielu dziedzinach.

Pojęcie systemów odpornych na błędy odnosi się do zdolności systemu do kontynuowania prawidłowego działania w obliczu awarii sprzętowych, błędów oprogramowania czy innych nieprzewidzianych zdarzeń. Kluczowym celem jest zapewnienie ciągłości usług i ochrona integralności danych, nawet w sytuacjach, gdy poszczególne komponenty systemu ulegają awarii. Osiągnięcie tego celu wymaga zastosowania różnorodnych technik i strategii, które wspólnie tworzą kompleksowe podejście do budowy niezawodnych systemów.

Jednym z podstawowych pojęć w dziedzinie systemów odpornych na błędy jest redundancja. Polega ona na wprowadzeniu dodatkowych, często nadmiarowych komponentów lub zasobów do systemu. Redundancja może przyjmować różne formy, takie jak duplikacja sprzętu, replikacja danych czy równoległe wykonywanie obliczeń. Dzięki temu, w przypadku awarii jednego z elementów, system może płynnie przełączyć się na alternatywny zasób, minimalizując lub całkowicie eliminując przestoje.

Nadmiarowość, ściśle związana z redundancją, odnosi się do celowego wprowadzenia dodatkowych zasobów lub funkcjonalności wykraczających poza minimalne wymagania systemu. Może to obejmować dodatkowe serwery, zwiększoną pojemność pamięci masowej czy rozbudowane mechanizmy kontroli błędów. Nadmiarowość stanowi swego rodzaju bufor bezpieczeństwa, pozwalający systemowi na absorpcję zwiększonego obciążenia lub radzenie sobie z nieoczekiwanymi sytuacjami bez utraty wydajności czy stabilności.

Kolejnym kluczowym aspektem systemów odpornych na błędy jest zdolność do odtwarzania po awarii. Obejmuje to szereg technik i procedur mających na celu przywrócenie systemu do stanu operacyjnego po wystąpieniu poważnej awarii. Proces odtwarzania może obejmować automatyczne przełączanie na systemy zapasowe, odtwarzanie danych z kopii zapasowych czy uruchamianie specjalnych procedur naprawczych. Skuteczne strategie odtwarzania po awarii wymagają nie tylko odpowiednich rozwiązań technicznych, ale także starannego planowania, regularnych testów i ciągłego doskonalenia procedur.

W kontekście implementacji systemów odpornych na błędy, programiści i architekci systemów muszą brać pod uwagę szereg czynników. Obejmują one projektowanie architektury systemu w sposób umożliwiający łatwe dodawanie redundancji, implementację mechanizmów wykrywania i izolacji błędów, a także tworzenie efektywnych strategii

przełączania i synchronizacji między redundantnymi komponentami. Dodatkowo, kluczowe znaczenie ma monitorowanie stanu systemu w czasie rzeczywistym oraz implementacja zaawansowanych mechanizmów logowania i analizy błędów, które pozwalają na szybkie diagnozowanie i rozwiązywanie problemów.

Warto zauważyć, że budowa systemów odpornych na błędy nie ogranicza się jedynie do aspektów technicznych. Równie istotne są kwestie organizacyjne, takie jak odpowiednie szkolenie personelu, tworzenie i aktualizacja planów awaryjnych czy regularne przeprowadzanie testów i symulacji awarii. Holistyczne podejście do odporności na błędy obejmuje również analizę ryzyka, zarządzanie ciągłością działania oraz ciągłe doskonalenie procesów i procedur.

W miarę jak systemy informatyczne stają się coraz bardziej złożone i wzajemnie powiązane, wyzwania związane z zapewnieniem ich odporności na błędy również rosną. Nowe technologie, takie jak chmury obliczeniowe, systemy rozproszone czy Internet Rzeczy, wprowadzają dodatkowe warstwy złożoności i potencjalne punkty awarii. Jednocześnie jednak oferują one nowe możliwości w zakresie budowy systemów odpornych na błędy, np. poprzez łatwiejsze skalowanie zasobów czy wykorzystanie geograficznie rozproszonych centrów danych.

2.Wstęp do przetwarzania danych w Javie

W dzisiejszym świecie, gdzie dane stają się coraz bardziej cennym zasobem, umiejętność efektywnego przetwarzania i zarządzania nimi jest kluczowa dla każdego programisty. Java, jako jeden z najpopularniejszych języków programowania, oferuje szeroki zakres narzędzi i technik do pracy z danymi. Przetwarzanie danych w Javie to obszerny temat, który obejmuje wiele różnych aspektów i technik. Od podstawowych operacji na strumieniach, przez zaawansowane techniki serializacji, aż po metody zabezpieczania danych – każdy z tych elementów odgrywa kluczową rolę w tworzeniu efektywnych i bezpiecznych aplikacji Java.
W kolejnych sekcjach tego dokumentu zagłębimy się w szczegóły każdego z tych tematów, dostarczając przykłady kodu i omawiając najlepsze praktyki. Zrozumienie tych koncepcji pozwoli programistom Java na tworzenie bardziej wydajnych, bezpiecznych i niezawodnych aplikacji, zdolnych do efektywnego przetwarzania i zarządzania danymi w różnorodnych scenariuszach.

3.Co to są strumienie danych?

Czasami nasz program powinien przetwarzać dane znajdujące się poza nim lub zapisywać wyniki w zewnętrznym miejscu docelowym. Java zapewnia abstrakcję zwaną **strumieniem** , która ujednolica pracę z dyskami, plikami, lokalizacjami sieciowymi i innymi zasobami. W pewnym sensie strumień Java jest podobny do strumienia wody w świecie rzeczywistym, który ma początek (źródło) i koniec (przeznaczenie). W oparciu o te same zasady strumienie we/wy można podzielić na dwie grupy:

- **Strumień wejściowy,** który odczytuje dane ze źródła;
- **Strumień wyjściowy** , który zapisuje dane w określonym miejscu docelowym.

Strumienie można dalej podzielić na dwie kategorie na podstawie tego, jak reprezentują sekwencje danych:

- **Strumienie bajtów** używane do odczytu i zapisu danych w bajtach;
- **Strumienie znaków** używane do odczytywania i zapisywania danych w postaci znaków zgodnie z 16-bitowym formatem Unicode.

Strumienie znaków znacznie ułatwiają programistom przetwarzanie danych tekstowych. W porównaniu z nimi strumienie bajtów są dość niskopoziomowe, ale mogą pracować z danymi dowolnego typu, w tym z multimediami.

Niektóre strumienie używają tymczasowej lokalizacji w pamięci. Na początku takie strumienie odczytują lub zapisują dane w tymczasowej lokalizacji, a następnie dane są z niej przenoszone do źródła lub miejsca docelowego. Ta tymczasowa lokalizacja to zazwyczaj tablica bajtów lub znaków zwana **buforem**, a cały proces nazywa się **buforowaniem**. Powodem wprowadzenia pośredniej lokalizacji pamięci jest to, że odwołanie się do niektórych źródeł lub miejsc docelowych zajmuje znaczny odstęp czasu. Buforowanie jest więc rodzajem optymalizacji, która minimalizuje liczbę interakcji z nimi.

Niektóre strumienie wejściowe mają również funkcję buforowania. Gdy strumień odczytuje dane po raz pierwszy, odczytuje tyle, ile może pomieścić bufor. Nawet jeśli zażądano tylko kilku bajtów lub znaków, buforowany strumień wejściowy będzie odczytywał bajty do momentu zapełnienia bufora. Następny odczyt najpierw sprawdza, czy w buforze są jakieś nieprzeczytane dane. W przypadku, gdy bufor zawiera jakieś nieprzeczytane dane, strumień pobiera je z bufora i nie musi wchodzić w interakcje ze źródłem. W przeciwnym razie żąda danych ze źródła, tak jak za pierwszym razem.

4.Strumienie wejściowe

Dane można uzyskać z wielu punktów uznawanych za „dostawców treści". Oprócz standardowych danych wejściowych lub plików mogą to być na przykład połączenia sieciowe, bufory w pamięci, a nawet obiekty. Wszystkie z nich nazywane są **źródłami strumieni wejściowych**. W rzeczywistości źródłem są dowolne dane, które mogą być wykorzystywane i przetwarzane przez program. Praca z danymi to dość specyficzna rzecz, a każde źródło potrzebuje wyspecjalizowanej klasy.

4.1.Strumienie znaków

Istnieje kilka klas do czytania tekstu. Są one nazywane strumieniami wprowadzania znaków i umożliwiają odczytywanie danych tekstowych: typu String lub char. Przykładowe klasy: **FileReader**, **CharArrayReader**, czy **StringReader**. Jak można zauważyć nazwa klasy wskazuje jakiego typu źródła używa jako danych wejściowych. Wynika to z faktu, że wszystkie te klasy rozszerzają klasę Reader. Każda klasa zawiera zestaw przydatnych metod, ale mają też wspólne metody odczytywania danych:

- **int read()** - odczytuje pojedynczy znak. Jeśli zostanie osiągnięty koniec strumienia, metoda zwraca wartość -1, w przeciwnym razie zwraca liczbową reprezentację znaku zgodnie z bieżącym kodowaniem;
- **int read(char[] buff)** odczytuje sekwencję znaków do przekazanej tablicy aż do jej pojemności i zwraca liczbę faktycznie odczytanych znaków. Może również zwrócić -1 w przypadku, gdy żadne dane nie zostały odczytane;

Te metody zwracają liczbę faktycznie odczytanych znaków lub -1. Blokują one również uruchomienie programu do czasu, gdy dostępne są dane wejściowe lub do osiągnięcia końca strumienia. Inną ważną metodą jest to **void close()**, należy ją wywoływać po użyciu strumienia.

4.2.Czytanie znaków

Rozważmy **FileReader** jako przykład klasy Reader. **FileReader** posiada zestaw konstruktorów. Tutaj są niektóre z nich:

- **new FileReader(String fileName);**
- **new FileReader(String fileName, Charset charset);**
- **new FileReader(File file);**
- **new FileReader(File file, Charset charset);**

Charset to klasa, która deklaruje kodowanie od sekwencji bajtów do znaków. Domyślnie java używa kodowania UTF-16, odpowiedniego do większości zadań. Jednak czasami plik może mieć inne kodowanie i będziemy musieli użyć innego zestawu znaków, aby poprawnie odczytać zawartość pliku.

4.3.Strumienie bajtów

Nazwa klasy strumienia bajtów wskazuje, jakiego typu źródło używa jako danych wejściowych i zwykle kończy się na **InputStream**, ponieważ wszystkie takie klasy rozszerzają klasę **java.io.InputStream**. Wszystkie klasy strumieni bajtów mają metody czytania podobne do strumieni wejściowych znaków:

- **abstract int read()** - odczytuje pojedynczy bajt;
- **int read(byte[] b)** - odczytuje liczbę bajtów i przechowuje je w tablicy bajtów;
- **byte[] readAllBytes()** - odczytuje wszystkie bajty.

Metoda, która wczytuje bajty do tablicy, zwraca int wartość. Jest to liczba faktycznie odczytanych bajtów ze źródła. Jeśli -1 zwracana jest wartość, oznacza to, że żadne bajty nie zostały odczytane. Każda klasa strumienia wejściowego ma również metodę **void close()**, która zwalniania zasobów systemowych. Załóżmy, że mamy plik java.txt zawierający następujący tekst: java. Przeczytajmy to za pomocą klasy **FileInputStream**. Można go stworzyć za pomocą kilku konstruktorów, takich jak:

- **new FileInputStream(String pathToFile);**
- **new FileInputStream(File file);**

Tutaj tworzymy strumień wejściowy pliku, określając nazwę pliku:

```
FileInputStream inputStream = new FileInputStream("java.txt");
```

Poniższy kod czyta pierwsze dziesięć bajtów:

```
FileInputStream inputStream = new FileInputStream("java.txt");

byte[] bytes = new byte[10];

int numberOfBytes = inputStream.read(bytes);

System.out.println(numberOfBytes); inputStream.close();
```

5.Strumienie wyjściowe

Strumień wyjściowy umożliwia zapisywanie danych w **miejscu docelowym**. Już wcześniej poznaliśmy takie miejsce docelowe, a była nim konsola. Miejscem docelowym mogą być również dyski, pliki, bufor pamięci, gniazda sieciowe lub inne lokalizacje sieciowe. Ogólnie rzecz biorąc, miejsce docelowe jest docelowym punktem końcowym, do którego docierają dane wysyłane do strumienia wyjściowego. Biblioteka standardowa Java zapewnia szeroką gamę klas do reprezentowania strumienia wyjściowego. Dość duża liczba tych klas jest wypadkową kilku czynników. Jednym z nich jest to, że każdy cel wymaga określonego sposobu pisania do niego.

5.1.Strumienie znakowe

Strumienie wyjściowe znaków umożliwiają zapisywanie danych tekstowych: char lub String. Przykładem takich strumieni są: **FileWriter** i **PrintWriter** i wykorzystuje się je do zapisywania danych tekstowych do plików. Oba, podobnie jak inne strumienie wyjściowe znaków, mają wspólnego abstrakcyjnego przodka **java.io.Writer**. Należy pamiętać o metodzie **close()**,

którą należy wywoływać w celu zapobiegania wyciekom zasobów. Poniżej wymieniono główne metody wykorzystywane do zapisu plików:

- **void write(char[] buff)** - zapisuje tablicę znaków;
- **void write(char[] buff, int off, int len)** - zapisuje część tablicy znaków;
- **void write(int c)** - pisze pojedynczy znak;
- **void write(String str)** - pisze ciąg;
- **void write(String str, int off, int len)** - pisze fragment łańcucha;

Writer ma kilka bezpośrednich podklas do różnych celów w standardowej bibliotece. Na przykład **FileWriter** jest przeznaczony do zapisywania do plików. **StringWriter** jest przeznaczony do konstruowania ciągu, a **CharArrayWriter** używa char[] jako miejsce docelowe. Przykładowo poza tym metody dziedziczone ze klasy **Writer** ma swoje własne metody **toCharArray()** i **writeTo**. Te pierwszy dostarczą treści w char[], a ostatni pisze treść do innego pisarza.

```
CharArrayWriter terminyPoprawkowe = new CharArrayWriter();

FileWriter tp1 = new FileWriter("pierwszyTermin.txt", true);

FileWriter tp2 = new FileWriter("drugiTermin.txt", true);


terminyPoprawkowe.write("Java: 15.06.2022, 08:30 Aula Główna 3");

terminyPoprawkowe.writeTo(tp1);

terminyPoprawkowe.writeTo(tp2);


char[] array = terminyPoprawkowe.toCharArray();

tp1.close();

tp2.close();

terminyPoprawkowe.close();
```

5.2.Strumienie bitów

Z punktu widzenia komputera wszelkie dane to tylko sekwencja bitów: 0 lub 1, które są zwykle składane w bajty po 8 cyfr. Innymi słowy, wszelkie dane są reprezentowane jako szeregowy zestaw bajtów. Oznacza to, że obrazy, audio, wideo itd. mają format binarny, tj. reprezentowany jako sekwencja bajtów. W rzeczywistości pliki tekstowe również mają reprezentację bajtową: jeśli pamiętasz, znaki są kombinacjami bajtów. Java ma zestaw klas zwanych **strumieniami wyjściowymi bajtów** do zapisywania bajtów. Klasy strumienia wyjściowego bajtów z biblioteki standardowej rozszerzają java.io.OutputStream klasę abstrakcyjną.

Klasa zawiera trzy metody pisania:

- **void write(byte[] b)** - zapisuje tablicę bajtów;
- **void write(byte[] b, int off, int len)** - zapisuje część tablicy bajtów;
- **abstract void write(int b)** - zapisuje jeden bajt;

Podobnie jest w podklasie **OutputStream** z biblioteki standardowej. Jest przeznaczony do zapisywania danych do pliku jako miejsca docelowego. Jak można się domyślić, umożliwia pisanie do miejsca docelowego. Takie klasy, nie mają miejsca docelowego tzn. punktu końcowego i zapisują dane w innych strumieniach wyjściowych. Klasy te mają być strumieniami pośrednimi do transformacji danych lub ewentualnie zapewniać dodatkową funkcjonalność. Przykładem może być **FileOutputStream**. Klasa posiada zestaw konstruktorów. Niektórzy z nich są:

- **FileOutputStream(String fileName)**
- **FileOutputStream(String fileName, boolean append)**
- **FileOutputStream(File file)**
- **FileOutputStream(File file, boolean append)**

Gdzie parametr **append** wskazuje, czy dołączyć **true**, czy nadpisać **false** istniejący plik. Warto wiedzieć, że **FileOutputStream** utworzy plik o podanej nazwie, jeśli jeszcze nie istnieje. Tworzy plik zaraz po **FileOutputStream** zainicjowaniu, nawet jeśli nie próbowałeś do niego pisać.

```
byte[] java = new byte[] {'j', 'a', 'v', 'a'};

OutputStream file = new FileOutputStream("java.txt", false);

file.write(java);

file.close();
```

Zauważmy, że wszystkie metody strumieni bajtów omówione powyżej pozwalają na zapisywanie tylko bajtów. Oznacza to, że nie możemy bezpośrednio pisać łańcuchów, musimy byte[] wcześniej przekonwertować. Więc jeśli chcemy napisać ciąg do pliku, musimy najpierw przekonwertować go na bajty. Na przykładowo można użyć do tego metody **getBytes()**.

```
String stringValue = "stream";

byte[] stringAsBytes = stringValue.getBytes();
```

5.3.Buforowanie strumieni danych

Strumienie wyjściowe mają dwie klasy ze standardowej biblioteki, które wykonują buforowanie. BufferedOutputStream opiera się na zasadzie buforowania. Ma tylko dwóch konstruktorów:

- **BufferedOutputStream(OutputStream out)**
- **BufferedOutputStream(OutputStream out, int size)**

Te klasy są pośrednimi strumieniami wyjściowymi. Przyjmują strumień wyjściowy jako dane wejściowe i wykonują buforowanie przed delegowaniem do innego strumienia. Dodatkowym parametrem **size** jest wielkość bufora. Jeśli chcemy uwolnić wszystkie dane z bufora, zapisując je w miejscu docelowym, możesz użyć metody **flush()**. Zwykle jest wywoływana automatycznie, gdy bufor jest pełny lub przed zamknięciem strumienia. Strumień wyjściowy to sposób zapisywania danych w miejscu docelowym. Miejscem docelowym jest docelowy punkt końcowy danych, którym może być plik, konsola, a nawet gniazdo sieciowe. Strumienie dzielą się na bajtowe i znakowe. Strumienie wyjściowe bajtów umożliwiają zapisywanie sekwencji bajtów. Jest niezbędny do pracy z plikami binarnymi. Strumienie wyjściowe znaków są przeznaczone do pisania tekstu. Klasy strumienia wyjściowego znaków zwykle kończą się na Writer , ponieważ **java.io.Writer** z reguły rozszerzają one jedną klasę abstrakcyjną. Podobnie strumienie wyjściowe bajtów kończą się na **OutputStream**. Niektóre strumienie używają buforowania pod maską. Jest to szeroko stosowana optymalizacja, która stara się zminimalizować kosztowną interakcję z miejscem docelowym.

5.4.Tworzenie plików

Często program musi przetwarzać i przechowywać dane znajdujące się poza bazą. Najprostszym sposobem przechowywania danych jest użycie plików obsługiwanych przez wszystkie nowoczesne systemy operacyjne. Plik można traktować jako zbiór danych przechowywanych na dysku lub innym urządzeniu, którym można manipulować jak pojedynczą jednostką, gdy zostanie zaadresowany jego nazwą. Pliki można organizować w katalogi, które działają jak foldery dla innych plików i katalogów. W java.io pakiecie znajduje się klasa o nazwie File. Obiekt tej klasy reprezentuje istniejący lub nieistniejący plik lub katalog. Klasa może służyć do manipulowania plikami i katalogami: tworzenia, usuwania, uzyskiwania dostępu do właściwości i nie tylko.

Najprostszym sposobem utworzenia obiektu pliku jest przekazanie ścieżki ciągu do jego konstruktora. Prawidłowy format ciągu zależy od systemu operacyjnego:

- Windows używa ukośników odwrotnych dla ścieżek ('\'),
- Linux, OS X, Android i inne systemy typu UNIX używają ukośnika ('/').

Należy pamiętać o tej różnicy podczas pracy z plikami. Jeśli system operacyjny to Windows, nie można zapomnieć o użyciu znaku ucieczki '\'.

```
File fileOnWin = new File("E:\\psk\\java\\zaliczenie.pdf"); // Path on
Windows system

File fileOnUnix = new File("/home/mm/java/Documents/zaliczenie.pdf"); //
Path on a UNIX system
```

Kod będzie działał, nawet jeśli plik lub katalog w rzeczywistości nie istnieje w twoim systemie plików. Nie tworzy nowego pliku ani katalogu. Reprezentuje po prostu „wirtualny" plik lub katalog, który już istnieje lub może zostać utworzony w przyszłości. Możemy rozróżnić dwie ścieżki, bezwzględne, które były w wcześniejszych przykładach i względne. Także prostu ścieżka jest bezwzględna, jeśli zaczyna się od elementu głównego systemu plików. Zawiera pełne informacje o lokalizacji pliku, w tym typ systemu operacyjnego. Lokalizacja pliku przy użyciu jego bezwzględnej ścieżki w programach jest uważana za złą praktykę, ponieważ traci się możliwość ponownego użycia programu na różnych platformach. Innym problemem jest to, że nie można przesłać pliku wraz z określonym katalogiem. Ścieżka względna to ścieżka, która nie zawiera elementu głównego systemu plików. To zawsze zaczyna się od katalogu roboczego. Ten katalog jest reprezentowany przez . (kropkę). Ścieżka względna nie jest kompletna i musi być połączona z bieżącą ścieżką katalogu, aby dotrzeć do żądanego pliku.

5.5.Podstawowa implementacja pisarzy

Wystąpienie File ma listę metod. Spójrz na niektóre z nich:

- **String getPath()** - zwraca ścieżkę ciągu do tego pliku lub katalogu;
- **String getName()** - zwraca nazwę tego pliku lub katalogu (tylko nazwisko ścieżki)
- **boolean isDirectory()** - zwraca true, jeśli jest katalogiem i istnieje, w przeciwnym razie false;
- **boolean isFile()** - zwraca true, jeśli jest to plik, który istnieje (nie katalog), w przeciwnym razie, false;
- **boolean exists()** - zwraca true, jeśli ten plik lub katalog rzeczywiście istnieje w twoim systemie plików, w przeciwnym razie false;
- **String getParent()** - zwraca ścieżkę ciągu do katalogu nadrzędnego, który zawiera ten plik lub katalog.

Lista nie jest kompletna, wszystkie metody znajdziemy tutaj.

```
File file = new File("E:\\java\\zaliczenie.pdf");

System.out.println("Exists: " + file.exists());

System.out.println("File name: " + file.getName());

System.out.println("File path: " + file.getPath());

System.out.println("Parent path: " + file.getParent());

System.out.println("Is file: " + file.isFile());

System.out.println("Is directory: " + file.isDirectory());
```

Do łatwiejszego umieszczania danych w plikach można użyć klas takich jak: FileWriter, czy PrintWriter obie rozszerzają klasę abstrakcyjną Writer i mają wiele podobieństw. Jest jednak PrintWriter operuje bardziej na wysokim poziomie i zapewnia kilka przydatnych metod. Wśród nich są metody formatowania i przeładowane metody drukowania do pisania typów pierwotnych.

5.5.1.Implementacja klasy FileWriter

Klasa **FileWriter** posiada zestaw konstruktorów do zapisywania znaków i łańcuchów do określonego pliku:

- **FileWriter(String fileName);**
- **FileWriter(String fileName, boolean append);**
- **FileWriter(File file);**
- **FileWriter(File file, boolean append);**

Dwa konstruktory przyjmują dodatkowy parametr **append**, który wskazuje, czy dodać (true), czy nadpisać (false) istniejący plik.

Wszystkie konstruktory mogą rzucać wyjątek IOException z kilku powodów:

- jeśli podany plik istnieje, ale jest katalogiem;
- jeśli plik nie istnieje i nie można go utworzyć;
- jeśli plik istnieje, ale nie można go otworzyć.

Kod poniżej dołącza do pliku nowy wiersz. Zauważmy, że używanie łamania wiersza w różnią zależności od używanej platformy::

- \n - System operacyjny podobny do uniksa;
- \r\n - System operacyjny Windows;

```
File file = new File("E:\\psk\zaliczenia\\java\\zaliczenie.txt");

FileWriter writer = new FileWriter(file, true);

writer.write("Java\r\n");

writer.close();
```

Ważne jest, aby zamknąć a FileWriter po użyciu, aby uniknąć wycieku zasobów. Odbywa się to poprzez wywołanie metody close: writer.close(); Od wersji Java 7 wygodnym sposobem na zamknięcie obiektu FileWriter jest użycie instrukcji try-with-resources .

Automatycznie zamknie pisarza:

```
File file = new File("E:\\psk\zaliczenia\\java\\zaliczenie.txt");

try (FileWriter writer = new FileWriter(file)) {

    writer.write("Java");

} catch (IOException e) {

   // wyjatek

}
```

5.5.2.Implementacja klasy PrintWriter

Klasa PrintWriter pozwala na zapisanie sformatowanych danych do pliku. Może wyprowadzać łańcuchy, typy podstawowe, a nawet tablicę znaków. Klasa ma kilka konstruktorów. Niektóre z nich są podobne do konstruktorów FileWriter:

- PrintWriter(String fileName);
- PrintWriter(File file).

Inne pozwalają na przekazanie FileWriter jako klasy, która rozszerza klasę abstrakcyjną Writer: **PrintWriter(Writer writer).**

```
File file = new File("E:\\psk\zaliczenia\\java\\zaliczenie.txt");

try (PrintWriter printWriter = new PrintWriter(file)) {

    printWriter.print("Hello");

    printWriter.print(" Java");

    printWriter.print(20);

    printWriter.print(36);

} catch (IOException e) {

    System.out.printf(e.getMessage());

}
```

6.Zarządzanie plikami

Klasa java.io.File reprezentuje abstrakcyjną ścieżkę do pliku lub katalogu, który może nawet nie istnieć. Oprócz przechodzenia przez hierarchię plików, z poziomu tej klasy jesteśmy w stanie zarządzać plikami i katalogami, tworząc, usuwając i zmieniając ich nazwy.

6.1.Tworzenie plików

Aby utworzyć plik w systemie plików, musimy wykonać następujące czynności:

- Utwórz wystąpienie java.io.File z określoną ścieżką abstrakcyjną.
- Wywołaj createNewFile metodę tego wystąpienia.

Po utworzeniu instancji File należy wywołać metodę createNewFile. Metoda zwraca true, jeśli plik został pomyślnie utworzony i false jeśli już istnieje. Nie usuwa zawartości istniejącego pliku.

```
File file = new File("/mmlodawski/psk/java/lab9/file.txt");

try {

    boolean createdNew = file.createNewFile();

    if (createdNew) {

        System.out.println("Plik został utworzony.");

    } else {

        System.err.println("Plik już istnieje.");

    }

} catch (IOException ignored) {

    System.err.println("Problem z utworzeniem pliku");

}
```

6.2.Tworzenie katalogów

Aby utworzyć katalog, musimy również zacząć od utworzenia instancji java.io.File. Następnie powinniśmy wywołać jedną z dwóch metod tej instancji:

- **boolean mkdir** tworzy katalog; zwraca true tylko wtedy, gdy katalog został utworzony, w przeciwnym razie zwraca false.
- **boolean mkdirs** tworzy katalog zawierający wszystkie niezbędne nieistniejące katalogi nadrzędne; zwraca true tylko wtedy, gdy katalog został utworzony wraz ze wszystkimi określonymi katalogami nadrzędnymi.

Obie metody nie rzucają wyjątku IOException, w przeciwieństwie do metody createNewFile.

6.3.Usuwanie katalogów

Do usuwania katalogów bądź plików istnieje metoda o nazwie, **delete**. Zwraca true wtedy i tylko wtedy, gdy plik lub katalog zostanie pomyślnie usunięty lub false, jeśli plik lub katalog nie istnieje. Należy pamiętać, że zwraca również, false jeśli katalog zawiera podkatalogi lub pliki. Oznacza to, że metoda nie usunie hierarchii, a jedynie konkretny plik lub pusty katalog.

```
File file = new File("E:\\psk\\java\\zaliczenia\\oceny.txt");

if (file.delete()) {

    System.out.println("Plik został usunięty");

} else {

    System.err.println("Problem z usunięciem pliku");

}
```

Istnieje również inna metoda usuwania plików. Jest wywoływany deleteOnExit i usuwa plik lub katalog po zatrzymaniu programu. Pamiętaj, że po zażądaniu usunięcia nie ma możliwości jego anulowania.

6.4.Zmiana nazw i przenoszenie plików

Metoda renameTo zmienia nazwę pliku, edytując go w abstrakcyjnej ścieżce. Zwraca true wtedy i tylko wtedy, gdy zmiana nazwy się powiodła, w przeciwnym razie zwraca false. Wiele aspektów zachowania tej metody pozostaje zależnych od platformy. Może nie być w stanie przenieść pliku z jednego systemu plików na inny i może się nie powieść, jeśli plik o tym samym miejscu docelowym już istnieje. Zwracana wartość powinna być zawsze sprawdzana, aby upewnić się, że operacja się powiodła. Metoda renameTo zgłasza NullPointerException w przypadku, gdy plik docelowy ma wartość NULL. Większość rozważanych metod zwraca wartość false w przypadku, gdy system nie ma uprawnień do wykonania odpowiedniej operacji: zmiany nazwy, przeniesienia lub usunięcia plików i katalogów. Jednak metoda create zgłasza wyjątek java.lang.IOException w podobnych przypadkach.

```
File file = new File("E:\\psk\java-2022\zaliczenia\zaliczenie0termin.txt");
File newFile = new File("E:\\psk\java-
2022\zaliczenia\zaliczenie1termin.txt");
boolean checkFile = file.renameTo(newFile);
if (checkFile) {
    System.out.println("Nazwa pliku została zmieniona.");
} else {
    System.err.println("Problem ze zmianą nazwy pliku.");
}
```

7.Co to jest serializacja i deserializacja

Formalnie wszystkie dane w aplikacji są dostępne tylko w czasie jej wykonywania. W momencie zatrzymania aplikacji i uruchomienia jej ponownie zaczynamy pracę od zera bez zachowania poprzedniego wyniku pracy. Na szczęście w Java istnieje mechanizm zapisywania obiektów w pamięci stałej np. dysku twardym, czy dysku przenośnym i odczytaniu ich przy następnym uruchomieniu programów. Na tym właśnie polega proces serializacji danych, czyli konwertuje obiekt na strumień bitów. Procesem odwrotnym jest deserializacja, odczytuje strumień bitów i rekonstruuje do rzeczywistego obiektu. Istnieje wiele metod serializacji danych, dane można przechowywać w plikach „jawnych" opartych o format JSON, YALM, czy XML lub w plikach binarnych takich jak BJSON, czy z wykorzystaniem domyślnego protokołu serializacji binarnej dostępnej w Java. Oczywiście takie pliki można wymieniać pomiędzy innymi użytkownikami, co pozwala na łatwiejszą wymianę danych.

7.1.Wykorzystanie serializacji

Żeby klasa była możliwa do serializacji należy zaimplementować interfejs **Serializable**. Jest to interfejs informujący kompilator, że zachowuje się w niestandardowy sposób. Oczywiście taka klasa może przechowywać dowolne obiekty w tym obiekty innej klasy, ale one także muszą mieć zaimplementowany interfejs Serializable. Możemy także ochronić dane pole przed serializacją stosując słowo kluczowe **transient**.

Dobrą praktyką programistyczną jest dodanie specjalnego pola o określonej wartości:

```
private static final long serialVersionUID = 2022L;
```

Pole to ma na celu sprawdzenie kompatybilności obiektów u odbiorcy są i czy załadowanie klasy dla tego obiektu są poprawne. Jeśli numery wersji klas nadawcy i odbiorcy nie są zgodne wtedy zostanie wyrzucony wyjątek **InvalidClassException**. Wartość pola powinna być zwiększana za każdym razem gdy dochodzi do zmiany wartości pól.

Zaprojektujmy klasę student oraz przedmiot wykorzystujący mechanizm serializacji danych. Klasa student będzie posiadać pola imię, nazwisko, numer indeksu, listę przedmiotów, która zawiera nazwę i ocenę jaką uzyskał student oraz pole PESEL, które zostało oflagowane słowem kluczowym **transient**. Klasa przedmiot posiada dwa pola z nazwą przedmiotu i oceną jaką uzyskał.

```java
package com.example.demo.lab6;

import lombok.AllArgsConstructor;
import lombok.Data;

import java.io.Serializable;
import java.util.Arrays;
import java.util.List;

@Data
public class Student implements Serializable {
    private static final long serialVersionUID = 20032022L;
    private transient Long peselNumber;
    private String name;
    private String surname;
    private Integer indexNumber;
    private final List<Subject> listOfSubjects = Arrays.asList(new
Subject("Architektura Systemów Komputerowych", 4.5F), new
Subject("Programowanie obiektowe w Java", 5.0F));

    public Student(Long peselNumber, String name, String surname, Integer
indexNumber) {
        this.peselNumber = peselNumber;
        this.name = name;
        this.surname = surname;
        this.indexNumber = indexNumber;
    }

    public Student(Long peselNumber, String name, String surname) {
        this.peselNumber = peselNumber;
        this.name = name;
        this.surname = surname;
    }

    // Konstruktor, gettery, settery, toString();
}
```

```java
package com.example.demo.lab6;

import lombok.AllArgsConstructor;
import lombok.Data;

import java.io.Serializable;

@AllArgsConstructor
@Data
public class Subject implements Serializable {
    private static final long serialVersionUID = 19032022L;
    private String name;
    private Float  grade;
    // Konstruktor, gettery, settery, toString();
}
```

Na koniec stwórzmy nasz obiekt w metodzie main i zaimplementujmy potrzebne metody do serializacji i deserializacji danych:

```
package com.example.demo.lab6;

import java.io.*;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.util.Base64;
import java.util.List;

public class lab6 {
    public static void main(String[] args) throws IOException,
ClassNotFoundException {
        final String pathToFile = "E:\\psk-java\\student.data";
        final Student student = new Student(22301559573L,"Michał",
"Młodawski",92136);
        serializeObjectToFile(student,pathToFile);

        final Student newStudent = (Student)
deSerializeObjectFromFile(pathToFile);
        List<Subject> subjectList = newStudent.getListOfSubjects();

        System.out.println(newStudent.toString());
        for (Subject subject : subjectList) {
            System.out.println(subject.toString());
        }
    }

    /**
     *  Metoda wykorzystująca mechanizm serializacji obiektu do pliku.
Dokonuje kodowania danych z wykorzystaniem Base64.
     * @param serializable Obiekt poddany mechanizmowi serializacji.
     * @param pathFile Pełna ścieżka do pliku, do którego ma zostać zapisany
serializowany obiekt.
     * @throws IOException Wyjątek I/O.
     */
    private static void serializeObjectToFile(Serializable serializable,
String pathFile) throws IOException, IOException {
        final FileOutputStream outputStream = new FileOutputStream(pathFile);
        final ByteArrayOutputStream byteArrayOutputStream = new
ByteArrayOutputStream();
        final ObjectOutputStream objectOutputStream = new
ObjectOutputStream(byteArrayOutputStream);
        objectOutputStream.writeObject(serializable);
        objectOutputStream.close();

outputStream.write(Base64.getEncoder().encodeToString(byteArrayOutputStream.to
ByteArray()).getBytes(StandardCharsets.UTF_8));
        outputStream.close();
    }

    /**
     * Metoda wykorzystująca mechanizm deserializacji pliku do obiektu.
     * @param pathFile  Pełna ścieżka do pliku, do którego ma zostać zapisany
```

```
serializowany obiekt.
     * @return deserializowany obiekt.
     * @throws IOException Wyjątek I/O.
     * @throws ClassNotFoundException Nie znaleziono definicji klasy o
określonej nazwie podczas deserializacji.
     */
    private static Object deSerializeObjectFromFile(String pathFile) throws
IOException, ClassNotFoundException {
        final String file = Files.readAllLines(Paths.get(pathFile)).get(0);
        byte[] data = Base64.getDecoder().decode(file);
        ObjectInputStream objectInputStream = new ObjectInputStream(new
ByteArrayInputStream(data));
        Object object = objectInputStream.readObject();
        objectInputStream.close();
        return object;
    }

}
```

W kodzie powyżej zostały zaimplementowane dwie nowe metody, serializeObjectToFile oraz deSerializeObjectFromFile, pierwsza z nich ma na celu dokonać serializacji obiektu do pliku i na koniec zakodować wartości binarne z wykorzystaniem kodowanie Base64, które zapewnia „ludzką formę" danych. Druga metoda ma na celu z odczytanego pliku zamienić go na obiekt. Do obydwu metod podajemy pełną ścieżkę do pliku – tak jak w przykładzie.

Wynik działania programu:

```
Student{peselNumber=null, name='Michał', surname='Młodawski',
indexNumber=92136}
Subject{name='Architektura Systemów Komputerowych', grade=4.5}
Subject{name='Programowanie obiektowe w Java', grade=5.0}

Process finished with exit code 0
```

Jak widzimy plik został poprawnie serializowany i odczytany ponownie. Wartość pola PESEL nie została uwzględniony w procesie serializacji, więc po odczytaniu ma wartość **null**.

7.2.Niestandardowa serializacja

Czasami zdarza się, że obiekty ulegają zmianą w czasie życia projektu, jeżeli będziemy chcieli deserializować stary obiekt nowymi wartościami dostaniemy wyjątek. Są też sytuacje kiedy potrzebujemy pól walidacji poprawności. Istnieje wiele innych powodów, dla których warto korzystać z serializacji niestandardowej np. kiedy musimy dokonać skomplikowanej serializacji, czy jeżeli chcemy dodatkowo zabezpieczyć ważne pola klasy.

W przypadku niestandardowej serializacji przychodzą nam z pomocą dwie funkcje **writeObject()** i **readObject()**. Pierwszą nowością jest to, że te metody po prostu implementujemy bez konieczności ich przeciążania, czy implementowania poprzez interfejs. W momencie wywołania metody writeObject() lub readObject() maszyna wirtualna sprawdza JVM, czy została dokonana implementacja metod, jeśli tak, to zostaje wykonany kod z tych

metod. Załóżmy, że chcemy zabezpieczyć numer PESEL studenta, przesłanie go w jawnej formie byłoby zbyt ryzykowne, należy go jakoś zaszyfrować. Możemy wykorzystać wiele różnych algorytmów szyfrowania, jednym z bardziej popularnych jest AES, użyjemy gotowej implementacji tych metod i skupimy się tylko na niestandardowej serializacji.

```
import static com.mmlodawski.serializacja.AESEncryption.decryptData;
import static com.mmlodawski.serializacja.AESEncryption.encryptData;


public class Student implements Serializable {
    private static final long serialVersionUID = 20032022L;
    private transient Long peselNumber;
    private String name;
    private String surname;
    private Integer indexNumber;

    private void writeObject(ObjectOutputStream oos) throws Exception {
        oos.defaultWriteObject();
        String encryptPesel = encryptData(peselNumber.toString());
        oos.writeObject(encryptPesel);
    }

    private void readObject(ObjectInputStream ois) throws Exception {
        ois.defaultReadObject();
        this.peselNumber =
Long.valueOf(Objects.requireNonNull(decryptData((String)
ois.readObject())));
    }
        // Konstruktor, gettery, settery, toString();
}
```

Metoda **writeObject** tworzy nowe pole typu String, które zawiera zaszyfrowany numer PESEL i dołącza go do reszty obiektu podczas serializacji. Metoda readObject najpierw dokonuje podstawowego deserializacji pól nie przejściowych z racji wykonania metody defaultReadObject. Na koniec przypisze wartość odszyfrowanego numeru PESEL do wartości pola w klasie.

Wynik działania programu:

```
Student{peselNumber=22301559573, name='Michał', surname='Młodawski',
indexNumber=92136}
Process finished with exit code 0
```

Plik z obiektem po otwarciu go w edytorze tekstu:

```
rO0ABXNyACNjb20ubW1sb2Rhd3NraS5zZXJpYWxpemFjamEuU3R1ZGVudAAAAAABMaoWA
gADTAALaW5kZXhOdW1iZXJ0ABNMamF2YS9sYW5nL0ludGVnZXI7TAAEbmFtZXQAEkxqYXZh
L2xhbmcvU3RyaW5nO0wAB3N1cm5hbWVxAH4AAnhwc3IAEWphdmEubGFuZy5JbnRlZ2VyEu
KgpPeBhzgCAAFJAAV2YWx1ZXhyABBqYXZhLmxhbmcuTnVtYmVyhqyVHQuU4IsCAAB4cAABZ
+h0AAdNaWNoYcWCdAAKTcWCb2Rhd3NraQ==
```

Po dekodowaniu przy użyciu narzędzia CyberChef:

```
¬í..sr.
com.mmlodawski.adnotacje.Student.....Ö....L..indexNumbert..Ljava/lang/Integer;L..namet..Ljava
/lang/String;L..surnameq.~..xpsr..java.lang.Integer.â ¤÷..8...I..valuexr..java.lang.Number.¬....à....x
p..gèt..MichaÅ.t.
MÅ.odawski
```

8.Haszowanie danych

Haszowanie danych to proces przetwarzania danych wejściowych przez funkcję skrótu, która generuje wartość hasza (lub skrótu) z danych wejściowych. Ta wartość hasza jest zwykle krótsza niż dane wejściowe, co ułatwia przechowywanie, porównywanie i przetwarzanie danych.

Haszowanie danych jest szeroko stosowane w aplikacjach do bezpiecznego przechowywania haseł użytkowników, unikatowych identyfikatorów sesji, lub do szybkiego wyszukiwania elementów w kolekcjach danych. Haszowanie również jest stosowane do weryfikacji integralności danych - wartość hasza przedstawia stan danych wejściowych, a jakakolwiek zmiana danych wejściowych skutkuje inną wartością hasha.

Przykładowo, w przypadku hashowania haseł użytkowników, hasz jest przechowywany w bazie danych zamiast samego hasła. Gdy użytkownik loguje się do aplikacji, jego hasło jest ponownie hashowane i porównywane z wartością hasza w bazie danych, co umożliwia potwierdzenie poprawności hasła bez przechowywania go w formie jawnej. W ten sposób bezpieczeństwo hasła jest zwiększone, ponieważ nawet jeśli baza danych zostanie naruszona, hasła nie będą bezpośrednio dostępne dla osoby atakującej.

Zastosowania haszowania

1. Bezpieczne przechowywanie haseł użytkowników
2. Generowanie unikatowych identyfikatorów sesji
3. Szybkie wyszukiwanie elementów w kolekcjach danych
4. Weryfikacja integralności danych
5. Cyfrowe podpisy
6. Blockchain i kryptowaluty

Kluczowe cechy dobrych funkcji haszujących

1. Deterministyczność: Ta sama wartość wejściowa zawsze powinna generować ten sam hasz.
2. Szybkość obliczeniowa: Funkcja powinna szybko generować hasz.
3. Efekt lawiny: Mała zmiana w danych wejściowych powinna powodować znaczącą zmianę w haszu wyjściowym.
4. Odporność na kolizje: Powinno być trudno znaleźć dwie różne wartości wejściowe, które dają ten sam hasz.
5. Nieodwracalność: Nie powinno być możliwe odtworzenie danych wejściowych z hasza.

Popularne algorytmy haszujące

1. MD5 (Message Digest 5) - obecnie uważany za niebezpieczny do zastosowań kryptograficznych
2. SHA-1 (Secure Hash Algorithm 1) - również uważany za przestarzały
3. SHA-2 (SHA-256, SHA-512) - szeroko stosowany, uważany za bezpieczny
4. SHA-3 - najnowszy standard, zaprojektowany jako alternatywa dla SHA-2
5. bcrypt, scrypt, Argon2 - specjalistyczne funkcje do haszowania haseł

Haszowanie haseł

Przy haszowaniu haseł stosuje się dodatkowe techniki:

1. Solenie: Dodawanie unikalnej, losowej wartości (soli) do hasła przed haszowaniem.
2. Peppering: Dodawanie tajnego klucza do hasła przed haszowaniem.
3. Wielokrotne haszowanie: Powtarzanie procesu haszowania wiele razy.
4. Adaptacyjne funkcje haszujące: Funkcje takie jak bcrypt, które celowo spowalniają proces haszowania.

Ataki na funkcje haszujące

1. Ataki brute-force: Próbowanie wszystkich możliwych kombinacji.
2. Ataki słownikowe: Używanie listy popularnych haseł.
3. Ataki tęczowych tablic: Używanie prekomputowanych tablic haszów.
4. Ataki kolizyjne: Poszukiwanie dwóch różnych wartości dających ten sam hasz.

W Javie można wykorzystać różne algorytmy hashowania, takie jak np. SHA-256, SHA-512, MD5, itp. Poniżej znajduje się przykład hashowania ciągu znaków za pomocą algorytmu SHA-256:

```
import java.nio.charset.StandardCharsets;
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;

public class HashExample {
    public static void main(String[] args) {
        String input = "To jest tekst, który zostanie zahaszowany";
        String hashed = hashString(input);
        System.out.println("Wynik hashowania: " + hashed);
    }

    private static String hashString(String input) {
        try {
            MessageDigest digest = MessageDigest.getInstance("SHA-256");
            byte[] hash =
digest.digest(input.getBytes(StandardCharsets.UTF_8));
            StringBuilder hexString = new StringBuilder();

            for (byte b : hash) {
                String hex = Integer.toHexString(0xff & b);
                if (hex.length() == 1) hexString.append('0');
                hexString.append(hex);
            }

            return hexString.toString();
        } catch (NoSuchAlgorithmException e) {
            e.printStackTrace();
            return null;
        }
    }
}
```

W tym przykładzie, ciąg znaków "To jest tekst, który zostanie zahaszowany" jest hashowany za pomocą algorytmu SHA-256, a wynik hashowania jest wyświetlany w konsoli.

9.Szyfrowanie danych

Szyfrowanie danych to proces przekształcania danych w taki sposób, aby stały się one nieczytelne dla osób, które nie mają uprawnionego dostępu do klucza deszyfrującego. Jest to jedna z podstawowych technik stosowanych w celu zapewnienia bezpieczeństwa danych.

Rodzaje szyfrowania danych:

1. Szyfrowanie symetryczne - polega na wykorzystaniu tego samego klucza do szyfrowania i deszyfrowania danych. Jest to szybka metoda, ale wymaga bezpiecznego sposobu udostępniania klucza.
2. Szyfrowanie asymetryczne - polega na wykorzystaniu pary kluczy: publicznego i prywatnego. Klucz publiczny jest wykorzystywany do szyfrowania danych, natomiast klucz prywatny do ich odszyfrowania. Jest to bezpieczniejsza metoda, ponieważ klucz prywatny jest chroniony i nie jest udostępniany innym użytkownikom.
3. Szyfrowanie mieszane (hybrydowe) - polega na połączeniu szyfrowania symetrycznego i asymetrycznego. Dane są szyfrowane przy użyciu klucza symetrycznego, a klucz ten jest szyfrowany przy użyciu klucza publicznego, który jest udostępniany innym użytkownikom.

W przypadku szyfrowania symetrycznego i mieszanej konieczne jest bezpieczne przechowywanie klucza, a w przypadku szyfrowania asymetrycznego klucz prywatny musi być odpowiednio zabezpieczony przed nieuprawnionym dostępem.

9.1.Szyfrowanie symetryczne z wykorzystaniem algorytmu AES

AES (Advanced Encryption Standard) to standard szyfrowania symetrycznego, który został zaakceptowany przez National Institute of Standards and Technology (NIST) w 2001 roku. W zależności od długości klucza, AES posiada trzy wersje: AES-128, AES-192 i AES-256. Oto krótki opis każdej z nich:

1. AES-128: wykorzystuje klucz o długości 128 bitów i zapewnia wysoki poziom bezpieczeństwa.
2. AES-192: wykorzystuje klucz o długości 192 bitów i oferuje wyższy poziom bezpieczeństwa niż AES-128.
3. AES-256: wykorzystuje klucz o długości 256 bitów i zapewnia najwyższy poziom bezpieczeństwa, ale wymaga większych zasobów obliczeniowych w porównaniu do AES-128 i AES-192.

Ostatecznie wybór wersji AES zależy od wymagań bezpieczeństwa i zasobów systemowych, jakie są dostępne.

```
import javax.crypto.Cipher;
import javax.crypto.KeyGenerator;
import javax.crypto.SecretKey;
import javax.crypto.spec.SecretKeySpec;
import java.nio.charset.StandardCharsets;
import java.security.Key;
import java.security.NoSuchAlgorithmException;
import java.util.Base64;

public class SymmetricEncryptionExample {

    private static final String ALGORITHM = "AES";
    private static final int KEY_SIZE = 128;

    public static void main(String[] args) {
        String plaintext = "This is a secret message.";

        // Generate a symmetric key
        SecretKey secretKey = generateKey();

        // Encrypt the plaintext
        String ciphertext = encrypt(plaintext, secretKey);

        // Decrypt the ciphertext
        String decryptedText = decrypt(ciphertext, secretKey);

        // Print the results
        System.out.println("Plaintext: " + plaintext);
        System.out.println("Ciphertext: " + ciphertext);
        System.out.println("Decrypted text: " + decryptedText);
    }

    private static SecretKey generateKey() {
        try {
            KeyGenerator keyGenerator = KeyGenerator.getInstance(ALGORITHM);
            keyGenerator.init(KEY_SIZE);
            return keyGenerator.generateKey();
        } catch (NoSuchAlgorithmException e) {
            throw new RuntimeException("Error generating AES key", e);
        }
    }

    private static String encrypt(String plaintext, Key key) {
        try {
            Cipher cipher = Cipher.getInstance(ALGORITHM);
            cipher.init(Cipher.ENCRYPT_MODE, key);
            byte[] ciphertext =
cipher.doFinal(plaintext.getBytes(StandardCharsets.UTF_8));
            return Base64.getEncoder().encodeToString(ciphertext);
        } catch (Exception e) {
            throw new RuntimeException("Error encrypting plaintext", e);
        }
    }
```

```
    private static String decrypt(String ciphertext, Key key) {
        try {
            Cipher cipher = Cipher.getInstance(ALGORITHM);
            cipher.init(Cipher.DECRYPT_MODE, key);
            byte[] decryptedText =
cipher.doFinal(Base64.getDecoder().decode(ciphertext));
            return new String(decryptedText, StandardCharsets.UTF_8);
        } catch (Exception e) {
            throw new RuntimeException("Error decrypting ciphertext", e);
        }
    }
}
```

W tym przykładzie najpierw generujemy klucz symetryczny za pomocą **generateKey()**, a następnie używamy go do zaszyfrowania i odszyfrowania tekstu. Metoda **encrypt()** przyjmuje tekst jawnie oraz klucz, a następnie zwraca zaszyfrowany tekst jako ciąg znaków Base64. Metoda **decrypt()** przyjmuje zaszyfrowany tekst oraz klucz i zwraca odszyfrowany tekst jako ciąg znaków.

9.2.Szyfrowanie asymetryczne

Szyfrowanie asymetryczne, zwane także kryptografią klucza publicznego, to rodzaj kryptografii, w której dane są szyfrowane i deszyfrowane przy użyciu różnych kluczy - klucza publicznego i klucza prywatnego. Klucz publiczny jest dostępny publicznie i służy do szyfrowania wiadomości, natomiast klucz prywatny jest znany tylko właścicielowi i służy do odszyfrowywania wiadomości. Szyfrowanie asymetryczne umożliwia bezpieczną wymianę kluczy, a także uwierzytelnianie i podpisywanie danych. Przykłady algorytmów szyfrowania asymetrycznego to RSA, DSA czy ECC.

W tym przykładzie korzystamy z algorytmu RSA z kluczem o długości 2048 bitów oraz dostawcy JCE o nazwie "BC" (Bouncy Castle). Szyfrowanie polega na zaszyfrowaniu bajtów tekstu jawnej przy użyciu klucza publicznego. Odszyfrowanie odbywa się przy użyciu klucza prywatnego.

```
    <!-- https://mvnrepository.com/artifact/org.bouncycastle/bcprov-jdk18on
-->
    <dependency>
      <groupId>org.bouncycastle</groupId>
      <artifactId>bcprov-jdk18on</artifactId>
      <version>1.78.1</version>
    </dependency>
```

```java
import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.security.PrivateKey;
import java.security.PublicKey;
import java.security.SecureRandom;
import java.security.Security;
import java.util.Base64;

import javax.crypto.Cipher;

public class AsymmetricEncryptionExample {
    public static void main(String[] args) throws Exception {
        Security.addProvider(new
org.bouncycastle.jce.provider.BouncyCastleProvider()); // Dodanie dostawcy
JCE

        KeyPairGenerator keyPairGenerator =
KeyPairGenerator.getInstance("RSA", "BC"); // Wybór algorytmu i dostawcy
        SecureRandom secureRandom = new SecureRandom(); // Inicjalizacja
generatora liczb losowych
        keyPairGenerator.initialize(2048, secureRandom); // Inicjalizacja
generatora kluczy

        KeyPair keyPair = keyPairGenerator.generateKeyPair(); // Generowanie
pary kluczy

        // Pobranie kluczy
        PrivateKey privateKey = keyPair.getPrivate();
        PublicKey publicKey = keyPair.getPublic();

        String plainText = "Tekst do zaszyfrowania";
        byte[] plainTextBytes = plainText.getBytes();

        // Szyfrowanie
        Cipher encryptCipher = Cipher.getInstance("RSA", "BC"); // Wybór
algorytmu i dostawcy
        encryptCipher.init(Cipher.ENCRYPT_MODE, publicKey); // Inicjalizacja
szyfrowania
        byte[] encryptedBytes = encryptCipher.doFinal(plainTextBytes);

        // Konwersja zaszyfrowanych danych na postać tekstową Base64
        String encryptedBase64 =
Base64.getEncoder().encodeToString(encryptedBytes);

        System.out.println("Zaszyfrowany tekst (Base64): " +
encryptedBase64);

        // Odszyfrowanie
        byte[] encryptedBytes2 =
Base64.getDecoder().decode(encryptedBase64);

        Cipher decryptCipher = Cipher.getInstance("RSA", "BC"); // Wybór
algorytmu i dostawcy
        decryptCipher.init(Cipher.DECRYPT_MODE, privateKey); //
Inicjalizacja odszyfrowywania
        byte[] decryptedBytes = decryptCipher.doFinal(encryptedBytes2);
```

```
        String decryptedText = new String(decryptedBytes);

        System.out.println("Odszyfrowany tekst: " + decryptedText);
    }
}
```

10. Obsługa plików w języku Python

Otwieranie, zapisywanie i zamykanie plików w Pythonie jest niezwykle proste i efektywne dzięki wbudowanej funkcji open() oraz konstrukcji with. Dzięki odpowiedniej obsłudze wyjątków możemy tworzyć niezawodne programy, które radzą sobie z błędami i przypadkami nieprzewidzianymi, takimi jak brak plików czy problemy z zapisem.

Należy pamiętać, aby zawsze zamykać pliki, najlepiej przy użyciu konstrukcji with, która automatycznie zajmuje się zamykaniem plików po zakończeniu pracy.

10.1.Otwieranie plików

Otwieranie plików w Pythonie jest niezwykle proste dzięki wbudowanej funkcji open(). Funkcja ta zwraca obiekt pliku, który można użyć do odczytu, zapisu lub innych operacji na plikach.

Składnia funkcji open():

```
open(file, mode='r', buffering=-1, encoding=None, errors=None,
newline=None, closefd=True, opener=None)
```

- **file**: Ścieżka do pliku, który chcemy otworzyć.
- **mode**: Tryb otwierania pliku. Może być jednym z następujących:
    - 'r': Odczyt (domyślnie). Jeśli plik nie istnieje, zostanie zgłoszony błąd.
    - 'w': Zapis. Jeśli plik istnieje, jego zawartość zostanie nadpisana. Jeśli plik nie istnieje, zostanie utworzony.
    - 'a': Dodawanie na końcu pliku. Jeśli plik nie istnieje, zostanie utworzony.
    - 'b': Tryb binarny (np. 'rb' – odczyt w trybie binarnym).
    - 'x': Utworzenie pliku. Zgłasza błąd, jeśli plik już istnieje.
    - 't': Tryb tekstowy (domyślny).
    - '+': Otwiera plik do odczytu i zapisu (np. 'r+').

Przykład otwierania pliku w trybie odczytu:

```
file = open('plik.txt', 'r')
```

W tym przykładzie otwieramy plik plik.txt w trybie odczytu ('r'). Domyślnie pliki są otwierane w trybie tekstowym ('t'), więc każdy plik otwarty w ten sposób będzie traktowany jako tekst.

10.2.Zamykanie plików

Po zakończeniu pracy z plikiem zawsze powinniśmy go zamknąć, aby zwolnić zasoby systemowe. Plik można zamknąć za pomocą metody close():

```
file.close()
```

Jeśli nie zamkniemy pliku, system operacyjny może pozostawić otwarte uchwyty plików, co w dłuższym czasie może prowadzić do wyczerpania zasobów i błędów. Dlatego zawsze ważne jest, aby zamykać pliki po zakończeniu operacji.

10.3.Otwieranie plików za pomocą with

Python dostarcza bardziej elegancki sposób otwierania i zamykania plików, który automatycznie dba o zamknięcie pliku, nawet jeśli wystąpi wyjątek podczas operacji na pliku. Jest to konstrukcja with.

Przykład otwierania i zamykania pliku za pomocą with:

```
with open('plik.txt', 'r') as file:
    data = file.read()
    print(data)
```

W tym przykładzie plik plik.txt zostanie automatycznie zamknięty po zakończeniu bloku with, niezależnie od tego, czy operacje zakończą się sukcesem, czy nie. Korzystanie z konstrukcji with jest zalecanym sposobem pracy z plikami w Pythonie, ponieważ zapobiega problemom z niezamkniętymi plikami.

10.4.Tryby otwierania plików

Jak wspomniano wcześniej, Python obsługuje różne tryby otwierania plików w zależności od potrzeb. Oto kilka popularnych trybów i ich zastosowania:

Tryb odczytu ('r'):

Służy do odczytu pliku. Jeśli plik nie istnieje, zostanie zgłoszony błąd FileNotFoundError.

```
with open('plik.txt', 'r') as file:
    content = file.read()
    print(content)
```

Tryb zapisu ('w'):

Służy do zapisu do pliku. Jeśli plik istnieje, jego zawartość zostanie nadpisana. Jeśli plik nie istnieje, zostanie utworzony.

```
with open('plik.txt', 'w') as file:
    file.write('To jest nowa zawartość pliku.')
```

Tryb dodawania ('a'):

Służy do dodawania nowej zawartości na końcu pliku. Jeśli plik nie istnieje, zostanie utworzony.

```
with open('plik.txt', 'a') as file:
    file.write('Dodanie nowej linii.')
```

Tryb binarny ('rb', 'wb', 'ab'):

Służy do pracy z plikami binarnymi, takimi jak obrazy czy pliki audio. Odczyt i zapis odbywa się w formacie binarnym, a nie tekstowym.

```
# Odczyt pliku binarnego
with open('image.png', 'rb') as file:
    binary_content = file.read()

# Zapis pliku binarnego
with open('output.bin', 'wb') as file:
    file.write(binary_content)
```

10.5.Odczyt danych z pliku

Python oferuje wiele metod do odczytu danych z plików, w zależności od tego, jak chcemy uzyskać dostęp do zawartości pliku.

Metoda read()

Metoda ta odczytuje całą zawartość pliku jako pojedynczy ciąg znaków.

```
with open('plik.txt', 'r') as file:
    content = file.read()
    print(content)
```

Metoda readline()

Odczytuje jedną linię z pliku na raz.

```
with open('plik.txt', 'r') as file:
    line = file.readline()
    while line:
        print(line.strip())  # strip() usuwa znaki nowej linii
        line = file.readline()
```

Metoda readlines()

Odczytuje wszystkie linie z pliku i zwraca je jako listę.

```
with open('plik.txt', 'r') as file:
    line = file.readline()
    while line:
        print(line.strip())  # strip() usuwa znaki nowej linii
        line = file.readline()
```

10.6.Zapis danych do pliku

Podobnie jak z odczytem, Python oferuje różne metody do zapisywania danych do plików.

Metoda write()

Metoda write() pozwala na zapisanie danych do pliku. Uwaga: nie dodaje automatycznie znaków nowej linii (\n), więc musimy je dodać ręcznie, jeśli chcemy zapisać nową linię.

```
with open('plik.txt', 'w') as file:
    file.write('To jest nowa linia.\n')
```

Metoda writelines()

Metoda writelines() służy do zapisywania listy linii do pliku. Również nie dodaje automatycznie znaków nowej linii, więc musimy to zrobić ręcznie.

```
lines = ['Pierwsza linia\n', 'Druga linia\n', 'Trzecia linia\n']
with open('plik.txt', 'w') as file:
    file.writelines(lines)
```

10.7.Obsługa wyjątków przy pracy z plikami

Podczas pracy z plikami mogą wystąpić różne błędy, np. gdy plik nie istnieje lub gdy próbujemy zapisać do pliku, do którego nie mamy uprawnień. Ważne jest, aby te błędy odpowiednio obsługiwać.

Przykład z obsługą błędów:

```
try:
    with open('plik_nie_istnieje.txt', 'r') as file:
        content = file.read()
        print(content)
except FileNotFoundError:
    print("Plik nie został znaleziony.")
except IOError:
    print("Błąd I/O.")
finally:
    print("Operacja zakończona.")
```

W tym przykładzie, jeśli plik nie istnieje, zostanie zgłoszony wyjątek FileNotFoundError, a program wypisze komunikat "Plik nie został znaleziony". Sekcja finally zawsze się wykona, niezależnie od tego, czy wyjątek został zgłoszony, czy nie.

Obsługa błędów przy zapisie do pliku:

```
try:
    with open('plik.txt', 'w') as file:
        file.write('Nowe dane.')
except IOError:
    print("Wystąpił błąd przy zapisie do pliku.")
```

10.8.Buforowanie danych

Python obsługuje również buforowanie przy pracy z plikami. Możemy kontrolować rozmiar bufora za pomocą parametru buffering w funkcji open().

- **buffering = -1**: Używa buforowania domyślnego (systemowe ustawienia).
- **buffering = 0**: Wyłącza buforowanie (tylko dla trybu binarnego).
- **buffering > 1**: Ustawia rozmiar bufora w bajtach.

Przykład:

```
with open('plik.txt', 'w', buffering=8192) as file:  # Bufor 8 KB
    file.write('Dane z buforowaniem.')
```

11.Przykładowe operacje na plikach

Sprawdzanie, czy plik istnieje:

Python dostarcza moduł os, który pozwala na wykonywanie różnych operacji na plikach i katalogach. Możemy użyć os.path.exists(), aby sprawdzić, czy plik istnieje.

```
import os

if os.path.exists('plik.txt'):
    print("Plik istnieje")
else:
    print("Plik nie istnieje")
```

Usuwanie pliku:

```
import os

if os.path.exists('plik.txt'):
    os.remove('plik.txt')
    print("Plik został usunięty")
else:
    print("Plik nie istnieje")
```

Tworzenie katalogu:

```
import os

os.mkdir('nowy_katalog')
```

Zmienianie nazwy pliku:

```
import os

os.rename('plik_stary.txt', 'plik_nowy.txt')
```

12.Serializacja oraz deserializacja danych z wykorzystaniem języka Python

Serializacja i deserializacja to procesy, które umożliwiają przekształcanie obiektów do formatu, który można zapisać, przesłać przez sieć lub zapisać w bazie danych, a następnie przekształcenie ich z powrotem do pierwotnej postaci. W Pythonie istnieje kilka sposobów na serializację i deserializację danych, które obejmują standardowe biblioteki oraz formaty takie jak pickle, json, czy inne.

12.1.Co to jest serializacja i deserializacja?

- **Serializacja** to proces konwersji obiektu w strumień bajtów lub inną formę (np. JSON), którą można przechowywać lub przesłać. Umożliwia zapisanie stanu obiektu, aby można było go później odtworzyć.
- **Deserializacja** to proces odwrotny do serializacji. Polega na odtworzeniu obiektu z zapisanych danych (np. z pliku, strumienia danych lub przesyłki sieciowej).

Podczas deserializacji danych, szczególnie z niezaufanych źródeł, należy zachować szczególną ostrożność. Moduły takie jak pickle mogą wykonywać niebezpieczny kod w trakcie deserializacji, jeśli dane pochodzą z nieznanego źródła.

12.2.Serializacja i deserializacja z użyciem pickle

Pickle jest standardowym modułem Pythona do serializacji i deserializacji obiektów. Działa on bezpośrednio na dowolnych obiektach Pythona i pozwala na ich serializację do strumienia bajtów.

Serializacja z pickle

```
import pickle

# Tworzenie obiektu do serializacji
data = {'name': 'Michal', 'age': 25, 'city': 'Kielce'}

# Serializacja obiektu do pliku
with open('data.pkl', 'wb') as file:
    pickle.dump(data, file)
```

W tym przykładzie:

1. Tworzymy obiekt data, który jest słownikiem Pythona.
2. Serializujemy go przy użyciu funkcji pickle.dump() do pliku binarnego data.pkl.
3. Plik otwieramy w trybie binarnym zapisu ('wb').

Deserializacja z pickle

```
import pickle

# Deserializacja obiektu z pliku
with open('data.pkl', 'rb') as file:
    loaded_data = pickle.load(file)

print(loaded_data)
```

W tym przykładzie:

1. Otwieramy plik data.pkl w trybie binarnym odczytu ('rb').
2. Używamy funkcji pickle.load(), aby zdeserializować obiekt.
3. Wyświetlamy zdeserializowany słownik.

Obsługa wyjątków w pickle

Podczas serializacji i deserializacji mogą wystąpić błędy, szczególnie jeśli próbujemy deserializować uszkodzony plik. Dlatego warto zastosować obsługę wyjątków:

```
import pickle

try:
    with open('data.pkl', 'rb') as file:
        loaded_data = pickle.load(file)
        print(loaded_data)
except (FileNotFoundError, pickle.PickleError) as e:
    print(f"Wystąpił błąd: {e}")
```

12.3.Bezpieczna serializacja i deserializacja z JSON

JSON jest bezpieczniejszy niż pickle, ponieważ nie pozwala na wykonanie kodu podczas deserializacji. Dlatego zaleca się używanie JSON do wymiany danych między systemami.

json to jeden z najpopularniejszych formatów wymiany danych. Jest to tekstowy format, który może być łatwo odczytywany przez człowieka. W Pythonie standardowa biblioteka json obsługuje serializację i deserializację do i z formatu JSON.

Serializacja do JSON

```
import json

# Tworzenie obiektu do serializacji
data = {'name': 'Paulina', 'age': 24, 'city': 'Krakow'}

# Serializacja obiektu do pliku JSON
with open('data.json', 'w') as file:
    json.dump(data, file, indent=4)
```

W tym przykładzie:

1. Serializujemy obiekt do pliku data.json w formacie tekstowym JSON.
2. Używamy json.dump(), aby zapisać dane. Parametr indent=4 ustawia wcięcia w pliku JSON, co ułatwia jego czytanie.

Deserializacja z JSON

```
import json

# Deserializacja obiektu z pliku JSON
with open('data.json', 'r') as file:
    loaded_data = json.load(file)

print(loaded_data)
```

W tym przykładzie:

1. Otwieramy plik data.json w trybie odczytu ('r').
2. Używamy json.load(), aby zdeserializować obiekt z pliku.

Serializacja do łańcucha znaków JSON

Możemy również serializować dane bezpośrednio do ciągu znaków:

```
import json

data = {'name': 'Pulina', 'age': 24, 'city': 'Krakow'}
json_string = json.dumps(data, indent=4)

print(json_string)
```

Funkcja json.dumps() serializuje obiekt do formatu JSON i zwraca go jako ciąg znaków.

Deserializacja z łańcucha znaków JSON

```
import json

json_string = '{"name": "Michal", "age": 25, "city": "Kielce"}'
data = json.loads(json_string)

print(data)
```

Obsługa obiektów niestandardowych w JSON

JSON nie obsługuje bezpośrednio typów specyficznych dla Pythona, takich jak obiekty klas. Możemy jednak dostosować serializację i deserializację, aby obsługiwały te typy.

Serializacja niestandardowych obiektów

```
import json


class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

# Serializacja obiektu klasy Person
def serialize_person(obj):
    if isinstance(obj, Person):
        return {'name': obj.name, 'age': obj.age}
    raise TypeError(f"Object of type {obj.__class__.__name__} is not
serializable")

person = Person("Paulina", 24)
person_json = json.dumps(person, default=serialize_person, indent=4)
print(person_json)
```

Deserializacja niestandardowych obiektów

Aby zdeserializować obiekt niestandardowy, możemy użyć niestandardowej funkcji dekodującej:

```
import json

class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

# Deserializacja obiektu klasy Person
def deserialize_person(dct):
    return Person(dct['name'], dct['age'])

json_string = '{"name": "Paulina", "age": 24}'
person = json.loads(json_string, object_hook=deserialize_person)

print(f"Imię: {person.name}, Wiek: {person.age}")
```

13.Serializacja obiektów do formatu binarnego

Czasami potrzebujemy serializować dane do formatu binarnego, np. w celu przesłania danych przez sieć lub zapisania do pliku w celu szybkiego odczytu.

Serializacja binarna za pomocą pickle

Już wcześniej pokazaliśmy, jak pickle serializuje dane do formatu binarnego. Poniżej jest przykład serializacji i deserializacji obiektów klas użytkownika za pomocą pickle:

```
import pickle

class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

person = Person("Michal", 25)

# Serializacja obiektu do pliku binarnego
with open('person.pkl', 'wb') as file:
    pickle.dump(person, file)

# Deserializacja obiektu z pliku binarnego
with open('person.pkl', 'rb') as file:
    loaded_person = pickle.load(file)

print(f"Imię: {loaded_person.name}, Wiek: {loaded_person.age}")
```