Java API components

Exceptions, file management, serialization

Exception

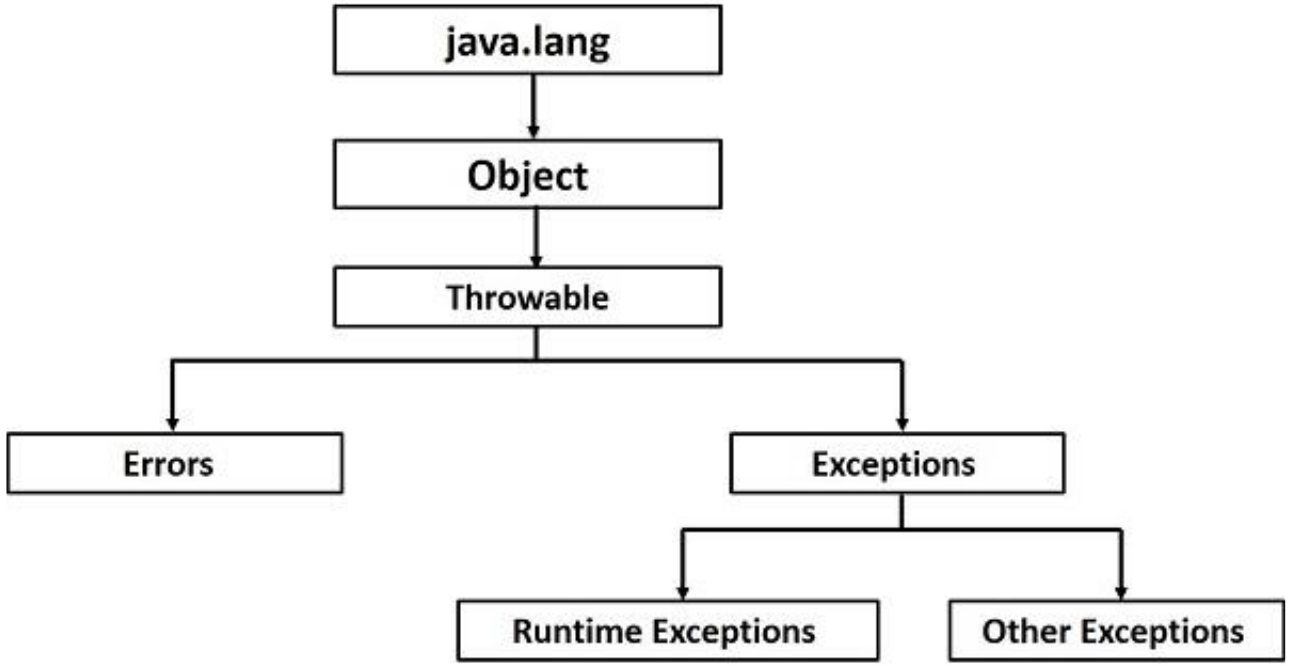
- Exception is an **event** that rises during program execution
- When the **normal flow** of the program is **disrupted**
- Program execution is terminated **abnormally**
- Not recommended from UX point
  - Lose position/data in a process
  - No information about error to correct next time

Reasons of exception

- Programming bug
  IndexOutOfRange, NullPointer
- User entered invalid data
  not checked before use
- Unavailable resource
  File, Network, Memory

Exception hierarchy

- Errors
- Exception objects
  - Unchecked exceptions
  - Checked excetions



Exception methods

- **getMessage –** Returns a detailed message about the exception
- **getCause –** Returns the cause of the exception
- **printStackTrace –** Prints toString() and stack trace to System.err
- **getStackTrace –** Returns an array containing each element on the stack trace
- **fillInStackTrace –** Fills the stack trace of this object with the current stack trace

Stack trace

Method calls

- Parameters are stored in stack
- On return, they are removed

Stack trace is

- Sequence of method calls in the stack
- A path to get from main to the place of error

Stack trace

```
public static void main(String[] args) {
System.out.println(method1()); }
public static int method1() { return method2(); }
public static int method2() { return method3(); }
public static int method3() { return method4(); }

public static int method4() {
    int[] storage = new
    return storage[5];
}
```

```
"C:\Program Files\Java\jdk1.8.0_172\bin\java.exe" ...
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 5
    at Main.method4(Main.java:22)
    at Main.method3(Main.java:17)
    at Main.method2(Main.java:13)
    at Main.method1(Main.java:9)
    at Main.main(Main.java:5)

Process finished with exit code 1
```

Exception handling

```
try {
    // Protected code
} catch (ExceptionType exceptionName) {
    // Catch block
}
```

Exception handling

```
public static void main(String args[]) {
   try {
        int a[] = new int[5];
        System.out.println("Access element:" + a[5]);
   } catch (ArrayIndexOutOfBoundsException e) {
        System.out.println("Exception thrown  :" + e);
   }
}
```

Handling exceptions differently

```
try {
    // Protected code
} catch (ExceptionType1 exceptionName1) {
    // Catch block to handle exceptions of ExceptionType1
} catch (ExceptionType2 exceptionName2) {
    // Catch block to handle exceptions of ExceptionType2
} catch (ExceptionType3 exceptionName3) {
    // Catch block to handle exceptions of ExceptionType3
}
```

Handler for descendants

```
try {
    // Protected code
} catch (Exception ex) {
    // Catch block for Exception
} catch (IOException ioEx) {
    // Catch block for IOException
} catch (FileNotFoundException fnfEx) {
    // Catch block FileNotFoundException
}
```

java.io

Class FileNotFoundException

java.lang.Object
- java.lang.Throwable
  - java.lang.Exception
    - java.io.IOException
      - java.io.FileNotFoundException

Handling multiple exceptions

```
try {
   // Protected code
} catch (ExceptionType1 exceptionName1) {
   // SAME catch block to handle exceptions -> use method
} catch (ExceptionType2 exceptionName2) {
   // SAME catch block to handle exceptions -> use method
}
```

Handling multiple exceptions – Java 7

```
try {
    // Protected code
}
catch (ExceptionType1 | ExceptionType2 exceptionName1)
{
    // Single catch block to handle exceptions
}
```

Code after exception handling

Usually used to release allocated resources.

```
try {
    // Protected code
} catch (ExceptionType exceptionName1) {
    // Catch block
} finally {
        // Block runs after try-catch, regardless of result
}
```

Code after exception handling – difference?

```
try {
    // Protected code
} catch (ExceptionType exceptionName1) {
    // Catch block
} finally {
        // Finally block
}
```

```
try {
    // Protected code
} catch (ExceptionType exceptionName1) {
    // Catch block
}
// Block after try-catch
```

Handling exceptions with resource

The allocated resource will be released after try, no need for finally.

```
try(<resource allocation>) {
   // Protected code
} catch (ExceptionType1 exceptionName1) {
   // Catch block to handle exceptions
}
```

Handling exceptions with resource

```
try(FileReader fr = new FileReader("file.txt")) {
   fr.read(a);
} catch (IOException e) {
   e.printStackTrace();
} finally {
      try {
            fr.close();
      } catch (IOException ex) {
            ex.printStackTrace();
      }
}
```

No need for finally

Throwing exception

Using `throw` keyword – `throw exceptionInstance;`

- When detecting an execution problem – newly created exception instance
- When partially handling an exception – previously caught exception object

Throw new exception

```
public int getItem(int index) <throws see later> {
   if(index < storage.length) {
      return storage[index];
   } else {
      throw new IndexOutOfRangeException();
   }
}
```

Partial exception handling

```
public int sumOfItems(int[] indices) <throws see later> {
   for(int i=0; i < indices.lenght; i++) {
      try {
            sum += getItem(indices[i]);
      } catch (IndexOutOfRangeException ex) {
            // partial handling of the exception
            throw ex;
      }
   }
}
```

Checked exception

- Defined and thrown by the programmer
- Handling is checked by the **compiler**
- Also called compile time exceptions – but they are not indeed
- Programmer can not ignore them – they need to be **handled**

Handling checked exceptions

Checked exceptions has to handled

- Explicitly by **`try-catch-finally`**
- When not handling explicitly, propagate to the caller by marking the method with **`throws`** keyword
- When exception is thrown, it is not handled → **`throws`** required

Throwing new exception

```
public int getItem(int index) throws IndexOutOfRangeException {
   if(index < storage.length) {
      return storage[index];
   } else {
      throw new IndexOutOfRangeException();
   }
}
```

Propagation without handling

```
public int calcSum() throws IndexOutOfRangeException {
   int sum = 0;
   for(int i=0; i<itemCount; i++) {
      sum += getItem(i);
   }
   return sum;
}
```

File management

Read and write binary and text data

File management

- File is the place for storing application data
- Stores primitive types
- Directly or indirectly, but files are used
- Outer resource, handler can throw checked exceptions

- Design can be generalized

File

- Infinite sequence of **bytes**
- Sequence is **serial**, no direct access to elements

- Before writing, after reading, data can be handled differently
- Objects have to be **converted** to sequence of bytes → ***Serialization***
- Sequence of bytes has to be converted to Objects → ***Deserialization***

Reading file – by all bytes

```
try {
   byte[] contents = Files.readAllBytes(
      Paths.get(fileName)
   );
} catch (IOException e) {
      e.printStackTrace();
}
```

Reading file – by all lines

```
try {
    String[] contents = Files.readAllLines(
        Paths.get(fileName),
        StandardCharsets.UTF_8
    );
} catch (IOException e) {
        e.printStackTrace();
}
```

Reading file – by bytes

```
try {
   FileReader fr = new FileReader(fileName);
   int i;
   while ((i = fr.read()) != -1) {
     System.out.print((char) i);
   }
   fr.close();
} catch (IOException e) {
      e.printStackTrace();
}
```

Reading file – by lines

```
try {
    BufferedReader br = new BufferedReader(
        new FileReader( new File(fileName) )
    );
    String st;
    while ((st = br.readLine()) != null)
        System.out.println(st);
    }
    br.close();
} catch (IOException e) {
        e.printStackTrace();
}
```

Reading file – Scanner

```
// Scanner does not throw checked exception
Scanner sc = new Scanner(new File(fileName) );

while (sc.hasNextLine()) {
   System.out.println(sc.nextLine());
}
```

Reading file

- Static members of Files class – full content
- FileReader instance – sequence of bytes
- Buffered reader – line formatting in buffer
- Scanner – any formatting

Writing file – split write text

```
try {
    FileWriter writer = new FileWriter(new
    File(fileName));
    writer.write(textContent);
    writer.flush();
    writer.close();
} catch (IOException e) {
        e.printStackTrace();
}
```

Writing file – buffered write

```
try {
   BufferedWriter writer = new BufferedWriter(
      new FileWriter(fileName)
   );
   writer.write(textContent);
   writer.close();
} catch (IOException e) {
      e.printStackTrace();
}
```

Writing file – full content

```
try {
      Files.write(
            Paths.get(fileName),
            byteContents
      );
} catch (IOException e) {
      e.printStackTrace();
}
```

Writing file – binary output stream

```
try {
   FileOutputStream outputStream = new
   FileOutputStream(fileName);
   outputStream.write(byteContents);
   outputStream.close();
} catch (IOException e) {
      e.printStackTrace();
}
```

Writing file – buffered binary output stream

```
try {
   FileOutputStream fos = new
   FileOutputStream(fileName);
   BufferedOutputStream bos = new
   BufferedOutputStream(fos);
   bos.write(binaryData, 0, binaryData.length);
   bos.close();
} catch (IOException e) {
      e.printStackTrace();
}
```

Writing file – binary data output stream

```
try {
    FileOutputStream fos = new
    FileOutputStream(fileName);
    BufferedOutputStream bos = new
    BufferedOutputStream(fos);
    DataOutputStream outStream = new
    DataOutputStream(bos);
    outStream.writeUTF(stringContents);
    outStream.write(intData);
    outStream.close();
} catch (IOException e) {
        e.printStackTrace();
```

Serialization

Serialization / Deserialization

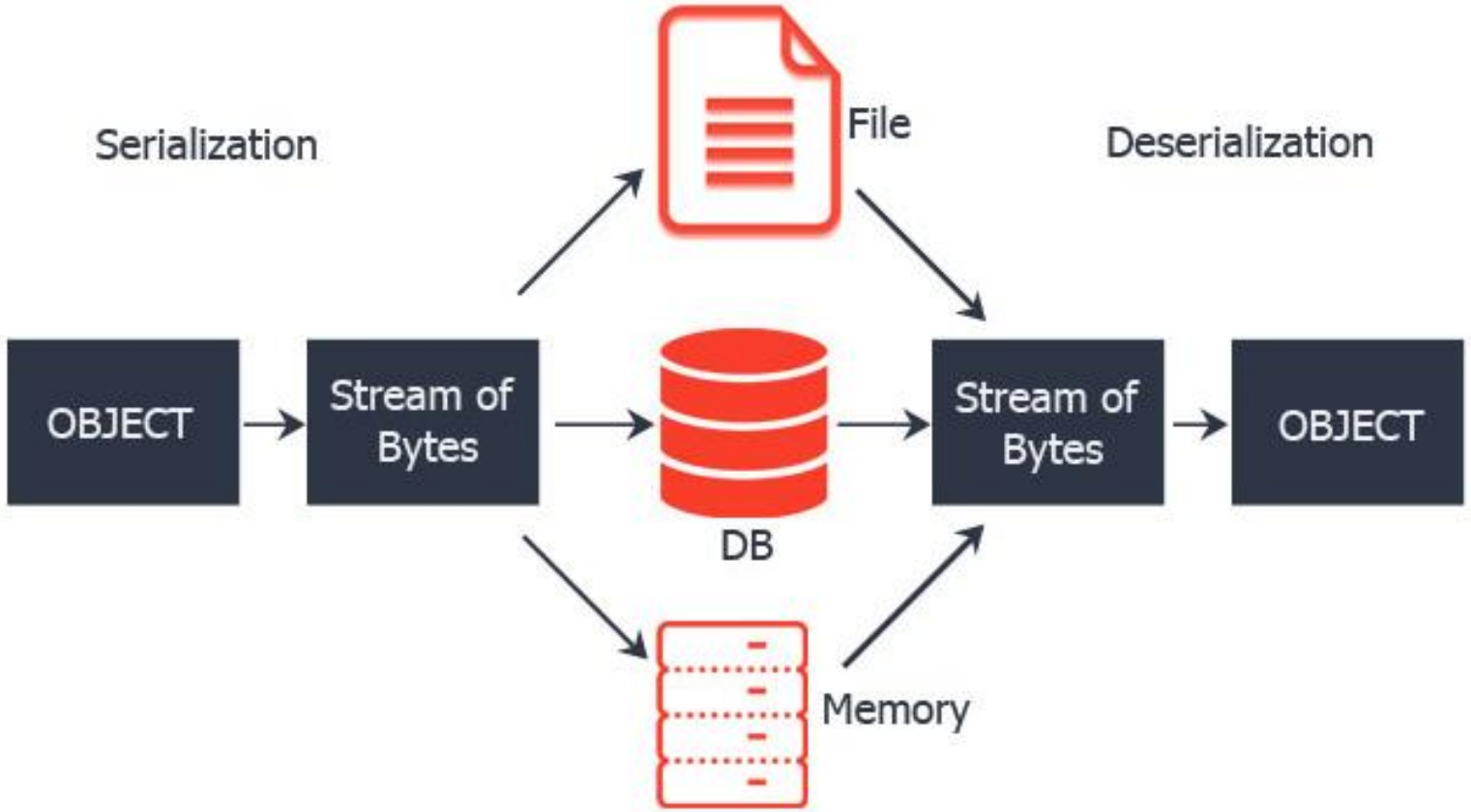
A mechanism of converting the **state of an object** into a byte stream.

This mechanism is used to persist the object.

Deserialization is the reverse process, used to **restore a persisted object**.

Object has to be **serializable**

Serialization - deserialization



Serializable

- Object state is serialized – `static` and `final` are not part of object state
- To set an object as serializable, implement `Serializable` interface
- Serializable interface has no data member nor method
- Only a marker interface, used by JVM
- `transient` properties are not serialized
- JVM does a lot of magic behind the scenes to get and set state descriptors.

Serializable

```
public class MyClass implements Serializable {
        public int num1;
        private int num2;
        public transient int num3;

        public int getNum2() { return num2; }
        public void setter(int num2) { this.num2 = num2; }
}
```

Serialization example

```
ByteArrayOutputStream baos = new ByteArrayOutputStream();
ObjectOutputStream oos = new ObjectOutputStream( baos );
oos.writeObject( o );
oos.close();
String stateString = Base64.getEncoder().encodeToString(
      baos.toByteArray()
);
```

rO0ABXNyAAdNeUNsYXNzIcNnJpD1hyICAAJJAARudW0xSQAEbnVtMnhwAAAACgAAABQ=

Deserialization example

```
byte [] data = Base64.getDecoder().decode(stateString );
ObjectInputStream ois = new ObjectInputStream(
      new ByteArrayInputStream( data )
);
Object object  = ois.readObject();
ois.close();
```

JavaScript Object Notation

- Coming from JavaScript, but language independent
- Text based descriptor
- Readable for humans
- Shorter than XML, more readable than simple text data
- No structure descriptor
- Collection of objects (associative arrays) and arrays
- Valid data: `number`, `string`, `boolean`, `array`, `object`, `null`

JSON example

```
{
    ”lastName": ”Smith",
    ”firstName": ”Agent",
    ”age": 22,
    „address": { „street": „Boszorkany street 2.", ”city": „Pecs", ”postalCode": „7624"},
    ”phone": [
            {”type": ”home",”number": ”+36201234567"},
            { ”type": „office", ”number": „+36301234567"}
    ]
 }
```

Using JAR files

- ZIP formatted file
- Contains Java classes and metadata
- Can contain manifest file to describe usage of contents
- Can contain executable program

- JAR files have to be added to project as dependencies

Google JSON serialization

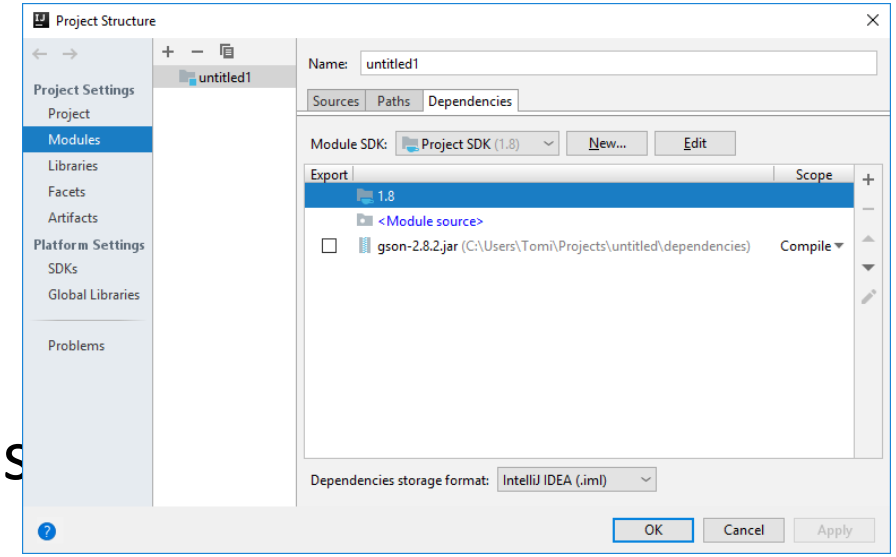
- Download gson-2.8.2.jar

  https://repo1.maven.org/maven2/com/google/code/gson/gson/2.8.2/

- Add gson.jar to project as dependency

  File/Project structure…/Modules/Dependencies

- Create instance of Gson class
- Use this instance to serialize object to json
- Use this instance to deserialize object from js



MyClass

```
public class MyClass
                implements Serializable {
    public int num1;
    private int num2;
    public transient int num3;
    public MyClass(int num1, int num3) {
        this.num1 = num1;
        this.num3 = num3;
    }
}
```

```
public void setter() {
    num2 = 20;
}
public int getNum2() {
    return num2;
}
```

GSON example

```
MyClass c1 = new MyClass(10, 30);
c1.setter();

Gson gson = new Gson();
String jsonString = gson.toJson(c1);

MyClass c3 = gson.fromJson(jsonString, MyClass.class);
```

Homework

First implementation

Homework

1. Download the first available version of the Poker framework (https://drive.google.com/file/d/1San5R6Fv11KQsP9Lt0DvAVMmawSaLgoj/view?usp=sharing)
2. Add JAR to the project as dependency
3. Create players which implement Player interface – Console, automatic
4. Create an instance of the Party
5. Add players – both types
6. Call Play method
7. Check player internal states by debugging

Cards

- Colors:
  1 character: c-clubs, h-hearts, d-diamonds, s-spades
- Values:
  1 character: 2,3,4,5,6,7,8,9,t,j,q,k,a

Representation examples: c7, d2, s8, ht, dj, sk, cq, sa

Player interface

- void deal(String cards, int numberOfPlayers)
  - cards: contains 2 cards ”, ” (coma and space) separated
  - numberOfPlayers: number of players still playing
- boolean areYouIn() – if no, player folds
- void flop(String cards , int numberOfPlayers) – contains 3 cards
- void turn(String cards , int numberOfPlayers) – contains 1 card
- void river(String cards , int numberOfPlayers) – contains 1 card
- void result(boolean youWon, int numberOfBetsWon)

Play method

1. Deal
2. Are you in? If yes, pay 1 bet
3. Flop
4. Are you in? If yes, pay 1 bet;
5. Turn
6. Are you in? If yes, pay 1 bet;
7. River
8. Are you in? If yes, pay 1 bet;
9. Result

Initial chips are not counted, as many parties can be played as wanted.

Even with the same party object.