Advanced Programming Methods

Iuliana Bocicor
maria.bocicor@ubbcluj.ro

Babes-Bolyai University

2024

Overview

Exceptions I

- Exception - an abnormal situation that occurs during program execution.

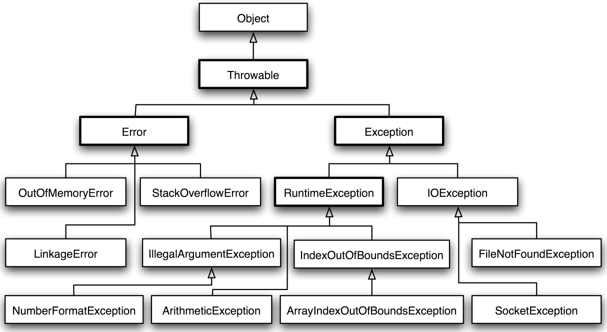

Figure: Figure source: ProTech

Exceptions II

- An *Error* "indicates serious problems that a reasonable application should not try to catch. Most such errors are abnormal conditions" (Java API).
- An *Exception* "indicates conditions that a reasonable application might want to catch." (Java API).
- *Exception* and any subclasses that are not also subclasses of *RuntimeException* are *checked exceptions*.
- Checked exceptions are verified at compile time and need to be declared in a method or constructor's throws clause.

Exceptions III

- *Error* and *RuntimeException* and their subclasses are *unchecked exceptions*.
- The JVM throws *Error* (or any of its subclasses) in fatal circumstances, when there is no way for the application to recover.
- Unchecked exceptions can be generated without having to be declared in throws clauses.

Throw and throws

- throw is used to explicitly throw an exception from a method.
- The thrown exception must be an instance of *Throwable*.
- throws is used in the signature of a method to indicate that the method might throw an exception.
- The caller of the method will have to handle the exception using a try-catch block.

try-catch-finally

```
try
{
    method1();
    method2(); // throws exception
    method3(); // is no longer executed if
               // the exception is thrown
}
catch (Type_Exception1 ex1){ // handle ex1 }
catch (Type_Exception1 ex2){ // handle ex2 }
catch (Type_Exception3 | Type_Exception4 ex3){
    // handle ex3
}
finally{
    // this code is always executed
}
```

Defining custom exceptions

- Java exceptions cover almost all exceptions that may arise in a program.
- However, there might be cases in which we need our own exceptions.
- Usually custom exception classes extend *Exception*.

Class Exception

- Constructors:
    - *Exception()*
    - *Exception(String message)*
    - *Exception(String message, Throwable cause)*
    - *Exception(Throwable cause)*
- Methods (most popular):
    - *getCause(): Throwable*
    - *getMessage(): String*
    - *printStackTrace()*
    - *printStackTrace(PrintStream s)*

Exceptions in inheritance

- The derived class constructor must specify all exceptions specified in the base class constructor and may add new exceptions.
- Overidden methods may only throw exceptions specified in the base class method or subclasses of these exceptions.
- The rule mentioned above is also applied to methods declared in interfaces.

Examples

Example

exceptions.Examples.

Java I/O

- The *java.io* package offers:
  - Classes that work with bytes: *InputStream*, *OutputStream*.
  - Classes that work with characters: *Reader*, *Writer*.
  - Classes for conversions bytes-characters: *InputStreamReader*, *OutputStreamWriter*.
  - Classes for object serialization: *ObjectInputStream*, *ObjectOutputStream*.

Streams - recap

- A stream is an abstraction for receiving/sending data in an input/output situation.
- The stream is used to pass on data.
- Concrete implementations receive data from a specific source (input - keyboard, memory, file, pipe) and send it to a specific destination (output - display, memory, file, pipe).
- Streams are generally associated to a physical source or destination of characters, like a disk file, the keyboard, or the console.
- Streams support various types of data: bytes, characters, primitive data types, objects.

InputStream I

- The abstract class *InputStream* is the superclass of all classes representing an input stream of bytes.
- All its derived (non-abstract) classes implement the method *read()*.

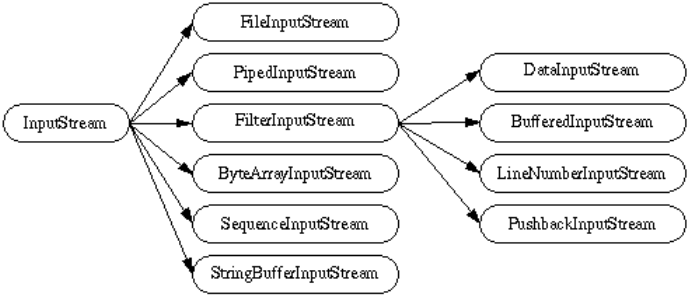

Figure: Figure source: Overview of Input and Output Streams

InputStream II

- *InputStream* defines methods for:
  - reading bytes or arrays of bytes;
  - marking locations in the stream;
  - skipping bytes of input;
  - resetting the current position.
- An input stream is automatically opened when it is created.
- It can be closed with the *close()* method, or let to be closed implicitly by the garbage collector. However, the best practice is to close the stream once it is no longer used.

OutputStream I

- The abstract class *OutputStream* is the superclass of all classes representing an output stream of bytes.
- The method *write()* must be defined by its non-abstract subclasses.

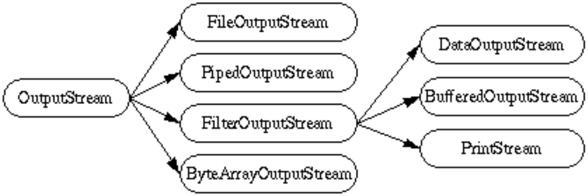

Figure: Figure source: Overview of Input and Output Streams

OutputStream II

- *OutputStream* offers methods for writing bytes or arrays of bytes.
- An output stream is automatically opened when it is created.
- It can be closed with the *close()* method, or let to be closed implicitly by the garbage collector. However, the best practice is to close the stream once it is no longer used.

Reader and Writer

- Reader and Writer are abstract classes for reading/writing characters from/to a stream.
- Important methods that must be implemented by subclasses:
  - *read()*
  - *write()*

Example

streams.I_O_Reader_Writer.

Available classes

Operation	Bytes	Char
Work with files.	FileInputStream, FileOutputStream	FileReader, FileWriter
Store in memory.	ByteArrayInputStream, ByteArrayOutputStream	CharArrayReader CharArrayWriter
Buffered operations.	BufferedInputStream BufferedOutputStream	BufferedReader BufferedWriter
Formatting.	PrintStream	PrintWriter
Conversion bytes<->chars.	InputStreamReader (byte ->char) OutputStreamWriter (char ->byte)	

Standard streams

- System.in (*InputStream*).
- System.out (*PrintStream*).
- System.err (*PrintStream*).

Buffered streams I

- Buffered input streams reads data from a memory area known as a buffer (an internal array of bytes) and the native input API is called only when the buffer is empty.
- When reading, the data are removed from the buffer rather than the underlying stream. When the buffer runs out of data, the buffered stream refills its buffer from the underlying stream.
- Buffered output streams write data to a buffer, and the native output API is called only when the buffer is full.

Buffered streams II

- Buffered output streams store data in the buffer until this buffer is full or the stream is flushed; then the data is written out to the underlying output stream all at once.
- The read/write operations are more effective, especially in situations when reading/writing several hundred bytes is as fast as reading a single byte from the underlying stream.

Example

streams.Buffered.

Reading from the standard input

```
BufferedReader br = new BufferedReader(new
                        InputStreamReader(System.in));
// or
Scanner input=new Scanner(System.in);
```

Serialization I

- Serialization is a mechanism of representing an object as a sequence of bytes (converting it into a byte stream).
- Deserialization is the reverse process where the byte stream is used to recreate the actual Java object in memory.
- Serialization is used to persist the object.
- The byte stream created is platform independent. Thus the object serialized on one platform can be deserialized on a different platform.
- A persistent (serializable) object is an object that can be written to the respective file / media or read from file or external media.

Serialization II

- Classes whose objects are serializable will implement the *Serializable* interface (java.io package).
- This interface does not contain any method.
- Any object referred by a serializable object must also be serializable.
- The static member fields of a serializable class are ignored during serialization.
- Notable methods:
  - *readObject(): Object*
  - *writeObject(Object)*

Serialization III

- Consider the following problem:
  - Objects of a class are serialized.
  - The class is modified (some fields are deleted, others are added).
  - We want to deserialize the serialized objects.
- For such cases a class a version number should be associated to each serializable class: *serialVersionUID*.
- This is used during deserialization to verify that the sender and receiver of a serialized object have loaded classes that are compatible with respect to serialization.
- An *InvalidClassException* will be thrown if the class of the serialized object has a different serialVersionUID than that of the deserialized object.

Serialization IV

- In some cases we do not want to serialize certain attributes (e.g. passwords).
- These attributes must be declared using the transient keyword.
- At deserialization the attributes declared transient will be initialized with the default value, according to their type.

Example

streams.Serialization.

JUnit I

- JUnit is a free test framework used to write repeatable tests.
- It is an instance of the xUnit architecture for unit testing frameworks.
- A *unit test* is written to assert a certain behaviour or state. It targets a small unit of code (e.g. method or class).
- The percentage of code which is tested by unit tests is typically called *test coverage*.
- Usually unit tests are created in a separate folder.

JUnit II

- A JUnit test is a method contained in a class which is only used for testing. This is called a *Test class*.
- To define that a certain method is a test method, annotate it with the *@Test* annotation.
- Naming convention: the test class usually has *Test* at the end of its name.
- Several test classes can be combined in a *test suite*.
- When a test suite is run all included test classes are run in the specified order.

- Configuring Testing Libraries in IntelliJ

Summary

- Exceptions in Java: checked and unchecked.
- Methods throwing checked exceptions must specify this in their signatures.
- The *java.io* package offers a wide variety of classes for working with streams.
- Serialization is a mechanism of representing an object as a sequence of bytes. Deserialization is the reverse process.
- JUnit is a frameworks for writing repeatable tests.
- *Next week:*
  - JDBC.
  - Java 8 features.