Java Input/Output API :-

Java I/O → Java NIO → Java NIO 2
Java 1 (1996) → Java 4 (2002) → Java 7 (2011)

→ Now we discussing Java I/O

Java I/O → How to read write information
- bytes and chars
- on many media : disk, network, memory

The API → collection of classes is little complex
Built on the decorator pattern.

Java I/O is organized on four base classes
- Reader and Writer
- InputStream and OutputStream

And two utility classes:
- File
- Path → interface not a class.

```
File file = new File("files/data.txt");
```

The File is a class.
1) creating a File object does not create anything on the disk
2) a File object can be a file or directory
file.exists().

```
file.isFile();
file.isDirectory();
file.canRead();
file.canWrite();
file.canExecute();
    :
```

There 30 methods

relative path of files/data.txt → files

absolute path of /tmp/files/data.txt → /tmp/files

. → represents current directory

.. → represents parent of current dir-

→ java7 interface

The path interface has same kind of methods as the File class

Reading Characters:-

Introducing ~~char~~ Readers:-

The Reader is an abstract class

it defines the basic operations like

→ Reading of a single character

→ Reading of an array of characters

→ Marking and resetting a given position

→ skipping positions

And it can be ...

```
Reader reader = ~~new~~ ~~Read~~ = ..... ;
    int nextChar = reader.read();
// When there is no more characters to read,
// the read() call returns -1
        while (nextChar != -1) {
            // do something with next char
            nextchar = reader.read();
        }
```

To Read ~~An~~ an array characters.

```
    char[] buffer = new char[1024];
    int number = reader.read(buffer);
        while(number != -1)
        { ~~read~~ number = reader.read(buffer);
        }
```

it can read up to 1024 or less than 1024

Dealing With Exceptions:-

```
Reader reader = null;
try {
    reader = ..... ;
    int nextChar = reader.read();
    while (nextchar != -1) {
        ~~next~~char = reader.read();
    }
} catch (IOException e) { }
```

Closing a Reader :-

We know that all system resources must be properly closed otherwise there may be a resource leak and application will be crashed.

A Reader uses a system resource so it must be properly closed.

There are two patterns for that

- call the close() method
- Use the try-with-resource pattern
  ↙ available in java 7

```
Reader reader = null;
try {
    reader = ---;
    :
} catch (IOException e) {
}
finally {
    if (reader != null)
    {
        try {
            reader.close();
        }
        catch (IOExceptione)
        {
        }
    }
}
```

We have to check this condition because if Except rised at first reader line and we try to close reader in finally block gives null pointer exception

try-with-resource

```
try (Reader reader. ---- ;) {
    ==
} catch (IOException e) {
    ==
}
```

To be used in this pattern, a resource must implement AutoClosable with only one method to implement : close(),

Marking, Resetting and Skipping :-

mark() call puts a flag on a given element

reset() call rewinds to the poeviously marked element, or the begining of the stream

skip() call skips the next elements.

Two ways of extending Reader class since it is a Abstract class :-

- 2 categories of concrete classes

1) classes for a certain type of input

Disk : FileReader

In-Memory : CharArrayReader, StringReader

2) classes that add behavior to Reader

- BufferedReader
- LineNumberedReader

- Append a single char or a string

And it can be closed.

Writer writer = ... ; // we will see later

writer.write('H');
writer.write("Hello World");

- The write() method does not return anything

it can also take precision and length;

writer.write("HelloWorld", 0, 5);
↳ it write 'Hello' only

all exceptions, patterns are same as Reader

- Disk : FileWriter
- In-Memory : CharArrayWriter
  fake - StringWriter
- BufferedWriter
- write with a format : PrintWriter

→ BW extends Writer And is built on an instance of Writer.

```
File file = new File("files/data.txt");
FileWriter fileWriter = new FileWriter(file);
BufferedWriter bWriter1 = new BufferedWriter(fileWriter);
```

→ This is for file writing with the UTF-8 charset

But one can also pass other charsets

```
Path path = Paths.get("files/data.txt");
BufferedWriter bWriter2 =
Files.newBufferedWriter(Path, StandardCharsets.ISO_8859_1);
```

Java 7 also brough richer patterns to open files for writing

StandardOpenOption is an enumeration that implements OpenOption

```
Path path = Paths.get("files/data.txt");
BufferedWriter bufferedWriter =
Files.newBufferedWriter(Path, StandardOpenOption.APPEND);
```

APPEND → Writes data if file exists otherwise creates file and write data

CREATE → create new file, if file already exists then throw Ex

DELETE_ON_CLOSE → deletes after closing

two limitations on the GZIP format

→ it can compress one file at a time

→ there may be a limit on the file size

Reading and Writing Data and Objects:-

Concept of Serialization

Create binary images of objects

How to override the standard mechanism

→ Why Serializations

Serializing Objects:-

→ It is about creating a portable representation of an object.

↙

It means binary file or text file - that can be stored on disk, fore later use or sent over a network to another application

```
public class Person{
private string Name;
private int age;
}
```

=>

XML

```
<Person>
  <name>Sarah</name>
<age>32</age>
</person>
```

JSON

```
{ "Person": {
    "name": "sarah",
    "age" : 32
  }
}
```

The java person class, only java code can use it

→ The instance of person, in XML. it is portable

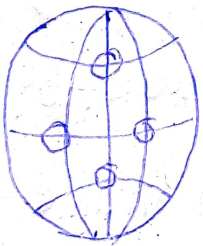
→ The JSON instance also portable

portable → any other application can access the data
(written JS, C#...)

→ XML and JSON are two ways of serializing Java objects in a portable way

But
XML is first publish in 1998
JSON in 2013 - 2014
Java in 1995

so java has it's own serialization mechanism.



Shopping Service java

Invoice Service java

requires person object

→ 000

000 →

?

serialized object

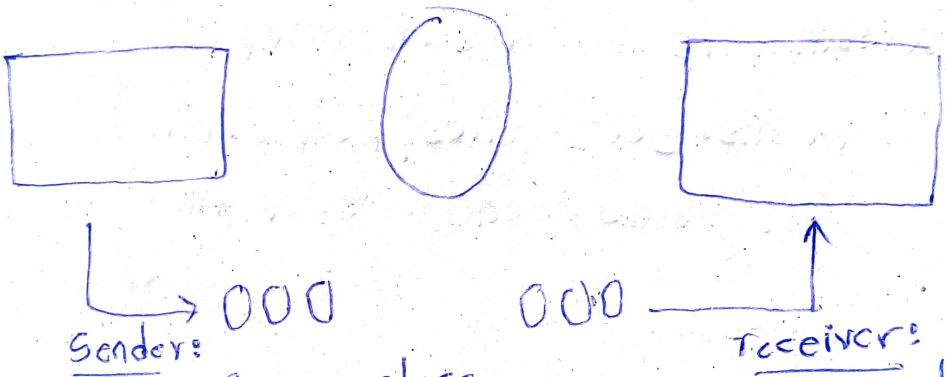
Consists: state
name of the class
The version of the class

How serial version UID is computed ? by JVM

It is a hash computed from the class Name, interfaces implemented, methods and fields using a SHA.

if the field is present in the class at compile time, then it will be used as, is with no-validation

Deserialization :-



case 1 :

✓ Person class
UID is 2L

Person class
UID is 2L

→ so it can be easily deserialized

case 2

✗ Person Class
UID is 2L

person class
UID is 3L

→ not deserializable

case 3

✓ Person class
UID is 2L

Person class (NOT same → some field are changed)
UID is 2L
will deserializable