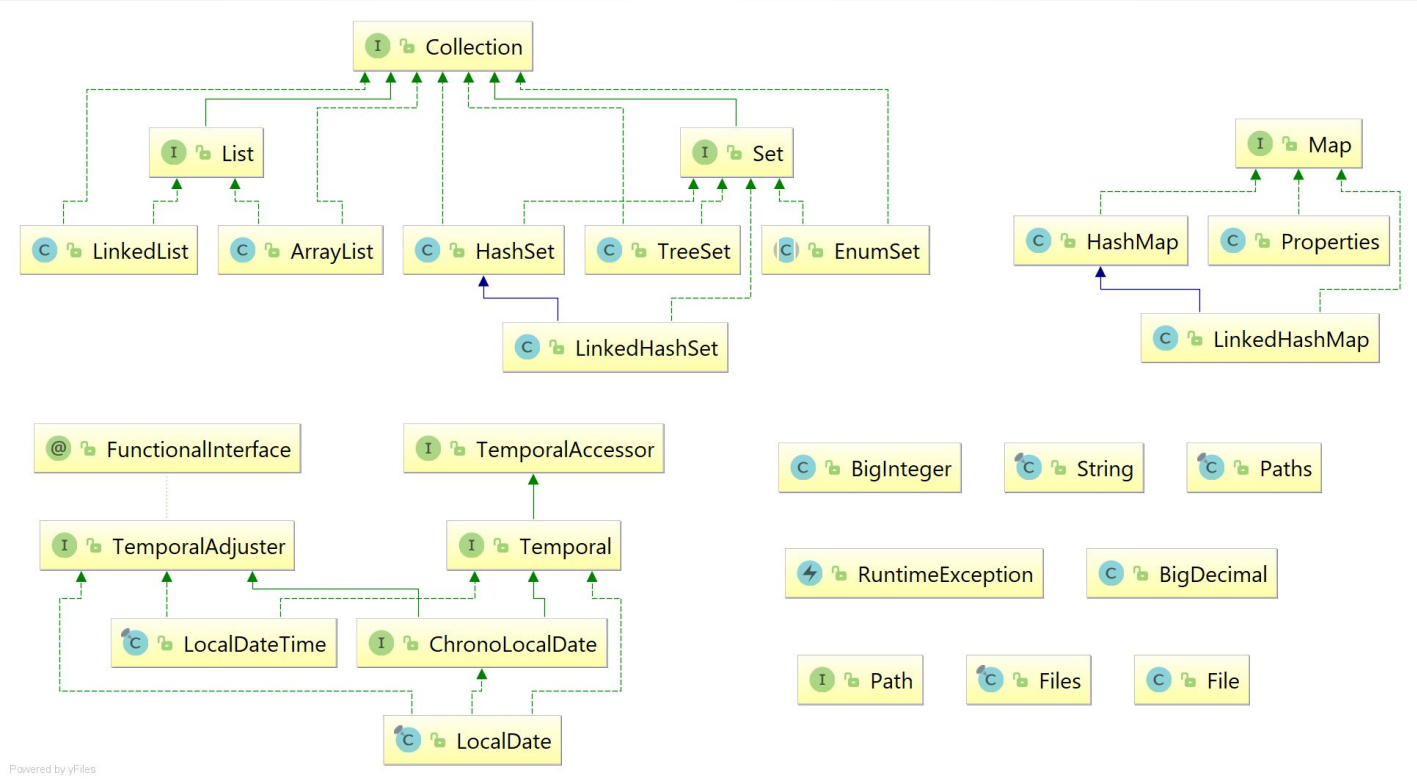
Data Structures Processing

Maciej Koziara

What is data structure?

Data structure is designed to organize data to suit a specific purpose so that it can be accessed and worked with in appropriate ways.

Java data structures



Let’s get started!

Immutable model

// Immutable means you cannot change internal values of the class
// Most Java structures are immutable
// Immutable is very important in professional software development (but sometimes not possible to achieve)
// Immutability makes it easier to reason about code

Strings - be user friendly!

// Strings are immutable!
// Use *trim()* to get rid of whitespaces at the beginning and end of the user input
// Use *equalsIgnoreCase()* instead of *equals()* to ignore the size of letters that the user entered

Regular expressions

// Used for searching a complicated patterns in texts
// Different languages may implement different number of features or implement them differently

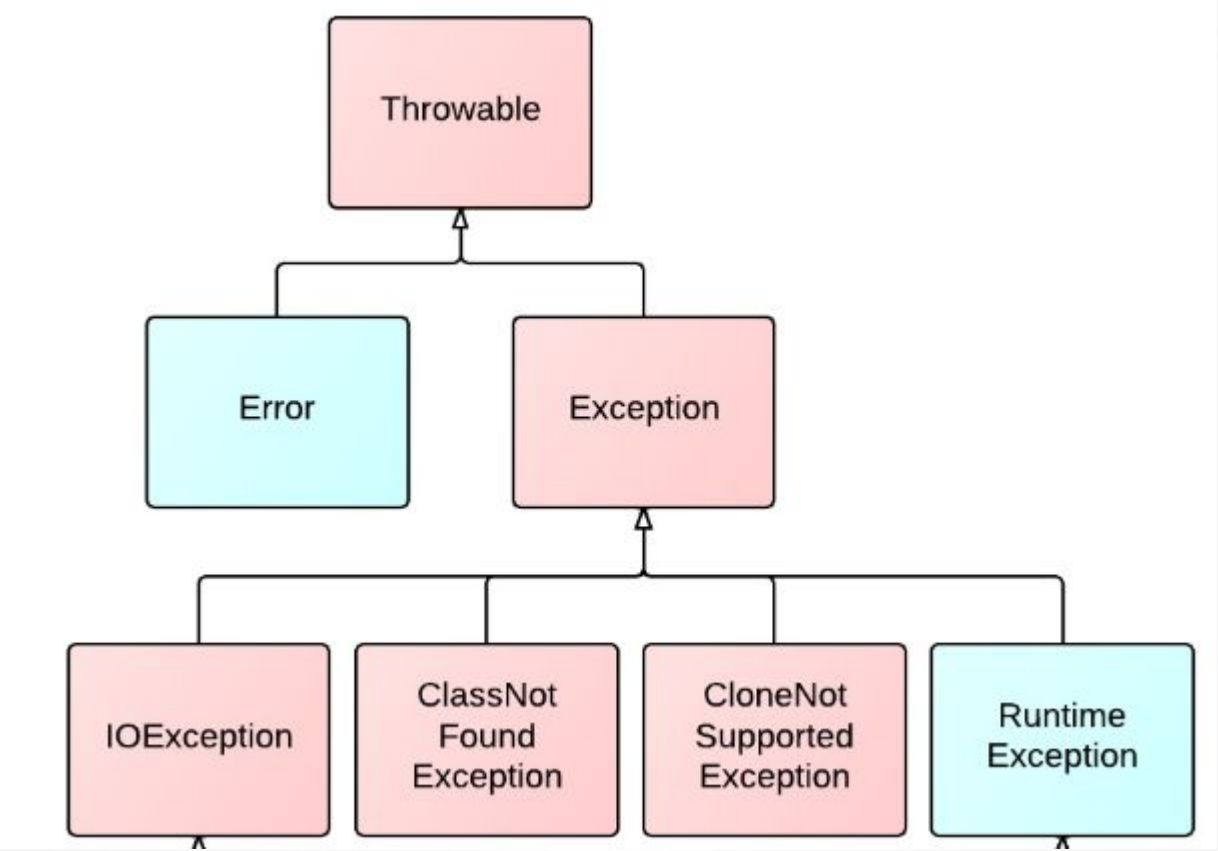
RegEx - basic operators

Operator	Meaning
abc	concrete letters
.	any character
?	one or zero character
*	zero or more character
+	one or more characters
\d	only digits allowed
\D	digits not allowed

RegEx - basic operators

Operator	Meaning
\w	only alphanumeric characters
\W	alphanumeric character not allowed
{4}	exactly 4 characters
{4, 6}	from 4 to 6 characters
^	starts with
$	ends with
[a-z0-5#]	only lowercase letters, digits from 0 to 5, and hash sign allowed

Exceptions - hierarchy



Exceptions - hierarchy

// **Throwable** - root interface of Exception hierarchy. Indicates that class may be used in **throw new** clause
// **Exception** - indicates exceptional situations that application may recover with a little help from programmer. Requires usage of **try catch.** Example: FileNotFoundException

Exceptions - hierarchy

// **RuntimeException** - exceptions that can be prevented programmatically by for example validating user inputs. NullPointerException is such exception.
// **Error** - exceptions reserved for internal usage by JVM. Indicate system or JVM problems. Example: OutOfMemoryError

Exceptions - good practices

// Prefer usage of **RuntimeExceptions**
// Do not catch general **Exception** class. Instead catch specific exception, like **FileNotFoundException**
// Catch and handle exceptions as soon as possible
// Do not ignore exceptions
// If you create custom exception include as many informations about exception cause as possible

Files / Paths

// New approach for handling files in Java
// **Path -** represents path to file or directory in a file system
// **Paths -** used for creating **Path** instances
// **Files** - contains different file operations, examples: *copy(), createFile(), readAllLines()*

Files / Paths - good practices

// Always use **Files** and **Paths** classes for operations on files. Do not use old **BufferedReader** approach
// While creating **Path** using **Paths.get()** method do not use it with full path String - **Paths.get(“*/usr/my/path*”)**. Each part of path should be pass separately - **Paths.get(“*usr*“, “*my*“, “*path*”)** - so Java may handle differences in path separators on different operating systems

Try with resources

// Syntax that handles closing streams automatically
// Class needs to implement **Closeable** interface to be used with this syntax

```
try(FileReader bufferedReader = new FileReader( fileName: "paths/data/example.txt")) {

}
```

Properties

// Used for configuring Java Applications
// May be kept in .properties or .xml files but:

Prefer configuration in .properties files instead of .xml

```
version=1.0
name=Infoshare Academy application
```

Json

// Lightweight data-interchange format
// Easy for humans to read and write
// Easy for machines to parse and generate
// Used for communication between different systems
// Used for storing data, both simple values and complex objects

Json

// Supports following types: **String**, **boolean**, **number**, **arrays**, **Object**

// Remember about curly braces at the beginning (Json Object start) and at the end (Json Object end).

```
{
  "name": "Maciek",
  "age": 25,
  "unemployed": false,
  "companies": ["Infoshare Academy", "Other company"]
}
```

Dates

// **LocalDate** and **LocalDateTime** has been introduced to simplify dates handling in Java
// Always use **LocalDate** or **LocalDateTime** when working with dates. Never legacy approaches with **Date** or **Calendar** classes
// Use **ChronoUnit.between()** method to count period between two dates
// Use *parse()* to create date from string
// Use *format()* to convert date to string with custom format

Dates - formatting

// **DateTimeFormatter** - used for formatting **LocalDate** and **LocalDateTime**

Operators:
y - year
M - month
d - days

Dates - formatting example

// **yyyy-MM-dd -** 2018-07-11
// **yy-dd-MMMM -** 2018-11-July
// **yyyy-MM-dddd -** 2018-07-Wednesday
// **yyyy-MMM-ddd -** 2018-Jul-Wed

// Remember to use **Locale** to specify language of month and day names

BigInteger, BigDecimal

// Solves problems with incorrect handling of mathematical operations while using **Double** types
// It is recommended to always use these classes for mathematical operations, especially while working with money or other fragile data

BigInteger, BigDecimal - equals

When working with BigDecimal never use *equals()* for equality checking. Use *compareTo() instead*.