

# WYJĄTKI

- Obsługa i rzucanie wyjątków (powtórka)
- Hierarchia wyjątków i jej konsekwencje

**Wykład częściowo oparty na materiałach A. Jaskot**

# WYJĄTKI (ang. Exceptions)

*Niechciane, nieoczekiwane zdarzenia występujące podczas wykonywania programu, które zakłócają jego normalne działanie*

Częste źródła wyjątków:

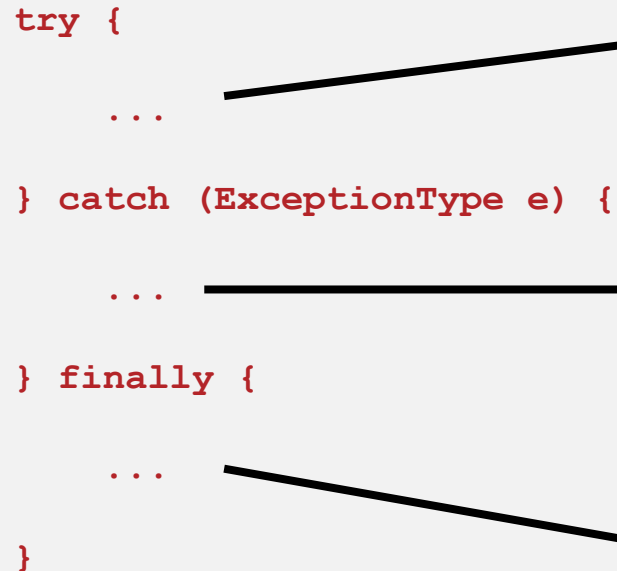
- Błędy w kodzie
- Nieprawidłowe dane wejściowe
- Awaria sprzętu
- Utrata połączenia sieciowego
- Brak dostępu do pliku lub błędna ścieżka

Część języków, np. Python, używa wyjątków do kontroli przepływu sterowania w programach

Inne języki, w tym Java, używają wyjątków tylko do sygnalizowania nieprzewidzianych, błędnych warunków

# OBSŁUGA WYJĄTKÓW

```
try {  
    ...  
} catch (ExceptionType e) {  
    ...  
} finally {  
    ...  
}
```



The diagram shows a code block on the left enclosed in a dashed box. Three arrows originate from this block and point to the right. The first arrow starts from the 'try {' line and points to the 'BLOK TRY' section. The second arrow starts from the 'catch (ExceptionType e) {' line and points to the 'BLOK CATCH' section. The third arrow starts from the 'finally {' line and points to the 'BLOK FINALLY' section.

## BLOK TRY

- zawiera kod, który może wygenerować wyjątek

## BLOK CATCH

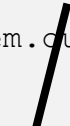
- wskazuje, jaki typ wyjątku ma być przechwytywany
- zawiera kod do obsługi wyjątku

## BLOK FINALLY

- zawiera kod wykonywany zawsze, nawet jeżeli nie pojawi się wyjątek
- zwykle używany do zwalniania zasobów

# OBSŁUGA WYJĄTKÓW


```
try {  
    int result = 2 / 0;  
    System.out.println("Result: " + result);  
} catch (ArithmeticException e) {  
    System.out.println("Caught exception: " + e);  
}
```



## ArithmeticException

występuje przy próbie wykonania niedozwolonych operacji matematycznych (najczęściej dzielenie całkowitoliczbowe przez 0)

```
try {  
    int[] numbers = new int[]{1, 3, 7, 29};  
  
    Scanner sc = new Scanner(System.in);  
    int selectedIndex = sc.nextInt();  
  
    System.out.println("Selected number: "  
        + numbers[selectedIndex]);  
} catch (ArrayIndexOutOfBoundsException e) {  
    System.out.println("Caught exception: " + e);  
}
```



## ArrayIndexOutOfBoundsException

występuje przy próbie dostępu do nieistniejącego elementu tablicy; analogicznie StringIndexOutOfBoundsException

# OBSŁUGA WYJĄTKÓW

```
try {  
    // some code that may generate exceptions  
} catch (InputMismatchException e) {  
} catch (ArrayIndexOutOfBoundsException e) {  
}
```

```
try {  
    // some code that may generate exceptions  
} catch (InputMismatchException  
        | ArrayIndexOutOfBoundsException e) {  
}
```

- konstrukcja try-catch może zawierać kilka bloków catch do obsługi różnych typów wyjątków

Od Javy 7:

- jeden blok catch może obsługiwać więcej niż jeden typ wyjątku
- typy wyjątków powinny być oddzielone znakiem |

# RZUCANIE WYJĄTKÓW

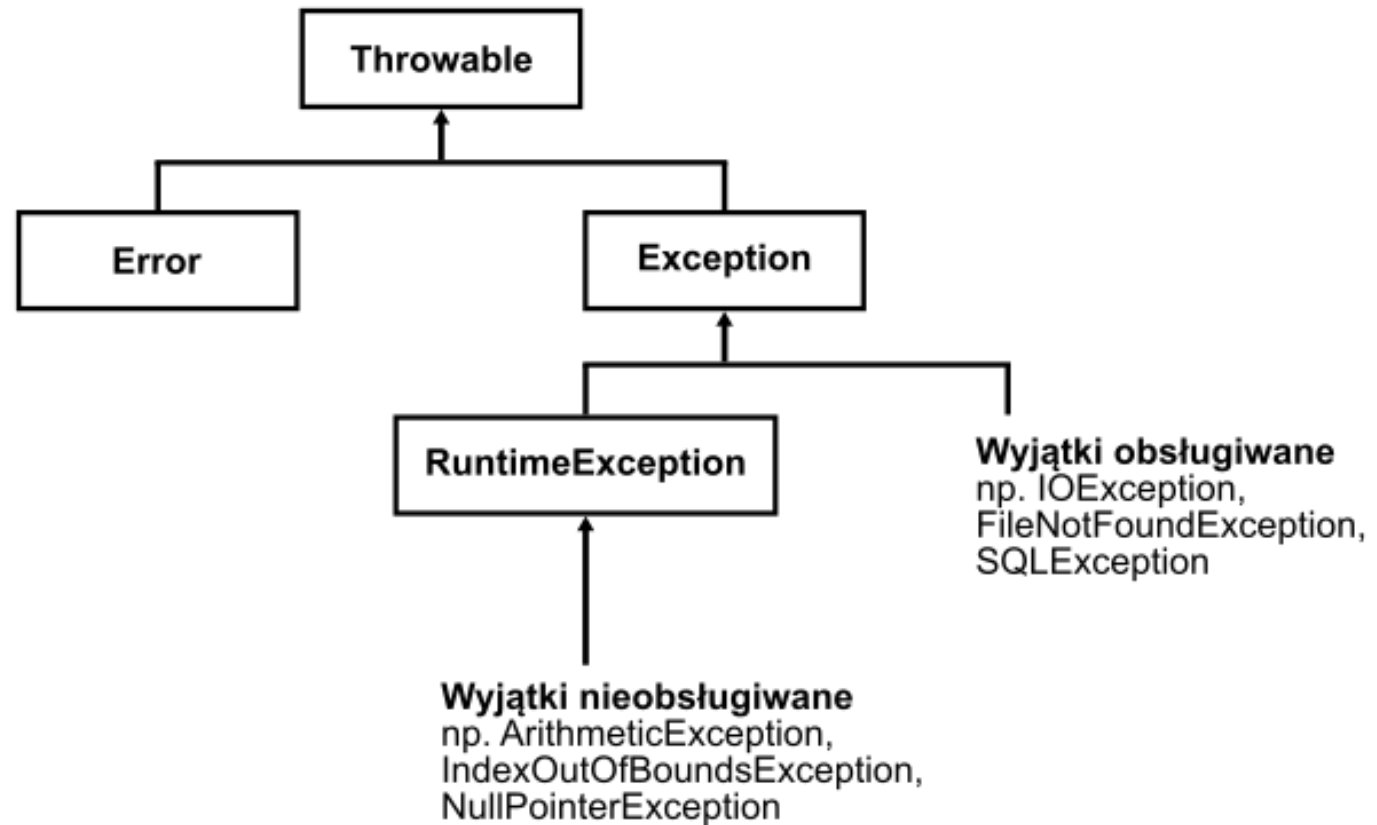
```
public class Calculator {  
    public double divide(int a, int b) {  
        if (b == 0) {  
            throw new ArithmeticException  
                ("Illegal division by 0.");  
        } else {  
            return a / b;  
        }  
    }  
}
```

```
try {  
    Calculator c = new Calculator();  
    System.out.println(c.divide(2, 0));  
} catch (ArithmeticException e) {  
    System.out.println("Caught exception: " + e);  
}
```

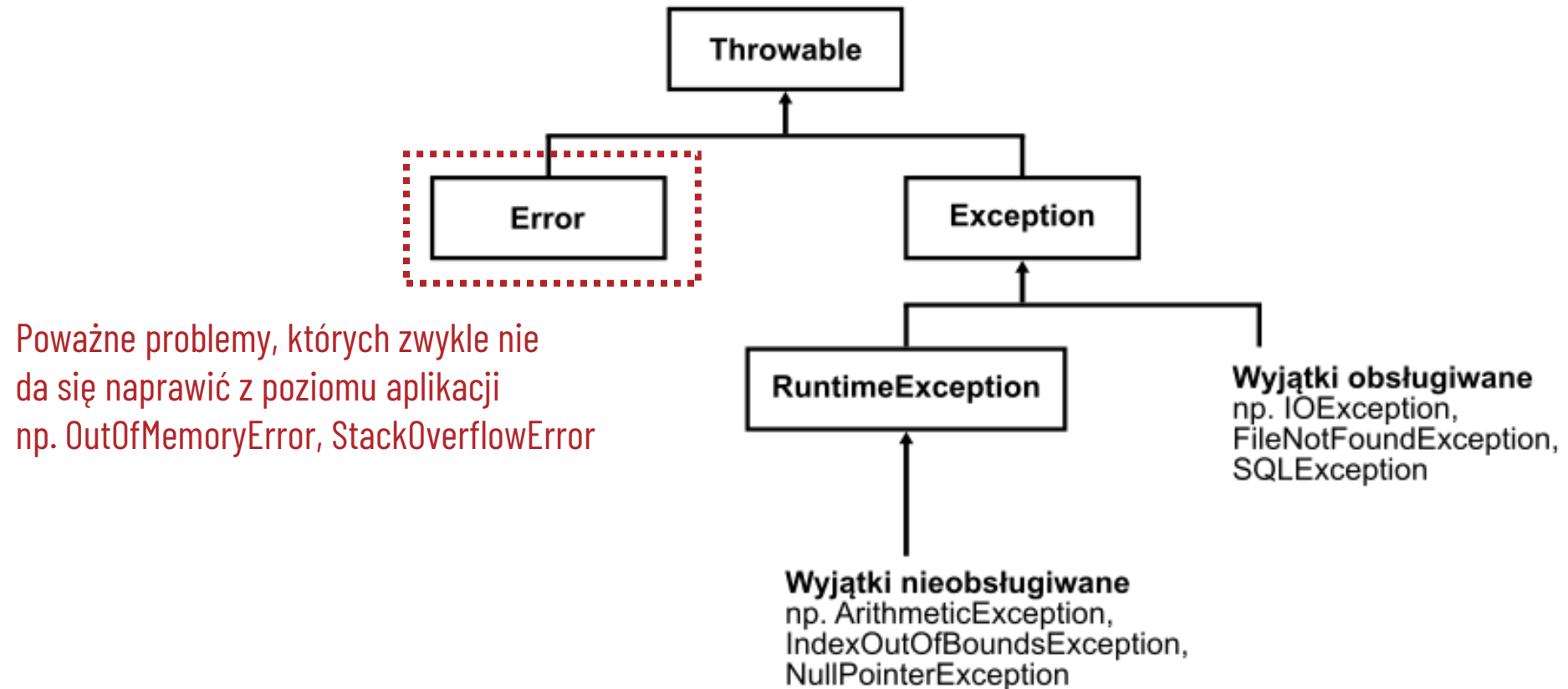
## RZUCANIE WYJĄTKÓW

- wykorzystuje słowo kluczowe *throw*
- polega na stworzeniu obiektu wyjątku odpowiedniej klasy (przy wyjątkach rzucanych z konstruktorów najczęściej *IllegalArgumentException*)
- rzucony wyjątek może być obsługiwany w dalszych częściach kodu

# HIERARCHIA WYJĄTKÓW

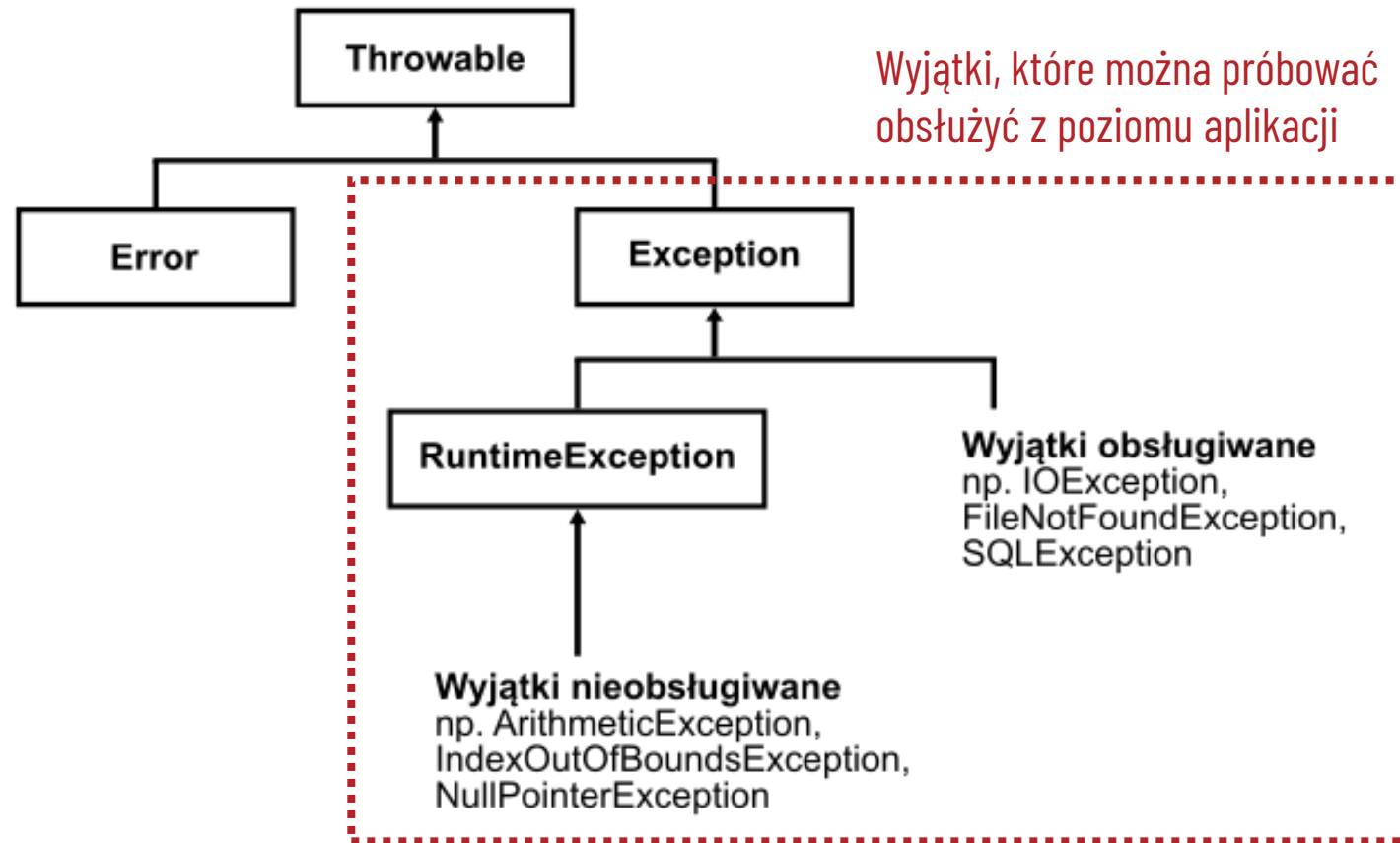


# HIERARCHIA WYJĄTKÓW





# HIERARCHIA WYJĄTKÓW



# WYJĄTKI NIEOBSŁUGIWANE (ang. Unchecked exceptions)


Także: *niekontrolowane*

- wyjątki dziedziczące po klasie *RuntimeException*
- pojawiają się na etapie wykonania programu
- mogą, ale nie muszą być obsługiwane
- zwykle można im zapobiec, dodając zabezpieczenia w kodzie

np. `ArithmeticException`, `IndexOutOfBoundsException`,  
`NullPointerException`, `IllegalArgumentException`

# WYJĄTKI NIEOBSŁUGIWANE (ang. Unchecked exceptions)

```
public static void main(String[] args) {  
  
    System.out.println(myMethod());  
  
}  
  
public static int myMethod() {  
    Scanner sc = new Scanner(System.in);  
    String inputStr = sc.nextLine();  
  
    int inputInt = Integer.parseInt(inputStr);  
    return inputInt;  
}
```



Może rzucić wyjątek `NumberFormatException`,  
jeżeli danego łańcucha nie da się  
przekonwertować na liczbę całkowitą

# WYJĄTKI NIEOBSŁUGIWANE (ang. Unchecked exceptions)

```
public static void main(String[] args) {  
  
    System.out.println(myMethod());  
}  
  
public static int myMethod() {  
    Scanner sc = new Scanner(System.in);  
    String inputStr = sc.nextLine();  
  
    int inputInt = Integer.parseInt(inputStr);  
    return inputInt;  
}
```



Może rzucić wyjątek `NumberFormatException`,  
jeżeli danego łańcucha nie da się  
przekonwertować na liczbę całkowitą

```
public static void main(String[] args) {  
  
    try {  
        System.out.println(myMethod());  
    } catch (NumberFormatException e) {  
        System.out.println  
            ("Could not convert to number.");  
    }  
}  
  
public static int myMethod() {  
    Scanner sc = new Scanner(System.in);  
    String inputStr = sc.nextLine();  
  
    int inputInt = Integer.parseInt(inputStr);  
    return inputInt;  
}
```

# WYJĄTKI NIEOBSŁUGIWANE (ang. Unchecked exceptions)

```
String str = ... // some String that can be null  
int length = str.length() ;
```



Może rzucić wyjątek `NullPointerException`,  
jeżeli zmienna `str` będzie nullem

# WYJĄTKI NIEOBSŁUGIWANE (ang. Unchecked exceptions)

```
String str = ... // some String that can be null

if (str != null) {

    int length = str.length();
    System.out.println("String length: " + length);

} else {

    System.out.println("Invalid input: string is null.");

}
```

## PODEJŚCIE LBYL

z ang. Look Before You Leap

- zakłada sprawdzenie poprawności danych i warunków przed wykonaniem operacji
- unika stosowania wyjątków do sterowania przepływem w programie

# WYJĄTKI NIEOBSŁUGIWANE (ang. Unchecked exceptions)

```
String str = ... // some String that can be null

try {
    int length = str.length();
    System.out.println("String length: " + length);
} catch (NullPointerException e) {
    System.out.println("Invalid input: string is null.");
}
```

## PODEJŚCIE EAFP

**z ang. Easier to Ask Forgiveness than Permission**

- zakłada próbę wykonania operacji bez sprawdzania danych i warunków koniecznych do jej poprawnego wykonania
- opiera się na przechwytywaniu wyjątków (które mogą, ale nie muszą się pojawić)

# WYJĄTKI OBSŁUGIWANE (ang. Checked exceptions)

Także: *kontrolowane*

- wyjątki dziedziczące po klasie *Exception*
- są wykrywane na etapie kompilacji
- muszą być obsługiwane lub deklarowane za pomocą klauzuli *throws*
- zwykle są związane z problemami nie do rozwiązania w kodzie, np. utrata połączenia, uszkodzony plik

np. `IOException`, `FileNotFoundException`,  
`EOFException`, `ParseException`



# WYJĄTKI OBSŁUGIWANE (ang. Checked exceptions)

```
public static void main(String[] args) {  
  
    FileReader file = new FileReader("someFile.txt");  
  
    BufferedReader fileInput = new BufferedReader(file);  
    for (int counter = 0; counter < 3; counter++) {  
        System.out.println(fileInput.readLine());  
    }  
  
    fileInput.close();  
}
```

`new FileReader( fileName: "someFile.txt");`

Unhandled exception: java.io.FileNotFoundException

[Add exception to method signature](#) Alt+Shift+Enter [More actions...](#) Alt+Enter

`readLine();`

Unhandled exception: java.io.IOException

[Add exception to method signature](#) Alt+Shift+Enter [More actions...](#) Alt+Enter

`close();`

Unhandled exception: java.io.IOException

[Add exception to method signature](#) Alt+Shift+Enter [More actions...](#) Alt+Enter

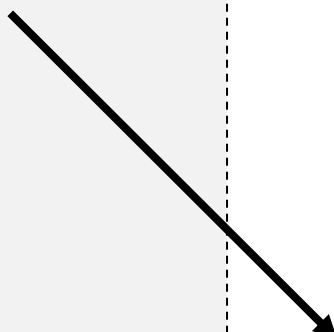
# WYJĄTKI OBSŁUGIWANE (ang. Checked exceptions)

```
public static void main(String[] args) {  
  
    try {  
        FileReader file = new FileReader("someFile.txt");  
  
        BufferedReader fileInput = new BufferedReader(file);  
        for (int counter = 0; counter < 3; counter++) {  
            System.out.println(fileInput.readLine());  
        }  
  
        fileInput.close();  
  
    } catch (FileNotFoundException e) {  
        System.out.println("File not found: " + e.getMessage());  
    } catch (IOException e) {  
        System.out.println("I/O exception: " + e.getMessage());  
    }  
}
```

- wyjątki obsługiwane muszą być wprost obsługowane w kodzie...

# WYJĄTKI OBSŁUGIWANE (ang. Checked exceptions)

```
public static void main(String[] args) throws IOException {  
  
    FileReader file = new FileReader("someFile.txt");  
  
    BufferedReader fileInput = new BufferedReader(file);  
    for (int counter = 0; counter < 3; counter++) {  
        System.out.println(fileInput.readLine());  
    }  
  
    fileInput.close();  
}
```



- ... albo zadeklarowane w sygnaturze metody za pomocą klauzuli *throws*

Uwaga: FileNotFoundException dziedziczy po IOException, więc wystarczy zadeklarować "ogólniejszy" typ wyjątku

```
throws FileNotFoundException, IOException {
```

There is a more general exception, 'java.io.IOException', in the throws list already.

Remove 'FileNotFoundException' from 'main()' throws list Alt+Shift+Enter Mor

# DEFINIOWANIE WŁASNYCH WYJĄTKÓW

- własne wyjątki nieobsługiwane dziedziczą po klasie *RuntimeException*

```
public class PaymentFailedException extends RuntimeException {  
    public PaymentFailedException(String message) {  
        super(message);  
    }  
}
```

We własnym kodzie zalecane jest tworzenie wyjątków nieobsługiwanych

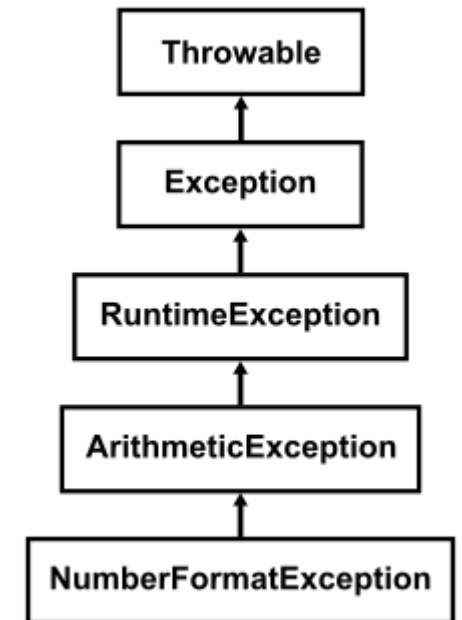
- własne wyjątki obsługiwane dziedziczą po klasie *Exception*

```
public class FileProcessingException extends Exception {  
    public FileProcessingException(String message) {  
        super(message);  
    }  
}
```

# HIERARCHIA A PRZECHWYTYWANIE WYJĄTKÓW

```
public static void main(String[] args) {  
    Scanner sc = new Scanner(System.in);  
  
    try {  
        String strToParse = sc.nextLine();  
        int parsedInt = Integer.parseInt(strToParse);  
    } catch (NumberFormatException e) {  
        System.out.println  
            ("NumberFormatException caught: " + e.getMessage());  
    } catch (ArithmeticException e) {  
        System.out.println  
            ("ArithmeticException caught: " + e.getMessage());  
    } catch (Exception e) {  
        System.out.println  
            ("Generic Exception caught: " + e.getMessage());  
    }  
  
    // some other code  
}
```

Bloki catch należy zawsze ustawiać w kolejności od najbardziej "szczegółowej" do najbardziej "ogólnej" klasy wyjątku



# STACK TRACE I PRZERZUCANIE WYJĄTKÓW

```
public class Calculator {  
    public double divide(int a, int b) {  
        if (b == 0) {  
            throw new ArithmeticException("Illegal division by 0.");  
        } else {  
            return a / b;  
        }  
    }  
}
```

```
public class Main {  
  
    public static void main(String[] args) {  
        Calculator c = new Calculator();  
        divisionTest(c);  
    }  
  
    public static void divisionTest(Calculator c) {  
        c.divide(2, 0);  
    }  
}
```

```
Exception in thread "main" java.lang.ArithmeticException Create breakpoint : Illegal division by 0.  
    at lecture.Calculator.divide(Calculator.java:6)  
    at lecture.Main.divisionTest(Main.java:18)  
    at lecture.Main.main(Main.java:13)
```

```
Process finished with exit code 1
```

# STACK TRACE I PRZERZUCANIE WYJĄTKÓW

```
public class Calculator {  
    public double divide(int a, int b) {  
        if (b == 0) {  
            throw new ArithmeticException("Illegal division by 0.");  
        } else {  
            return a / b;  
        }  
    }  
}
```

```
public class Main {  
  
    public static void main(String[] args) {  
        Calculator c = new Calculator();  
        divisionTest(c);  
    }  
  
    public static void divisionTest(Calculator c) {  
        try {  
            c.divide(2, 0);  
        } catch (ArithmeticException e) {  
            // empty catch block  
        }  
    }  
}
```

Pusty blok catch powoduje  
"zgubienie" wyjątku

```
Process finished with exit code 0
```

# STACK TRACE I PRZERZUCANIE WYJĄTKÓW

```
public class Calculator {  
    public double divide(int a, int b) {  
        if (b == 0) {  
            throw new ArithmeticException("Illegal division by 0.");  
        } else {  
            return a / b;  
        }  
    }  
}
```

```
public class Main {  
  
    public static void main(String[] args) {  
        Calculator c = new Calculator();  
        divisionTest(c);  
    }  
  
    public static void divisionTest(Calculator c) {  
        try {  
            c.divide(2, 0);  
        } catch (ArithmeticException e) {  
            throw e;  
        }  
    }  
}
```

Jeżeli nie możemy obsłużyć wyjątku w danej metodzie, powinniśmy "przerzucić" go wyżej, aż do miejsca, gdzie będzie go można sensownie obsłużyć