Podstawy języka JAVA

Wykład nr 4 – Napisy w języku Java. Wykorzystanie klasy String oraz StringBuilder. Wyjątki w języku Java.

Wyższa Szkoła Informatyki Stosowanej i Zarządzania
pod auspicjami Polskiej Akademii Nauk

Java - napisy

- Łańcuchy znaków w Java składają się z symboli Unicode.
- W Java nie ma wbudowanego typu String.
- Standardowa biblioteka Java zawiera natomiast predefiniowaną klasę o nazwie String.
- W Java każdy łańcuch w cudzysłowach jest reprezentantem klasy String.

```
public static void main(String[] args){
    String napis="To jest napis String";
    String pusty_napis="";
    System.out.println(napis);
    System.out.println(pusty_napis);
}
```

Java - podłańcuchy

- W celu uzyskania podłańcucha z łańcucha należy wykorzystać metodę **substring** klasy String.
- Metoda **substring** może przyjmować dwa parametry – indeks pierwszego znaku oraz indeks ostatniego znaku, który nie jest pobierany do podłańcucha.
- Druga implementacja metody **substring** przyjmuje tylko jeden parametr – indeks pierwszego znaku.

```
public static void main(String[] args){
    String napis="To jest napis String";
    String pod_napis = napis.substring(0, 6);
    String pod_napis_drugi = napis.substring(6);
    System.out.println(pod_napis);
    System.out.println(pod_napis_drugi);
}
```

```
To jes
t napis String
```

Java – konkatenacja łańcuchów

- W Java łańcuchy można łączyć (**konkatenować**) za pomocą znaku „+”.
- Znak „+” łączy dwa łańcuchy w takiej kolejności jak zostały wprowadzone, nie dokonując w nich żadnych zmian.
- Konkatenacja znaków bardzo często wykorzystywana jest przy wypisywaniu wartości do konsoli.

```
public static void main(String[] args){
    String napis="To jest";
    String napis2=" napis w Ja";
    String napis3="va czyli klasa String";
    String napis_calosc=napis+napis2+napis3;
    System.out.println(napis_calosc);
    System.out.println(napis+napis2+napis3);
}
```

```
To jest napis w Java czyli klasa String
To jest napis w Java czyli klasa String
```

Java – konkatenacja łańcuchów

- Kiedy trzeba połączyć symbole oddzielając je określonym symbole można wykorzystać znak „+” lub metodę **join** klasy String.
- Metoda join jako pierwszy parametr przyjmuje symbol przy pomocy którego wykonana zostanie konkatenacja.
- Kolejne parametry metody **join** to obiekty String, które zostaną połączone.

```
To jest\ napis w Ja\va czyli klasa String
To jest\ napis w Ja\va czyli klasa String
```

```
public static void main(String[] args){
    String napis="To jest";
    String napis2=" napis w Ja";
    String napis3="va czyli klasa String";
    String napis_calosc=napis+"\\"+napis2+"\\"+napis3;
    String napis_calosc_metoda = String.join("\\", napis,napis2,napis3);
    System.out.println(napis_calosc);
    System.out.println(napis_calosc_metoda);
}
```

Java – łańcuchy

- W klasie String brakuje metody, która umożliwia zmianę znaków w łańcuchu.
- Obiekty klasy String w dokumentacji Java określane są jako **niezmienialne (ang. Immutable**).
- W Java można jednak wykorzystać metodę **substring**, aby przypisać nową wartość do łańcucha (po jej wykonaniu obiekt klasy String będzie wskazywał na nową wartość w pamięci, a nie zmieniony łańcuch).

```
public static void main(String[] args){
    String napis="To jest";
    napis = napis.substring(0)+" napis w Java";
    System.out.println(napis);
}
```

```
To jest napis w Java
```

Java – porównywanie łańcuchów

- Do porównywania łańcuch w Java wykorzystuje się metodę **equals**.
- Wyrażenie **s.equals(t)** zwróci wartość true jeżeli łańcuchy s i t są identyczne, a **false** w przeciwnym przypadku.
- Łańcuchy s i t mogą być zarówno zmiennymi łańcuchowymi jak i stałymi łańcuchowymi.

```
public static void main(String[] args){
    String s="To jest napis";
    String t="To jest napis";
    System.out.println("1. Czy łańcuchy są takie same: "+s.equals(t));
    System.out.println("2. Czy łańcuchy są takie same: "+s.equals("To jest napis"));
    System.out.println("3. Czy łańcuchy są takie same: "+"".equals(s));
    System.out.println("4. Czy łańcuchy są takie same: "+"TO JEST NAPIS".equals(s));
}
```

```
1. Czy łańcuchy są takie same: true
2. Czy łańcuchy są takie same: true
3. Czy łańcuchy są takie same: false
4. Czy łańcuchy są takie same: false
```

Java – porównywanie łańcuchów

- Java umożliwia też porównywania łańcuchów z pominięciem wielkości znaków. W tym celu należy wykorzystać metodę **equalsIgnoreCase** klasy String.
- Do porównywania łańcuchów nie należy wykorzystywać operatora ==.
- Za jego pomocą można tylko sprawdzić czy dwa łańcuchy przechowywane są w tej samej lokalizacji.

```
public static void main(String[] args){
    String s="To jest napis";
    System.out.println("1. łańcuchy są takie same: "+"TO JEST NAPIS".equalsIgnoreCase(s));
    System.out.println("2. łańcuchy są takie same: "+("To jest napis"==s));
}
```

```
1. łańcuchy są takie same: true
2. łańcuchy są takie same: true
```

Java – łańcuchu puste i łańcuchy null

- Pusty łańcuch w Java to łańcuch o długości zerowej.
- W celu sprawdzenia czy łańcuch jest pusty można wykorzystać następujące konstrukcje:

```
String s="";
if(s.length()==0) System.out.println("Pusty");
if(s.equals("")) System.out.println("Pusty");
```

- Łańcuch w Java może również być wartością typu null – oznacza, że aktualnie ze zmienną nie jest powiązany żaden obiekt.
- W celu sprawdzeni czy łańcuch jest wartością null można wykorzystać następującą konstrukcję:

```
String s=null;
if(s==null) System.out.println("NULL");
```

Java – współrzędne kodowe

- Łańcuchy w Java są ciągami wartości typu char.
- Natomiast typ char jest jednostką kodową reprezentującą współrzędne kodowe znaków **Unicode** w systemie UTF-16.
- Metoda **length** klasy String zwraca liczbę jednostek kodowych Unicode w łańcuchu.
- Metoda **charAt** klasy String zwraca jednostkę kodową o zadanym indeksie z łańcucha znaków.

```
String napis="To jest napis";
System.out.println("Długośc napisu wynosi: "+napis.length());
System.out.println("Znak o indeksie 10: "+napis.charAt(10));
```

```
Długośc napisu wynosi: 13
Znak o indeksie 10: p
```

Java – współrzędne kodowe

Constructor Summary

Constructors

Constructor and Description
`String()` Initializes a newly created `String` object so that it represents an empty character sequence.
`String(byte[] bytes)` Constructs a new `String` by decoding the specified array of bytes using the platform's default charset.
`String(byte[] bytes, Charset charset)` Constructs a new `String` by decoding the specified array of bytes using the specified **charset**.
`String(byte[] ascii, int hibyte)` **Deprecated.** *This method does not properly convert bytes into characters. As of JDK 1.1, the preferred way to do this is via the `String` constructors that take a `Charset`, charset name, or that use the platform's default charset.*
`String(byte[] bytes, int offset, int length)` Constructs a new `String` by decoding the specified subarray of bytes using the platform's default charset.
`String(byte[] bytes, int offset, int length, Charset charset)` Constructs a new `String` by decoding the specified subarray of bytes using the specified **charset**.
`String(byte[] ascii, int hibyte, int offset, int count)` **Deprecated.** *This method does not properly convert bytes into characters. As of JDK 1.1, the preferred way to do this is via the `String` constructors that take a `Charset`, charset name, or that use the platform's default charset.*
`String(byte[] bytes, int offset, int length, String charsetName)` Constructs a new `String` by decoding the specified subarray of bytes using the specified charset.
`String(byte[] bytes, String charsetName)` Constructs a new `String` by decoding the specified array of bytes using the specified **charset**.

Java – współrzędne kodowe

`String(char[] value)` Allocates a new `String` so that it represents the sequence of characters currently contained in the character array argument.
`String(char[] value, int offset, int count)` Allocates a new `String` that contains characters from a subarray of the character array argument.
`String(int[] codePoints, int offset, int count)` Allocates a new `String` that contains characters from a subarray of the **Unicode code point** array argument.
`String(String original)` Initializes a newly created `String` object so that it represents the same sequence of characters as the argument; in other words, the newly created string is a copy of the argument string.
`String(StringBuffer buffer)` Allocates a new string that contains the sequence of characters currently contained in the string buffer argument.
`String(StringBuilder builder)` Allocates a new string that contains the sequence of characters currently contained in the string builder argument.

Java – współrzędne kodowe

Method Summary

Methods

Modifier and Type	Method and Description
char	`charAt(int index)` Returns the char value at the specified index.
int	`codePointAt(int index)` Returns the character (Unicode code point) at the specified index.
int	`codePointBefore(int index)` Returns the character (Unicode code point) before the specified index.
int	`codePointCount(int beginIndex, int endIndex)` Returns the number of Unicode code points in the specified text range of this String.
int	`compareTo(String anotherString)` Compares two strings lexicographically.
int	`compareToIgnoreCase(String str)` Compares two strings lexicographically, ignoring case differences.
String	`concat(String str)` Concatenates the specified string to the end of this string.
boolean	`contains(CharSequence s)` Returns true if and only if this string contains the specified sequence of char values.
boolean	`contentEquals(CharSequence cs)` Compares this string to the specified CharSequence.
boolean	`contentEquals(StringBuffer sb)` Compares this string to the specified StringBuffer.
static String	`copyValueOf(char[] data)` Returns a String that represents the character sequence in the array specified.

Java – współrzędne kodowe

static String	copyValueOf(char[] data, int offset, int count) Returns a String that represents the character sequence in the array specified.
boolean	endsWith(String suffix) Tests if this string ends with the specified suffix.
boolean	equals(Object anObject) Compares this string to the specified object.
boolean	equalsIgnoreCase(String anotherString) Compares this String to another String, ignoring case considerations.
static String	format(Locale l, String format, Object... args) Returns a formatted string using the specified locale, format string, and arguments.
static String	format(String format, Object... args) Returns a formatted string using the specified format string and arguments.
byte[]	getBytes() Encodes this String into a sequence of bytes using the platform's default charset, storing the result into a new byte array.
byte[]	getBytes(Charset charset) Encodes this String into a sequence of bytes using the given charset, storing the result into a new byte array.
void	getBytes(int srcBegin, int srcEnd, byte[] dst, int dstBegin) **Deprecated.** *This method does not properly convert characters into bytes. As of JDK 1.1, the preferred way to do this is via the getBytes() method, which uses the platform's default charset.*
byte[]	getBytes(String charsetName) Encodes this String into a sequence of bytes using the named charset, storing the result into a new byte array.
void	getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin) Copies characters from this string into the destination character array.
int	hashCode() Returns a hash code for this string.
int	indexOf(int ch) Returns the index within this string of the first occurrence of the specified character.

Java – współrzędne kodowe

int	**indexOf**(int ch, int fromIndex) Returns the index within this string of the first occurrence of the specified character, starting the search at the specified index.
int	**indexOf**(String str) Returns the index within this string of the first occurrence of the specified substring.
int	**indexOf**(String str, int fromIndex) Returns the index within this string of the first occurrence of the specified substring, starting at the specified index.
String	**intern**() Returns a canonical representation for the string object.
boolean	**isEmpty**() Returns true if, and only if, **length()** is 0.
int	**lastIndexOf**(int ch) Returns the index within this string of the last occurrence of the specified character.
int	**lastIndexOf**(int ch, int fromIndex) Returns the index within this string of the last occurrence of the specified character, searching backward starting at the specified index.
int	**lastIndexOf**(String str) Returns the index within this string of the last occurrence of the specified substring.
int	**lastIndexOf**(String str, int fromIndex) Returns the index within this string of the last occurrence of the specified substring, searching backward starting at the specified index.
int	**length**() Returns the length of this string.
boolean	**matches**(String regex) Tells whether or not this string matches the given **regular expression**.
int	**offsetByCodePoints**(int index, int codePointOffset) Returns the index within this String that is offset from the given index by codePointOffset code points.
boolean	**regionMatches**(boolean ignoreCase, int toffset, String other, int ooffset, int len) Tests if two string regions are equal.
boolean	**regionMatches**(int toffset, String other, int ooffset, int len) Tests if two string regions are equal.

Java – współrzędne kodowe

`String`	**`replace`**`(char oldChar, char newChar)` Returns a new string resulting from replacing all occurrences of `oldChar` in this string with `newChar`.
`String`	**`replace`**`(CharSequence target, CharSequence replacement)` Replaces each substring of this string that matches the literal target sequence with the specified literal replacement sequence.
`String`	**`replaceAll`**`(String regex, String replacement)` Replaces each substring of this string that matches the given **regular expression** with the given replacement.
`String`	**`replaceFirst`**`(String regex, String replacement)` Replaces the first substring of this string that matches the given **regular expression** with the given replacement.
`String[]`	**`split`**`(String regex)` Splits this string around matches of the given **regular expression**.
`String[]`	**`split`**`(String regex, int limit)` Splits this string around matches of the given **regular expression**.
`boolean`	**`startsWith`**`(String prefix)` Tests if this string starts with the specified prefix.
`boolean`	**`startsWith`**`(String prefix, int toffset)` Tests if the substring of this string beginning at the specified index starts with the specified prefix.
`CharSequence`	**`subSequence`**`(int beginIndex, int endIndex)` Returns a new character sequence that is a subsequence of this sequence.
`String`	**`substring`**`(int beginIndex)` Returns a new string that is a substring of this string.
`String`	**`substring`**`(int beginIndex, int endIndex)` Returns a new string that is a substring of this string.
`char[]`	**`toCharArray`**`()` Converts this string to a new character array.
`String`	**`toLowerCase`**`()` Converts all of the characters in this `String` to lower case using the rules of the default locale.
`String`	**`toLowerCase`**`(Locale locale)` Converts all of the characters in this `String` to lower case using the rules of the given `Locale`.
`String`	**`toString`**`()` This object (which is already a string!) is itself returned.

Java – współrzędne kodowe

`String`	`toLowerCase(Locale locale)` Converts all of the characters in this `String` to lower case using the rules of the given `Locale`.
`String`	`toString()` This object (which is already a string!) is itself returned.
`String`	`toUpperCase()` Converts all of the characters in this `String` to upper case using the rules of the default locale.
`String`	`toUpperCase(Locale locale)` Converts all of the characters in this `String` to upper case using the rules of the given `Locale`.
`String`	`trim()` Returns a copy of the string, with leading and trailing whitespace omitted.
`static String`	`valueOf(boolean b)` Returns the string representation of the `boolean` argument.
`static String`	`valueOf(char c)` Returns the string representation of the `char` argument.
`static String`	`valueOf(char[] data)` Returns the string representation of the `char` array argument.
`static String`	`valueOf(char[] data, int offset, int count)` Returns the string representation of a specific subarray of the `char` array argument.
`static String`	`valueOf(double d)` Returns the string representation of the `double` argument.
`static String`	`valueOf(float f)` Returns the string representation of the `float` argument.
`static String`	`valueOf(int i)` Returns the string representation of the `int` argument.
`static String`	`valueOf(long l)` Returns the string representation of the `long` argument.
`static String`	`valueOf(Object obj)` Returns the string representation of the `Object` argument.

Java – StringBuilder

- Czasami konieczne jest złożenie łańcucha z krótszych łańcuchów, takich jak znaki wprowadzane z klawiatury albo słowa zapisane w pliku.
- Wykorzystanie konkatencji w takim przypadku jest mało efektywne, ponieważ za każdym złączeniem tworzony byłby obiekt klasy String.
- Rozwiązaniem jest wykorzystanie klasy **StringBuilder**.
- W pierwszym kroku należy utworzyć obiekt klasy **StringBuilder** za pomocą domyślnego konstruktora.
- Następnie za pomocą metody **append** należy złączać kolejne obiekty klasy String.

Java – StringBuilder

- W celu wypisanie połączonych obiektów klasy String należy wywołać metodę toString na obiekcie klasy StringBuilder.

```
public static void main(String[] args){
    String napis="To jest";
    String napis2=" napis do";
    String napis3=" wykorzystania w klasie";
    String napis4=" StringBuilder";
    StringBuilder sb = new StringBuilder();
    sb.append(napis);
    sb.append(napis2);
    sb.append(napis3);
    sb.append(napis4);
    System.out.println(sb.toString());
}
```

```
To jest napis do wykorzystania w klasie StringBuilder
```

Java – StringBuilder

Constructor Summary

Constructors

Constructor and Description
`StringBuilder()` Constructs a string builder with no characters in it and an initial capacity of 16 characters.
`StringBuilder(CharSequence seq)` Constructs a string builder that contains the same characters as the specified `CharSequence`.
`StringBuilder(int capacity)` Constructs a string builder with no characters in it and an initial capacity specified by the `capacity` argument.
`StringBuilder(String str)` Constructs a string builder initialized to the contents of the specified string.

Java – StringBuilder

Method Summary

Methods

Modifier and Type	Method and Description
StringBuilder	append(boolean b) Appends the string representation of the boolean argument to the sequence.
StringBuilder	append(char c) Appends the string representation of the char argument to this sequence.
StringBuilder	append(char[] str) Appends the string representation of the char array argument to this sequence.
StringBuilder	append(char[] str, int offset, int len) Appends the string representation of a subarray of the char array argument to this sequence.
StringBuilder	append(CharSequence s) Appends the specified character sequence to this Appendable.
StringBuilder	append(CharSequence s, int start, int end) Appends a subsequence of the specified CharSequence to this sequence.
StringBuilder	append(double d) Appends the string representation of the double argument to this sequence.
StringBuilder	append(float f) Appends the string representation of the float argument to this sequence.
StringBuilder	append(int i) Appends the string representation of the int argument to this sequence.
StringBuilder	append(long lng) Appends the string representation of the long argument to this sequence.
StringBuilder	append(Object obj) Appends the string representation of the Object argument.
StringBuilder	append(String str) Appends the specified string to this character sequence.
StringBuilder	append(StringBuffer sb) Appends the specified StringBuffer to this sequence.

Java – StringBuilder

StringBuilder	appendCodePoint(int codePoint) Appends the string representation of the codePoint argument to this sequence.
int	capacity() Returns the current capacity.
char	charAt(int index) Returns the char value in this sequence at the specified index.
int	codePointAt(int index) Returns the character (Unicode code point) at the specified index.
int	codePointBefore(int index) Returns the character (Unicode code point) before the specified index.
int	codePointCount(int beginIndex, int endIndex) Returns the number of Unicode code points in the specified text range of this sequence.
StringBuilder	delete(int start, int end) Removes the characters in a substring of this sequence.
StringBuilder	deleteCharAt(int index) Removes the char at the specified position in this sequence.
void	ensureCapacity(int minimumCapacity) Ensures that the capacity is at least equal to the specified minimum.
void	getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin) Characters are copied from this sequence into the destination character array dst.
int	indexOf(String str) Returns the index within this string of the first occurrence of the specified substring.
int	indexOf(String str, int fromIndex) Returns the index within this string of the first occurrence of the specified substring, starting at the specified index.
StringBuilder	insert(int offset, boolean b) Inserts the string representation of the boolean argument into this sequence.
StringBuilder	insert(int offset, char c) Inserts the string representation of the char argument into this sequence.
StringBuilder	insert(int offset, char[] str) Inserts the string representation of the char array argument into this sequence.
StringBuilder	insert(int index, char[] str, int offset, int len) Inserts the string representation of a subarray of the str array argument into this sequence.

Java – StringBuilder

StringBuilder	**insert**(int dstOffset, CharSequence s) Inserts the specified CharSequence into this sequence.
StringBuilder	**insert**(int dstOffset, CharSequence s, int start, int end) Inserts a subsequence of the specified CharSequence into this sequence.
StringBuilder	**insert**(int offset, double d) Inserts the string representation of the double argument into this sequence.
StringBuilder	**insert**(int offset, float f) Inserts the string representation of the float argument into this sequence.
StringBuilder	**insert**(int offset, int i) Inserts the string representation of the second int argument into this sequence.
StringBuilder	**insert**(int offset, long l) Inserts the string representation of the long argument into this sequence.
StringBuilder	**insert**(int offset, Object obj) Inserts the string representation of the Object argument into this character sequence.
StringBuilder	**insert**(int offset, String str) Inserts the string into this character sequence.
int	**lastIndexOf**(String str) Returns the index within this string of the rightmost occurrence of the specified substring.
int	**lastIndexOf**(String str, int fromIndex) Returns the index within this string of the last occurrence of the specified substring.
int	**length**() Returns the length (character count).
int	**offsetByCodePoints**(int index, int codePointOffset) Returns the index within this sequence that is offset from the given index by codePointOffset code points.
StringBuilder	**replace**(int start, int end, String str) Replaces the characters in a substring of this sequence with characters in the specified String.
StringBuilder	**reverse**() Causes this character sequence to be replaced by the reverse of the sequence.
void	**setCharAt**(int index, char ch) The character at the specified index is set to ch.
void	**setLength**(int newLength) Sets the length of the character sequence.

Java – StringBuilder

CharSequence	**subSequence**(int start, int end) Returns a new character sequence that is a subsequence of this sequence.
String	**substring**(int start) Returns a new String that contains a subsequence of characters currently contained in this character sequence.
String	**substring**(int start, int end) Returns a new String that contains a subsequence of characters currently contained in this sequence.
String	**toString**() Returns a string representing the data in this sequence.
void	**trimToSize**() Attempts to reduce storage used for the character sequence.

Java – typ wyliczeniowy

- **Typ wyliczeniowy (ang. Enumerable)** umożliwia programiście wprowadzenie skończonej liczby nazwanych wartości dla nowego typu.
- Typ wyliczeniowy określany jest słowem kluczowym **enum**.
- Przykładowo w sklepie internetowym możliwy jest wybór rozmiaru koszulki.
- Wpisując rozmiar ręcznie, bardzo łatwo o pomyłkę bądź wpisanie rozmiaru, który nie jest możliwy do zrealizowania w sklepie.
- W celu uniknięcia powyższych problemów należy zdefiniować typ ROZMIAR, który może przyjmować jedynie wartości S,M,L,XL,XXL.

Java – typ wyliczeniowy

- Typ wyliczeniowy jako wartość może przyjmować jedną z określonych wartości lub wartość typu null.

```
public class Class1 {

    enum ROZMIAR {S,M,L,XL,XXL};
    enum ROZMIAR_DAMSKI {XS,S,M,L};
    public static void main(String[] args){
        ROZMIAR rozmiar_koszulki = ROZMIAR.XL;
        ROZMIAR_DAMSKI rozmiar_koszulki_damski = ROZMIAR_DAMSKI.XS;
        System.out.println("Rozmiar koszulki wynosi: "+rozmiar_koszulki);
        System.out.println("Rozmiar koszulki damskiej wynosi: "+rozmiar_koszulki_damski);
    }
}
```

```
Rozmiar koszulki wynosi: XL
Rozmiar koszulki damskiej wynosi: XS
```

Java – wyjątki

- Sytuacje wyjątkowe, które mogą wywołać awarię programu (np. wprowadzenie złej wartości, nieistniejący plik) w Java są obsługiwane za pomocą techniki **obsługi wyjątków**.
- W trakcie działania programu i wystąpienia błędu program powinien zachować się w jeden z dwóch sposobów:
  - powrócić do bezpiecznego stanu i pozwolić użytkownikowi wykonać inne polecenie,
  - pozwolić użytkownikowi zapisać całą pracę i zamknąć program.
- Powyższe zachowania są ciężkie do realizacji, ponieważ zazwyczaj do błędów dochodzi z dala od kodu, który umożliwiałby wrócenie programu do bezpiecznego stanu, bądź zapisanie pracy.

Java – wyjątki

- Obsługa wyjątków polega na przekazywaniu kontroli z miejsca wystąpienia błędu do procedur, które mogą rozwiązać ten problem.
- W celu obsługi sytuacji wyjątkowych w programie, należy przewidzieć jakiego rodzaju błędy i problemy mogą wystąpić.
- Należy rozważyć następujące elementy:
  - **Błędy danych wejściowych** – np. użytkownik wprowadza niepoprawną wartość (dzielenie przez zero) bądź nie trzyma się konwencji przy adresach **URL** (wystąpienie błędu związanego z warstwą sieci).

Java – wyjątki

- **Błędy urządzeń** – urządzenia we/wy nie zawsze działają prawidłowo. Należy przewidzieć np. możliwość braku papieru w drukarce, zacięcie papieru, brak sterowników urządzenia.
- **Ograniczenia fizyczne** – np. brak miejsca na dysku twardym niezbędnego do zapisu pliku, przepełniona baza danych.
- **Błędy w kodzie** – niepoprawnie działająca metoda np. poprzez zwracanie złych wyników lub złe wywołania innych metod. Najczęstsze błędy w kodzie dotyczą np. indeksów tablicy, odwołań do pustych elementów lub próby pobrania z pustego stosu.

Java – wyjątki

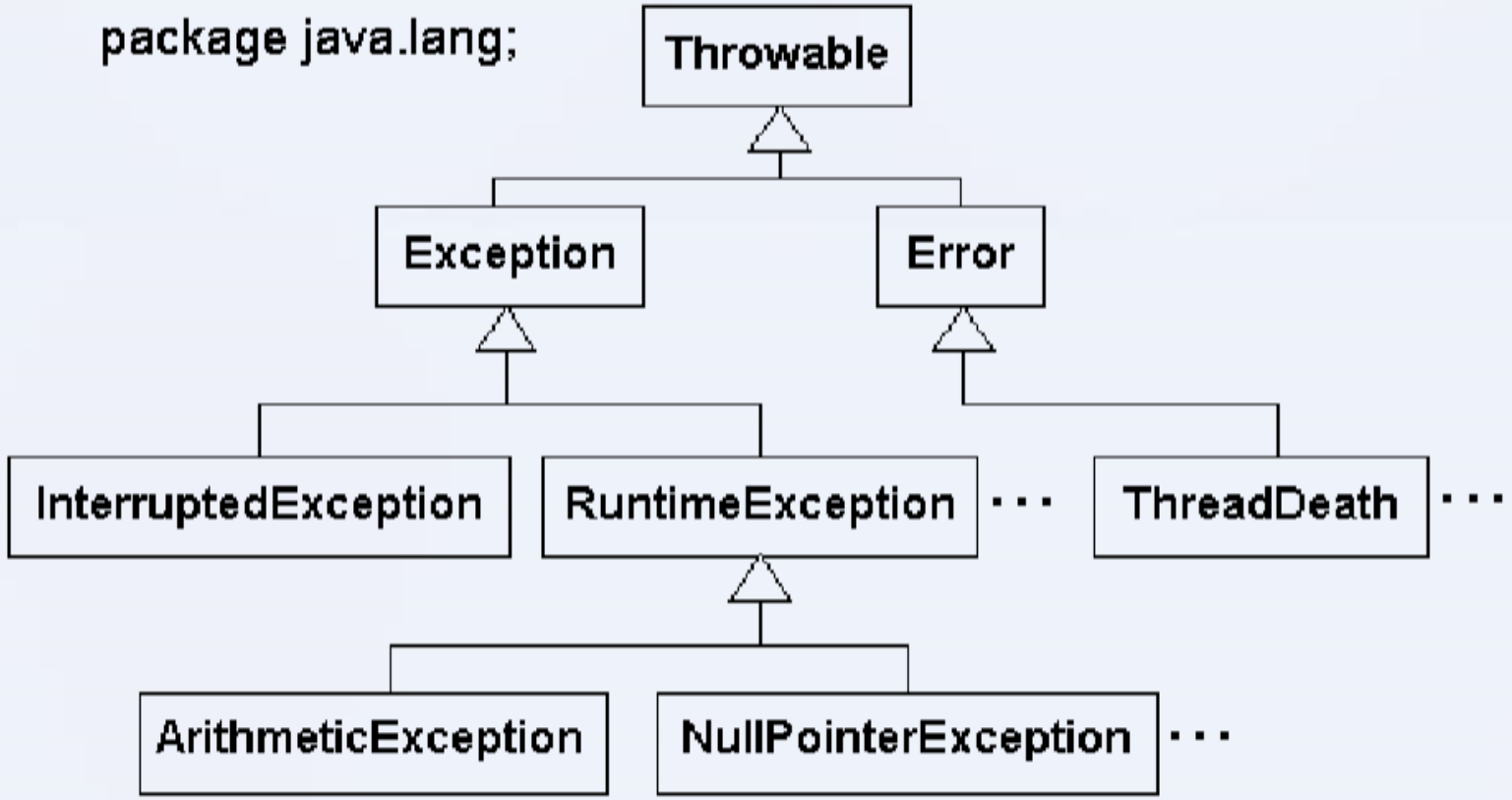
- Typową reakcją na błędy w metodzie jest zwrot specjalnego kodu błędu, który następnie przekazywany jest metodzie wywołującej.
- W trakcie dostępu do plików bardzo często zwraca się wartość -1 zamiast standardowego znaku końca pliku.
- Innym podejściem jest zwracanie wartości **null** z metody.
- Nie zawsze jednak zwracanie określonej wartości w przypadku wystąpienia błędu jest możliwe.
- Przykładowo metoda która oblicza różnicę dwóch liczb może zwrócić wartość -1 i nie oznacza to wystąpienia błędu.

Java – wyjątki

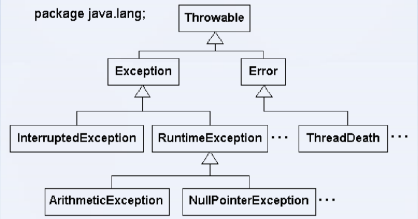
- W związku z tym jeżeli metoda nie może zakończyć działania w przewidziany sposób to nie zwraca wartości, ale **wyrzuca (ang. throw)** obiekt, który zawiera informacje o błędzie.
- W przypadku wyrzucenia obiektu z informacjami o błędzie, metoda jest kończona natychmiastowo bez zwracania wartości.
- Dodatkowo wykonywanie programu nie jest zwracane w miejsce wywołania metody.
- Zamiast tego mechanizm obsługi **wyjątków (ang. expcetion)** zaczyna poszukiwać mechanizmu obsługi błędów, który potrafi rozwiązać zadany problem.

Java – wyjątki

- Wyjątki posiadają własną składnię oraz wchodzą w skład specjalnej hierarchii dziedziczenia.
- Typ obiektu wyjątku w Java jest zawsze pochodną klasy **Throwable**.

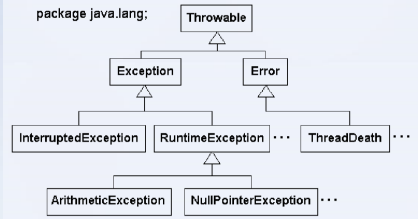


Java – wyjątki



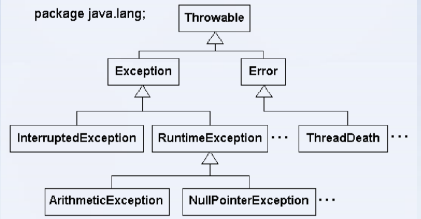
- Klasy będące potomkami klasy **Error** odpowiadają błędom wewnętrznym i wyczerpaniu zasobów w środowisku uruchomieniowym.
- Nie należy wyrzucać obiektów tego typu, ponieważ jeżeli wystąpił błąd wewnętrzny to nie można za dużo zrobić.
- Najlepszym rozwiązaniem w tym przypadku jest powiadomienie użytkownika i zamknięcie programu.
- Błędy typu **Error** występują jednak niezbyt często.

Java – wyjątki



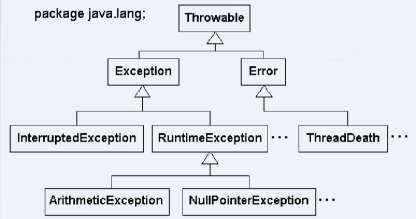
- W Java dużo częściej wykorzystuje się klasę **Exception**.
- Dzieli się ona na dwie gałęzie: wyjątki związane z wykonywaniem (**RuntimeException**) oraz pozostałe.
- Ogólna zasada mówi, że wyjątki typu **RuntimeException** są spowodowane przez błędy programisty.
- Pozostałe wyjątki mają związek z niepożądanymi zdarzeniami takimi jak np. błędy wejścia-wyjścia, które zaszły w dobrym programie.
- Do wyjątków dziedziących po **RuntimeException** należą:
  - Niepoprawne rzutowanie,
  - Dostęp do nieistniejącego elementu tablicy,
  - Dostęp do pustego wskaźnika.

Java – wyjątki



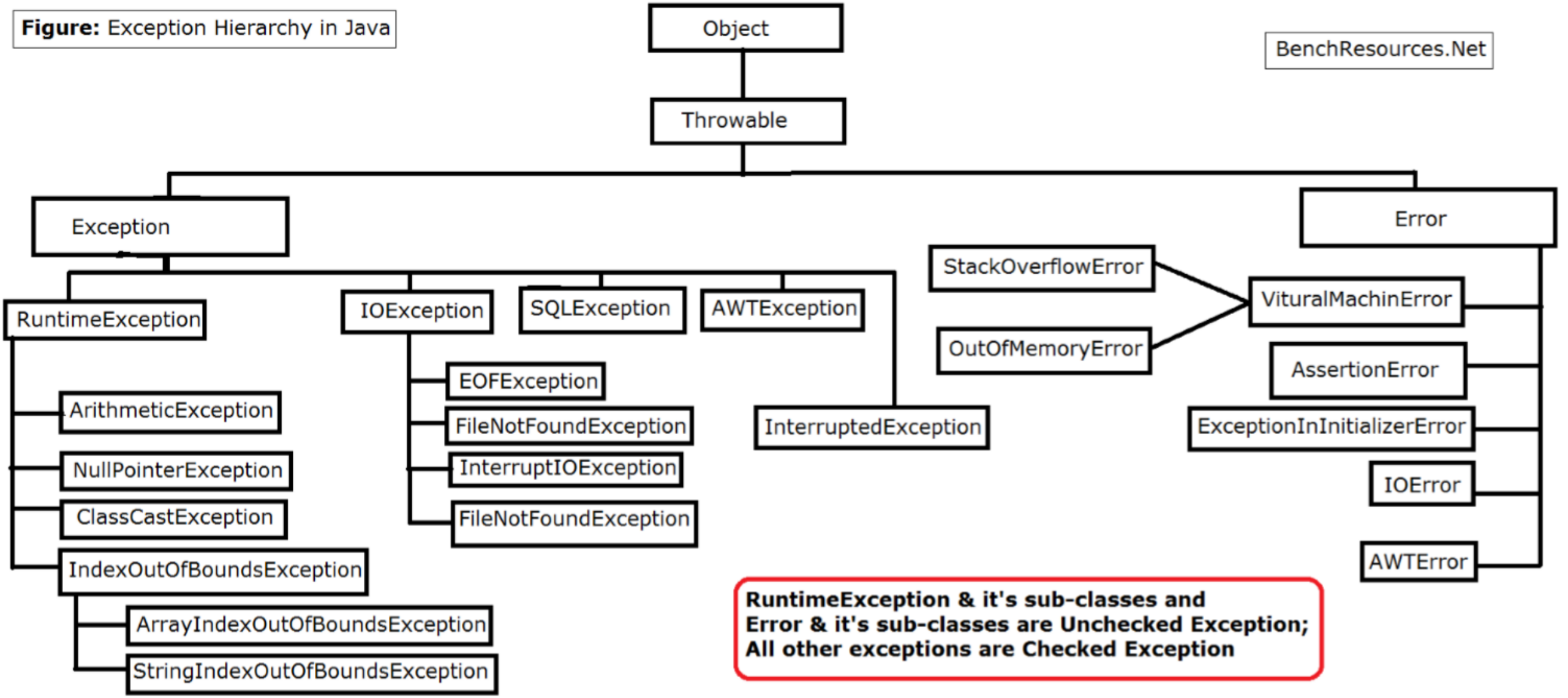
- Do wyjątków nie **dziedziących** po **RuntimeException** należą:
  - Próba odczytania za końcem pliku,
  - Próba otwarcia niepoprawnego adresu URL,
  - Próba znalezienia obiektu typu Class dla łańcucha, który nie zgadza się z żadną z istniejących klas.
- Zasada „jeśli wystąpił wyjątek **RuntimeException**, znaczy że popełniłeś błąd” sprawdza się w większości przypadków.
- Wyjątku **ArrayIndexOutOfBoundsException** można uniknąć sprawdzając indeks z rozmiarem tablicy.
- Wyjątku **NullPointerException** można uniknąć sprawdzając czy zmienna nie ma przypisanej wartości null.

Java – wyjątki

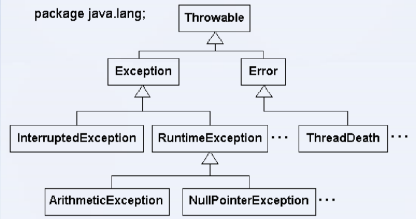


- W przypadku adresów URL można dokonać weryfikacji ogólnej jego składni, jednak różne przeglądarki różnie interpretują adresy URL. Przykładowo Chrome czy Firefox poradzi sobie z tagiem mailto:, jednak przeglądarka apletów już nie.
- Wszystkie wyjątki typu **Error lub RuntimeException** nazywane są **wyjątkami niekontrolowanymi (ang. Unchecked exception).**
- Wszystkie pozostałe wyjątki nazywane są **wyjątkami kontrolowanymi (ang. Checked exception).**
- Kompilator sprawdza jedynie czy dostarczono procedur obsługi dla wszystkich wyjątków kontrolowanych.

Java – wyjątki



Java – wyjątki

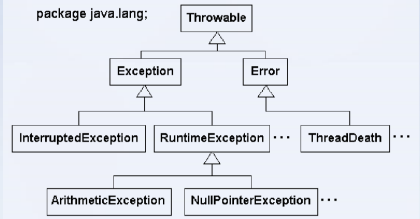


- Metoda w Java może nie tylko zwrócić wartość, ale również rzucić wyjątek.
- Metoda musi poinformować kompilator o możliwości rzucenia wyjątku poprzez dopisanie do niej słowa kluczowego throws oraz nazwy wyjątku, który może rzucić.

```
public static String ReadFile(String path) throws FileNotFoundException
```

- W przypadku rzucenia wyjątku **FileNotFoundException** program rozpocznie poszukiwanie procedury obsługi wyjątku.
- Pisząc metodę nie trzeba informować o każdym możliwym rodzaju wyjątku, który może zostać przez nią zgłoszony.

Java – wyjątki

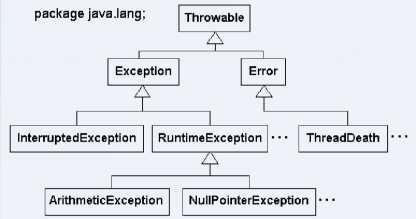


- Jeżeli metoda może zgłaszać wyjątki kontrolowane różnego typu, to w jej nagłówku musi znaleźć się ich list, oddzielona znakiem „,”

```
public static String ReadFile(String path) throws FileNotFoundException, EOFException
```

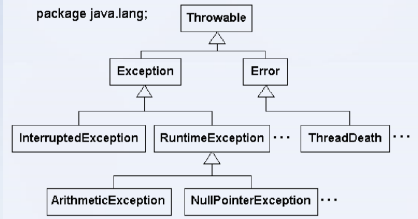
- Nie należy natomiast informować o wyjątkach wewnętrznych Java, czyli tych, które dziedziczą po klasie **Error.**
- Dodatkowo nie należy informować o wyjątkach dziedziczących po klasie **RuntimeException.**
- Wyjątki tego typu mogą być w pełni kontrolowane przez programistę.
- Jeżeli istnieje ryzyko wystąpienia błędu związanego z indeksem tablicy to należy się przed nim zabezpieczyć, a nie informować o możliwości jego rzucenia.

Java – wyjątki



- W Java możliwe jest również przechwycenie wyjątku, a nie jego rzucanie.
- W przypadku przechwytywanie wyjątku przez metodę nie jest potrzebna specyfikacja **throws**.
- Jeżeli metoda deklaruje możliwość rzucenia wyjątku określonej klasy, to może rzucić wyjątek tej klasy lub dowolnej po niej dziedziczącej.
- Przykładowo jeżeli metoda ma zapis **throws IOException** to może rzucić wyjątek **klasy IOException** lub wyjątek klasy **FileNotFoundException**, która dziedziczy po **IOException**.

Java – wyjątki



- Zgłaszanie wyjątku w Java wygląda następująco:

```
public static String ReadFile(String path) throws EOFException
{
    //.......
    throw new EOFException();
}
```

```
public static String ReadFile(String path) throws EOFException
{
    //.......
    EOFException e = new EOFException();
    throw e;
}
```

- Zgłaszanie wyjątku jest „proste” o ile istnieje odpowiednia klasa wyjątku.
- W przypadku istnienia takiej klasy należy:
    1. Znaleźć odpowiednią klasę wyjątku.
    2. Utworzyć obiekt tej klasy.
    3. Rzucić utworzony obiekt.
- Kiedy metoda zgłasza wyjątek to nie zwraca wartości do wywołującego!

Java – wyjątki

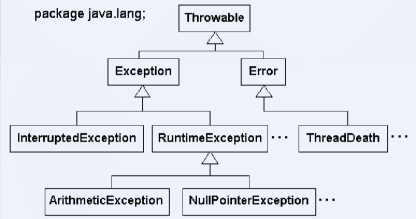
- Programista Java może napotkać sytuację, w której żadna klasa ze standardowych klas wyjątków nie pasuje.
- W takim przypadku można utworzyć własną klasę wyjątku, która powinna dziedziczyć po klasie **Exception** lub jednej z jej podklas np. **IOException**.
- Istnieje zwyczaj, w którym dostarcza się zarówno konstruktor domyślny dla klasy wyjątku jak i konstruktor pobierający szczegółowe dane.
- Metoda super pozwala wywołać konstruktor klasy nadrzędnej.

Java – wyjątki

```
import java.io.IOException;

public class FileFormatException extends IOException {
    public FileFormatException()
    {

    }
    public FileFormatException(String details)
    {
        super(details);
    }
}
```

```
public static String ReadFile(String path) throws FileFormatException
{
    //.......

    throw new FileFormatException();
}
```

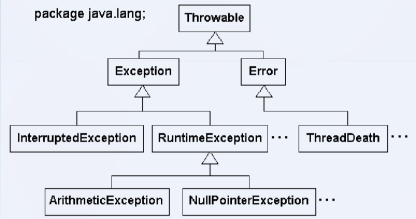
Java – wyjątki



- Rzucanie wyjątków przedstawione na poprzednich slajdach jest zadaniem prostym.
- Gorsze jest planowanie i przechwytywanie wyjątków.
- Jeżeli wyjątek nie zostanie przechwycony to program zakończy działanie, a w oknie **IDE** zostaną wypisane dane ze śledzenia **stosu**.
- Do przechwytywania wyjątków służy **blok try-catch**.

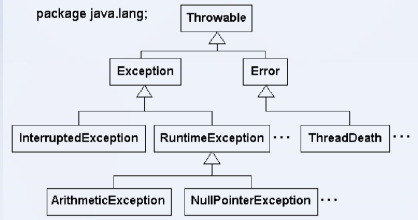
```
try
{
    String file_content = ReadFile("c:\\plik.mp3");
}
catch(FileFormatException e)
{
    System.out.println("Złapano wyjątek");
}
```

Java – wyjątki



- Jeżeli, którykolwiek fragment kodu w bloku try zgłosi wyjątek określonego typu w klauzuli catch to:
  - program pominie dalsze wykonywanie kodu w klauzuli **try**,
  - program wykona procedurę obsługi wyjątku w klauzuli **catch**.
- Jeżeli żaden fragment kodu w bloku **try** nie zgłosi wyjątku to klauzula **catch** zostanie pominięta.
- Jeżeli którykolwiek fragment kodu w bloku **try** zgłosi inny typ wyjątku niż określony w klauzuli **catch** to program natychmiast wychodzi z metody (może gdzieś wyżej w stosie wywołań znajduje się **catch** przeznaczony dla tego typu wyjątku).

Java – wyjątki



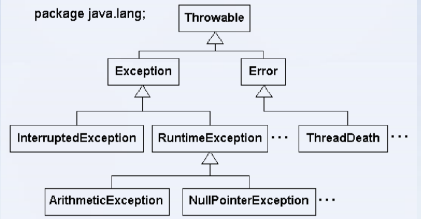
- W przypadku wyjątków są dwa podejścia.
- Obsłużenie wyjątku lub jego rzucenie do wywołującego metodę.
- W Java przyjęto konwencję, aby obsługiwać wyjątki, które jesteśmy w stanie.
- Jeżeli nie jesteśmy w stanie obsłużyć wyjątku to należy o tym poinformować za pomocą klauzuli throws.
- Za pomocą instrukcji **try-catch** można przechwycić wiele typów wyjątków.
- Dla każdego typu wyjątku należy w takim przypadku napisać oddzielną klauzulę catch.

Java – wyjątki

```
public static String ReadFile(String path) throws FileFormatException, FileNotFoundException
{
    //.......
    if(path=="" || path==null)
    {
        throw new FileNotFoundException();
    }
    else
    {
        throw new FileFormatException();
    }

}
```

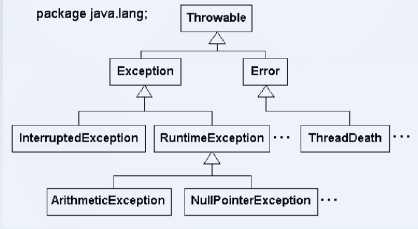
```
public static void main(String[] args){
    try
    {
        String file_content = ReadFile("c:\\plik.mp3");
    }
    catch(FileNotFoundException e)
    {
        System.out.println("Złapano wyjątek FileNotFoundException");
    }
    catch(FileFormatException e)
    {
        System.out.println("Złapano wyjątek FileFormatException");
    }
}
```

Java – wyjątki



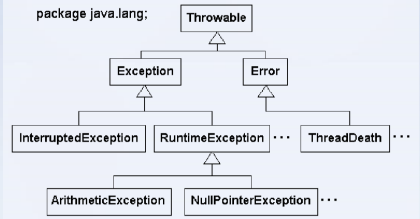
- Obiekt wyjątku może zawierać informacje o naturze wyjątku.
- W celu dowiedzenia się więcej o danym obiekcie należy wywołać na nim metodę **getMessage().**
- Istnieje również możliwość uzyskania szczegółowych informacji o wyjątku lub rzeczywistego typy wyjątku. W tym celu należy na obiekcie wyjątku wywołać metodę **getClass().getName().**
- Od Java SE7 istnieje możliwość przechwytywania różnych typów wyjątków w jednej klauzuli catch.
- Powyższa możliwość przynosi korzyści w przypadku kiedy dwa lub więcej typów wyjątków są obsługiwane w ten sam sposób.

Java – wyjątki




```
public static void main(String[] args){
    try
    {
        String file_content = ReadFile("");
    }
    catch(FileNotFoundException | FileFormatException e)
    {
        System.out.println("Złapano wyjątek typu: "+e.getClass().getName()+". "
                + "Szczegółowe informacje: "+e.getMessage());
    }
}
```

Java – wyjątki



- Wyjątki można również rzucać w klauzuli catch.
- Zazwyczaj robi się to w przypadku konieczności zmiany typu wyjątku.
- Dla dużych projektów zalecane jest rzucanie wyjątku takiego typu, aby określał w którym elemencie (podsystemie) doszło do błędu.

```
try
{
    String file_content = ReadFile("");
}
catch(FileNotFoundException | FileFormatException e)
{
    throw new IOException();
}
```

Java – wyjątki

- Do nowego wyjątku można dołączyć wiadomość ze starego wyjątku (w konstuktorze nowego wyjątku należy wywołać metodę **getMessage** na obiekcie przechwyconego wyjątku).
- Praktyczniejszym podejściem jest ustawienie pierwotnego wyjątku jako powód (**ang. Cause**) dla nowego wyjątku.

```
private static void Metoda() throws IOException
{
    try
    {
        String file_content = ReadFile("");
    }
    catch(FileNotFoundException | FileFormatException e)
    {
        throw new IOException(e.getMessage());
    }
}
public static void main(String[] args){
    try
    {
        Metoda();
    }
    catch(IOException e)
    {

    }
}
```

```
private static void Metoda() throws IOException
{
    try
    {
        String file_content = ReadFile("");
    }
    catch(FileNotFoundException | FileFormatException e)
    {
        IOException f = new IOException();
        f.initCause(e);
        throw f;
    }
}
public static void main(String[] args){
    try
    {
        Metoda();
    }
    catch(IOException e)
    {

    }
}
```

Java – wyjątki

- Kiedy zostaje rzucony wyjątek to reszta programu nie jest wykonywana.
- Może to spowodować problemy z dostęp do niektórych zasobów np. pozostawienie otwartego pliku lub połączenia do bazy danych.
- Rozwiązaniem jest klauzula finally.
- Kod w klauzuli **finally** jest wykonywany zawsze, bez względu na to czy wyjątek zostanie przechwycony czy też nie.

```
public static void main(String[] args){
    try
    {
        Metoda();
    }
    catch(IOException e)
    {

    }
    finally
    {
        //zamknięcie pliku
    }
}
```

Java – wyjątki

- W Java 7 istnieje możliwość zamykania zasobów bez klauzuli **finally**.
- Zasób musi jedynie implementować interfejs **AutoCloseable**, który posiada tylko jedną metodę **void close() throws Exception**.
- Niezależnie czy zostanie rzucony wyjątek czy też nie to zasób zostanie zamknięty.
- Taka konstrukcja pozwala na niewykorzystanie klauzuli **finally**.

Java – wyjątki

- Wskazówki dotyczące obsługi wyjątków:
  1. **Obsługa wyjątków nie może zastąpić prostych testów** – należy testować możliwość działania programu np. sprawdzenie czy stos nie jest pusty, a nie rzucenie wyjątku.

  2. **Nie rozdrabniać się** – nie należy używać klauzul try-catch dla każdej linii kodu. Należy grupować kod.

  3. **Właściwie wykorzystywać hierarchię wyjątków** – nie należy generować każdorazowo wyjątku typu RuntimeException. Należy respektować różnicę pomiędzy wyjątkami kontrolowanymi, a niekontrolowanymi. Nie należy bać się zmiany typu wyjątku, jeżeli nowy odpowiada bardziej do błędu który został wygenerowany w aplikacji.

Java – wyjątki

- Wskazówki dotyczące obsługi wyjątków:

  4. Nie ukrywać wyjątków – jeżeli metoda ma zgłaszać wyjątek to należy to zrobić i przekazać go dalej. Każda klasa wywołująca tą metodę będzie musiała go obsłużyć. Często programiści obsługują wyjątek wewnątrz metody nie przekazując go dalej.

  5. Nie unikać wyjątków – lepiej jest rzucić wyjątek niż zwrócić wartość, które nie określa go jednoznacznie np. zwrócić wartość null z metody.

  6. Nie bać się przekazywania wyjątków – jeżeli jest taka konieczność to należy wyjątek przekazać dalej, a nie na wszelkie możliwe sposoby wyłapać je wewnątrz metody.

Java – wyjątki

Następny wykład: Pola i metody statyczne w języku Java. Parametry metod. Dzienniki w Java

DZIĘKUJĘ ZA UWAGĘ