Lab 6. Files, Serialization and Exceptions

> Authors: Yida Tao
>
> Reference: Core Java Volume I. Cay S. Horstmann https://www.baeldung.com/java-serial-version-uid

Working with Files

Please download `FilesExample.java`. You should execute the code and make sure that you understand each of the output.

To fully understand Java File I/O mechanism and available APIs, please also refer to the official documentation.

Serialization

Suppose we have a serializable `Student` class:

```
class Student implements Serializable {
    String name;

    @Override
    public String toString(){
        return "Student name: " + name;
    }
}
```

We could instantiate a `Student` instance and serialize it as:

```
  public static void main(String[] args) throws IOException,
ClassNotFoundException {
      Student student = new Student();
      student.name = "alice";

      String str = serialize(student);
      System.out.println("Serialized: " + str);

  }

  public static String serialize(Serializable o) throws IOException {
      ByteArrayOutputStream baos = new ByteArrayOutputStream();
      ObjectOutputStream oos = new ObjectOutputStream(baos);
      oos.writeObject(o);
      oos.close();

      return Base64.getEncoder().encodeToString(baos.toByteArray());
  }
```

Executing the above code gives us the byte stream of the `Student` instance `str`. We could deserialize the byte stream and get back the same instance content.

```
  public static void main(String[] args) throws IOException,
ClassNotFoundException {
      ......
      Student str1 = (Student)
deserialize("rO0ABXNyABV0dXRvcmlhbC5sYWI2LlN0dWRlbnR/Q3EPkPqElgIAAUwABG5hb
WV0ABJMamF2YS9sYW5nL1N0cmluZzt4cHQABWFsaWNl");
      System.out.println("Deserialized: " + str1);
  }


  public static Object deserialize(String s)
          throws IOException, ClassNotFoundException {

      byte[] data = Base64.getDecoder().decode(s);
      ObjectInputStream ois = new ObjectInputStream(
              new ByteArrayInputStream(data));
      Object o = ois.readObject();
      ois.close();
      return o;
  }
```

- Case 1: Let's add a new field, `int age`, to the `Student` class, and re-execute the **deserialization** process. Observe the result.
- Case 2: Let's add `private static final long serialVersionUID = 1L` to the `Student` class. Repeat the serialization process, and use its output as the input of deserialization process.
    - Add a new field, `int age`, to the `Student` class, along with modification to the `toString()` method. Then repeat the deserialization process and observe the result.
    - Add a new field, `int age=20`, to the `Student` class, along with modification to the `toString()` method. Then repeat the deserialization process and observe the result.
    - Remove an exisiting field `name`, along with modification to the `toString()` method. Then repeat the deserialization process and observe the result.

Basically, if we don't define a `serialVersionUID` state for a `Serializable` class, then Java will define one based on some properties of the class itself such as the class name, instance fields, and so on. However, some changes to this class (e.g., adding a new field) may break the serialization compatibility, causing `InvalidClassException`. Because of this sort of unwanted incompatibility, it's always a good idea to declare a `serialVersionUID` in `Serializable` classes.

If we already declared a `serialVersionUID` in `Serializable` classes, if we added a new field, then during the deserialization process, default value will be used for that new field. If we deleted a field, then during the deserialization process, the value for that field is simply ignored.

Exception Flow

Download and modify `ExceptionDemo.java` as follows. Observe how the results differed.

- Case 1: The code throws no exceptions. In this case, the program first executes all the code in the `try` block. Then, it executes the code in the `finally` clause. Afterwards, execution continues with the first statement after the `finally` clause.

```
    try
    {
        InputStream in = new FileInputStream("exist-file");
        System.out.println("End of try.");
    }
    catch (IOException e)
    {
        System.out.println("Catch begins.");
    }
    finally
    {
        System.out.println("Finally.");
    }
    System.out.println("After finally.");
```

- Case 2a: The code throws an exception that is caught in a `catch` clause - in our case, an `IOException`. For this, the program executes all code in the `try` block, up to the point at which the exception was thrown. The remaining code in the `try` block is skipped. The program then executes the code in the matching `catch` clause, and then the code in the `finally` clause. If the `catch` clause does not throw an exception, the program executes the first line after the `finally` clause.

```
    try
    {
        InputStream in = new FileInputStream("nonexist-file");
        System.out.println("End of try.");
    }
    catch (IOException e)
    {
        System.out.println("Catch begins.");
    }
    finally
    {
        System.out.println("Finally.");
    }
    System.out.println("After finally.");
```

- Case 2b: If the `catch` clause throws an exception, then the exception is thrown back to the caller of this method after `finally` clause executes.

```
public static void main(String[] args) throws FileNotFoundException {

    try
    {
        InputStream in = new FileInputStream("nonexist-file");
        System.out.println("End of try.");
    }
    catch (IOException e)
    {
        System.out.println("Catch begins.");
        InputStream in = new FileInputStream("nonexist-file");
        System.out.println("End of catch");
    }
    finally
    {
        System.out.println("Finally.");
    }
    System.out.println("After finally.");

}
```

- Case 3: The code throws an exception that is not caught in any `catch` clause. Here, the program executes all code in the `try` block until the exception is thrown. The remaining code in the try block is skipped. Then, the code in the `finally` clause is executed, and the exception is thrown back to the caller of this method.

```
        try
        {
            InputStream in = new FileInputStream("exist-file");
            String s = null;
            s.length();

            System.out.println("End of try.");
        }
        catch (IOException e)
        {
            System.out.println("Catch begins.");
            InputStream in = new FileInputStream("nonexist-file");
            System.out.println("End of catch");
        }
        finally
        {
            System.out.println("Finally.");
        }
        System.out.println("After finally.");

    }
```

Method Call Chain and Stack Trace

A stack trace is a listing of all pending method calls at a particular point in the execution of a program. You have almost certainly seen stack trace listings — they are displayed whenever a Java program terminates with an uncaught exception.

You could use the `StackWalker` class that yields a stream of `StackWalker.StackFrame` instances, each describing one stack frame.

The `StackWalker.StackFrame` class has methods to obtain the file name and line number, as well as the class object and method name, of the executing line of code. The `toString` method yields a formatted string containing all of this information.

Run `StackTraceTest.java` to print the stack trace of a recursive function.