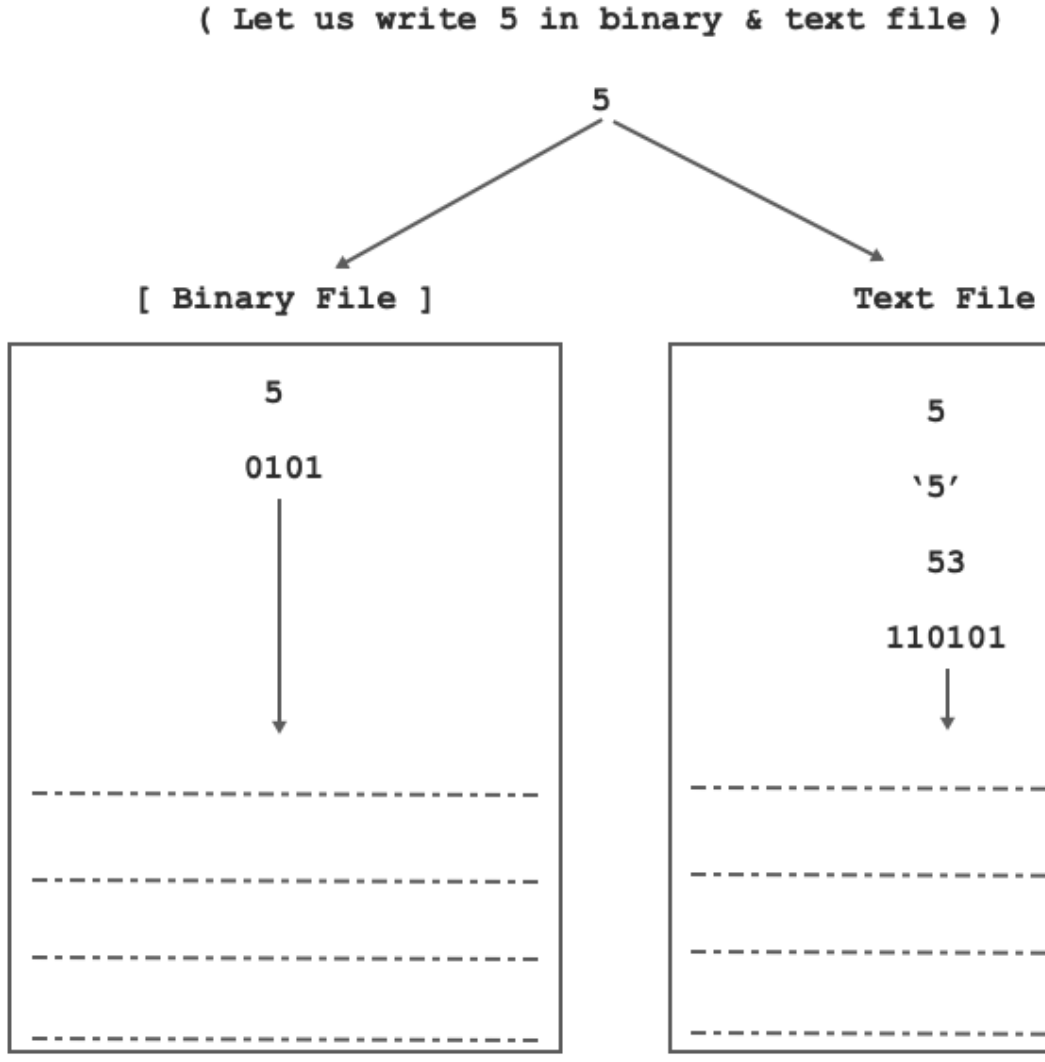
File Input and Ouput Operation in Java

- Variable
  - It is temporary conrainter which is used to store record in primary memory(RAM).
- File
  - It is permenant conrainter which is used to store record on HDD.
  - Types of files:



[Binary File versus Text File]

- Text File
  - Examples: .c, .cpp, .java. .cs, .html, css. .js, .txt, .doc, .docs, .xml, .json etc.
  - We can read text file using any text editor.
  - It requires more processing than binary file hence it is slower in performance.
  - If we want to save data in human readable format then we should create text file.
- Binary File
  - Examples: .jpg, .jpeg, .bmp, .gif, .mp3, .mp4, .obj, .class etc.
  - To read binary file, we must use specific program.
  - It requires less processing than text file hence it is faster in performance.
  - If we dont want to save data in human readable format then we should create binary file.

- Stream
  - It is an abtraction(instane) which either consume(read) or produce(write) information from source to destination.
  - Stream is always associated with resource.
  - Standard stream instances of Java programming languages which are associated with Console(Keyboard / Monitor):
    - System.in
    - System.out
    - System.err

- If we want to save data in file the we should use types declared in java.io package.

- java.io.Console class reprsents Console.

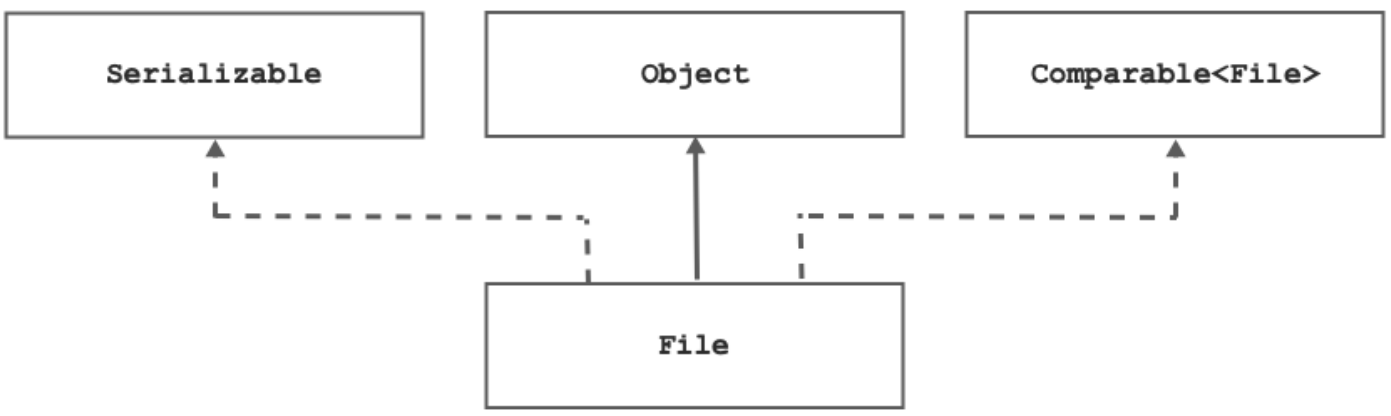
```
import java.io.Console;

public class Program {
  public static void main(String[] args) {
    System.out.print("Enter name  : ");
```

```
        Console  console = System.console();
        String name = console.readLine();
        //System.out.println("Name  : "+name);
        console.printf("Name  : %s\n", name);
      }
    }
```

- java.io.File class reprsents Physical file on HDD.

File

- It is a class declared in java.io package.



- We can use java.io.File class:
  - To create new file or to remove exisiting file.
  - To create new directory or to remove exisiting directory.
  - To read metadata of file, directory or drive.
- Constructor:
  - public File(String pathname)

```
String pathname = "Sample.txt";
File file = new File( pathName );
```

- Create new file:

```
public static void main(String[] args) {
try {
  String pathName = "Sample.txt";
  File file = new File(pathName);
  if( file.exists())
    System.out.println("File already exist.");
  else {
    boolean status = file.createNewFile();
    System.out.println("File creation is successful.");
  }
} catch (IOException e) {
  e.printStackTrace();
}
}
```

- Remove exisiting file:

```
  public static void main(String[] args) {
  try {
    String pathName = "Sample.txt";
    File file = new File(pathName);
    if( !file.exists())
      System.out.println("File does not exist.");
    else {
      boolean status = file.delete();
      System.out.println("File deletion is successful.");
    }
  } catch (Exception e) {
    e.printStackTrace();
  }
}
```

- Create new directory

```
public static void main(String[] args) {
  try {
    String pathName = "Sample";
    File file = new File(pathName);
    if( file.exists())
      System.out.println("Directory already exist.");
    else {
      boolean status = file.mkdir();
      System.out.println("Directory creation is successful.");
    }
  } catch (Exception e) {
    e.printStackTrace();
  }
}
```

- Remove existing directory

```
public static void main(String[] args) {
  try {
    String pathName = "Sample";
    File file = new File(pathName);
    if( !file.exists())
      System.out.println("Directory does not exist.");
    else {
      boolean status = file.delete();
      System.out.println("Directory deletion is successful.");
    }
  } catch (Exception e) {
    e.printStackTrace();
  }
}
```

- Read Metadata of files:

```
public static void main(String[] args) {
  try {
    //String pathName = "D:\\Users\\sandeep\\CDAC\\Quiz.txt"; //Windows
    String pathName = "/Users/sandeep/Desktop/CDAC/Quiz.txt"; //Linux
    File file = new File(pathName);
    if( file.exists()) {
      System.out.println( "File Name  :   "+file.getName());  //Quiz.txt
      System.out.println("Parent Directory:   "+file.getParent());
      System.out.println("Length      :   "+file.length());
      System.out.println("Last Modified   :   "+new SimpleDateFormat("dd MMM,yyyy hh:mm:ss").format(new
Date(file.lastModified())));
    }
  } catch (Exception e) {
    e.printStackTrace();
  }
}
```

- Read Metadata of directory:

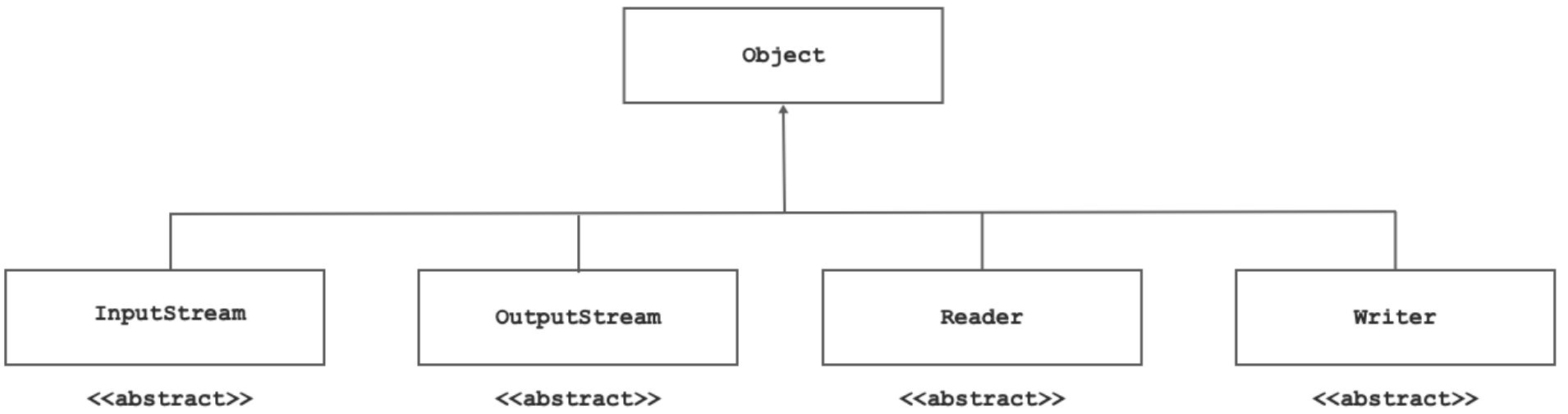
```
public static void main(String[] args) {
  try {
    String pathName = "/Users/sandeep/Desktop/CDAC/";
    File file = new File(pathName);
    if( file.exists() && file.isDirectory()) {
      //String[] nameList = file.list();
      File[] files = file.listFiles();
      for (File f : files) {
        if( !f.isHidden())
          System.out.println(f.getName());
      }
    }
  } catch (Exception e) {
    e.printStackTrace();
```

```
    }
  }
```

Assignment: Write a code in java to simulate windows command prompt?

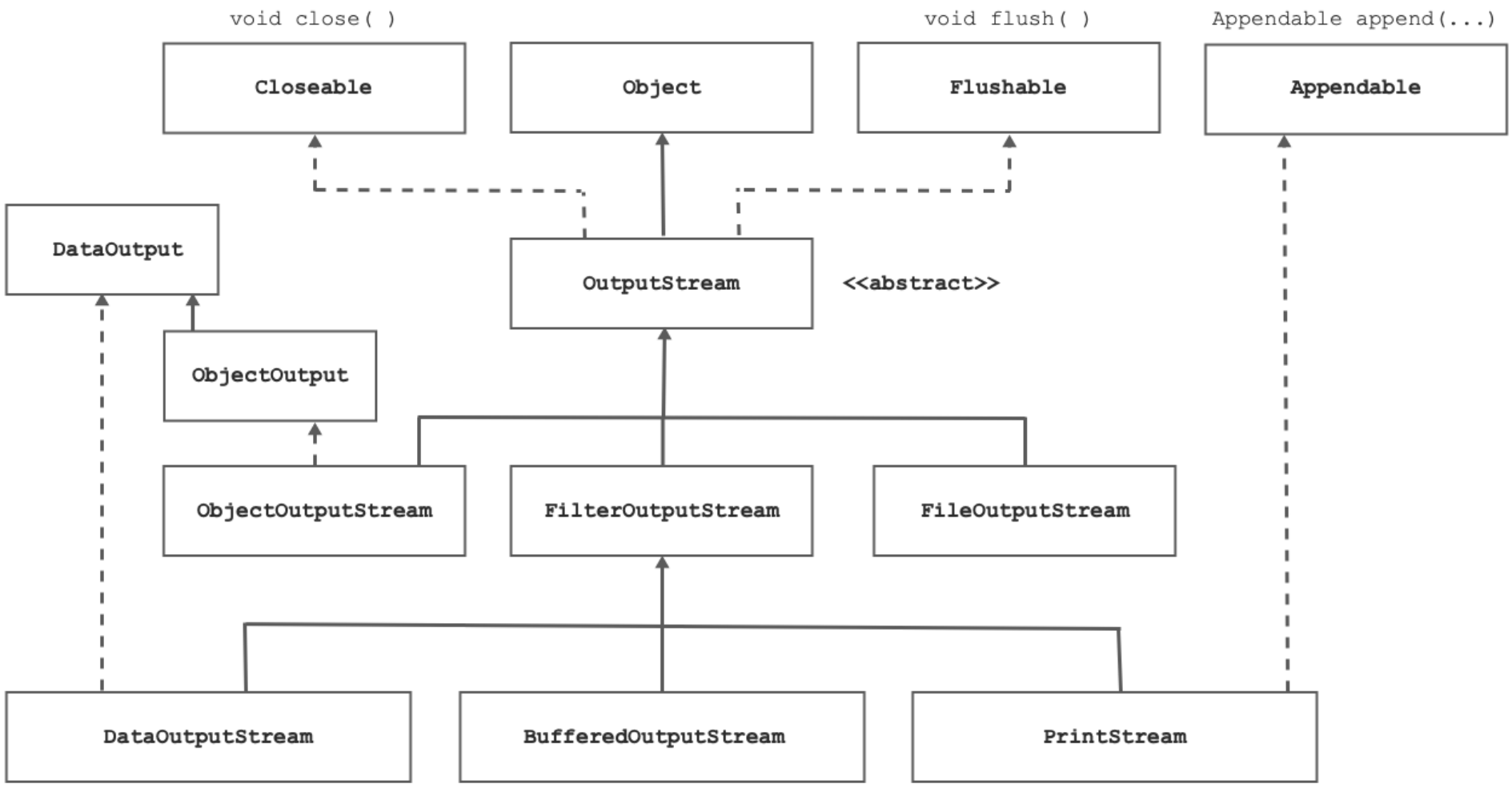
```
Enter path:\>
If it is file then display File metadata
If it is Directory then display File, Directory and Drive metadata
```

File data manipulation

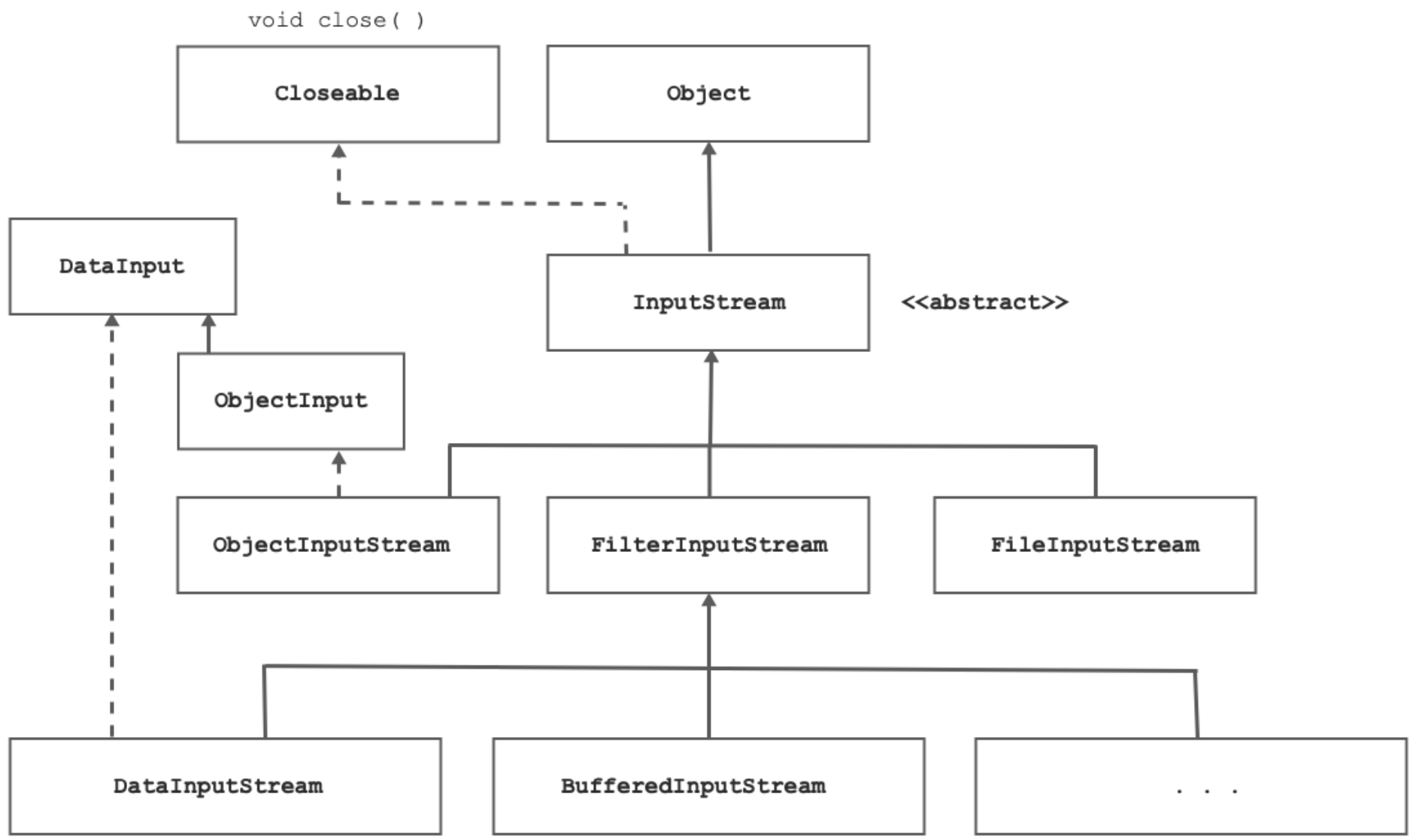


- Interfaces:
    - FileFilter
    - FilenameFilter
    - Closeable
    - Flushable
    - DataInput
    - DataOutput
    - ObjectInput
    - ObjectOutput
    - Serializable
    - Externalizable
- If we want to read/write data into binary file then we should use InputStream, OutputStream and their sub classes.

- Consider OutputStream hierarchy:

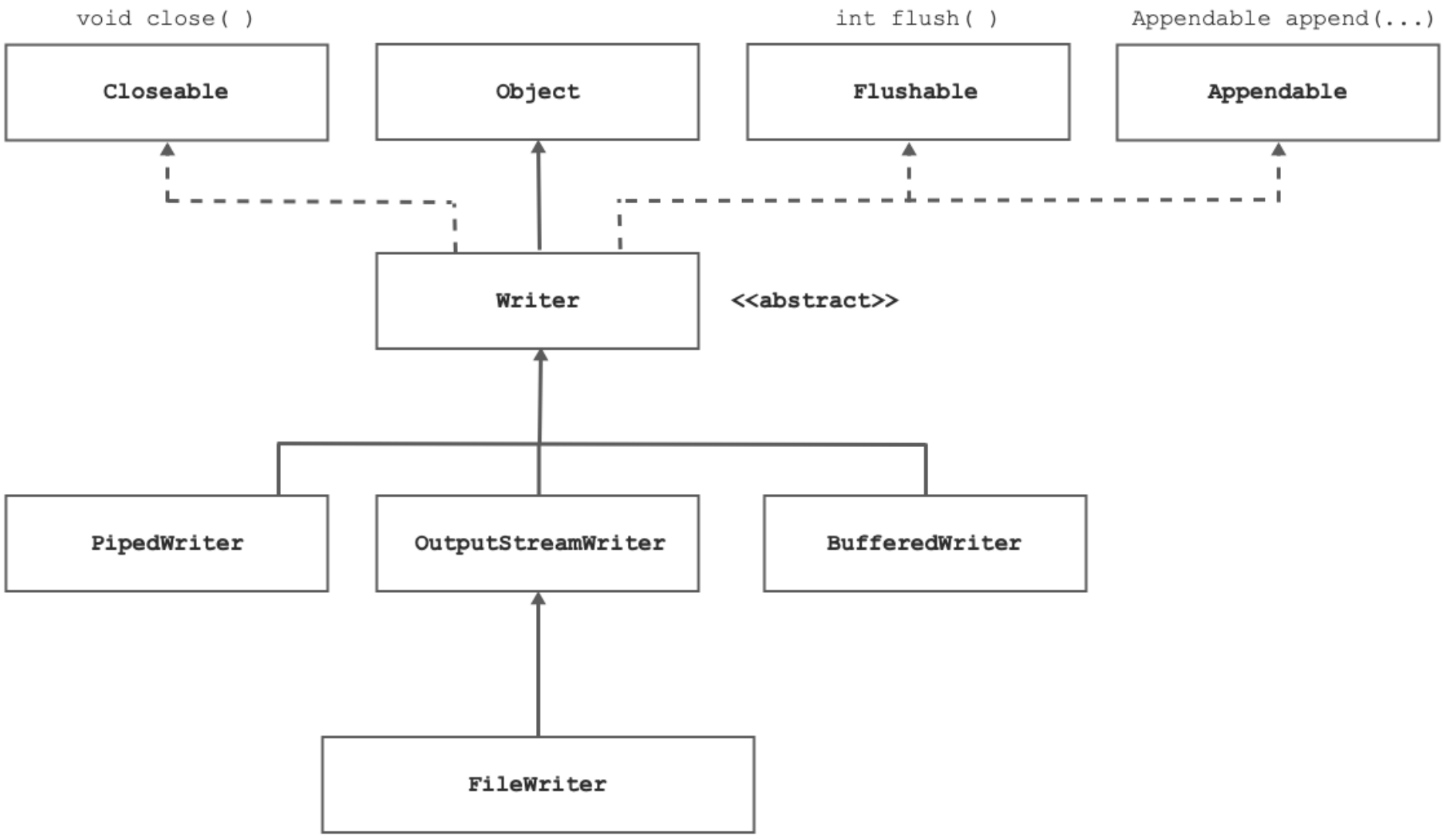


- Consider InputStream hierarchy:

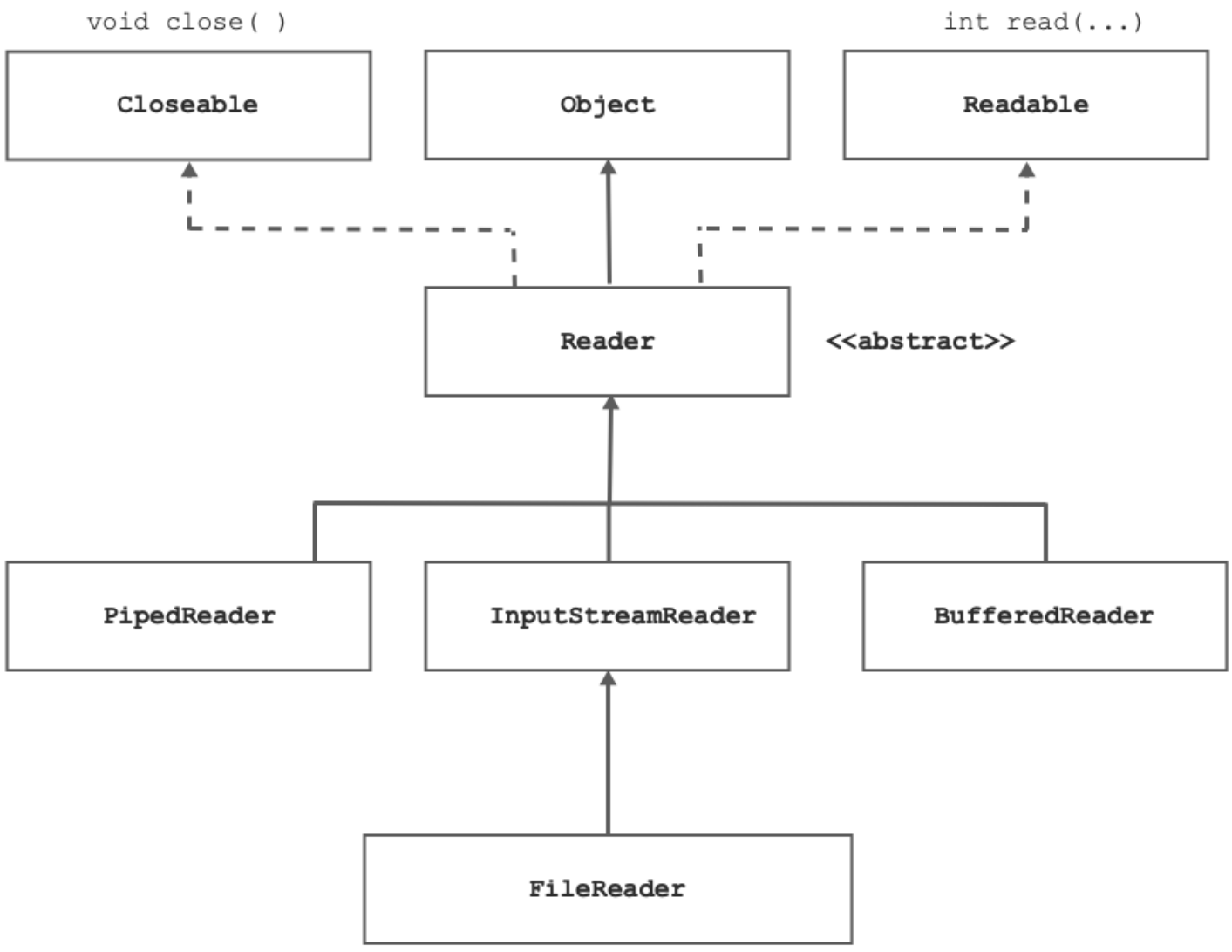


- If we want to read/write data into text file then we should use Reader, Writer and their sub classes.

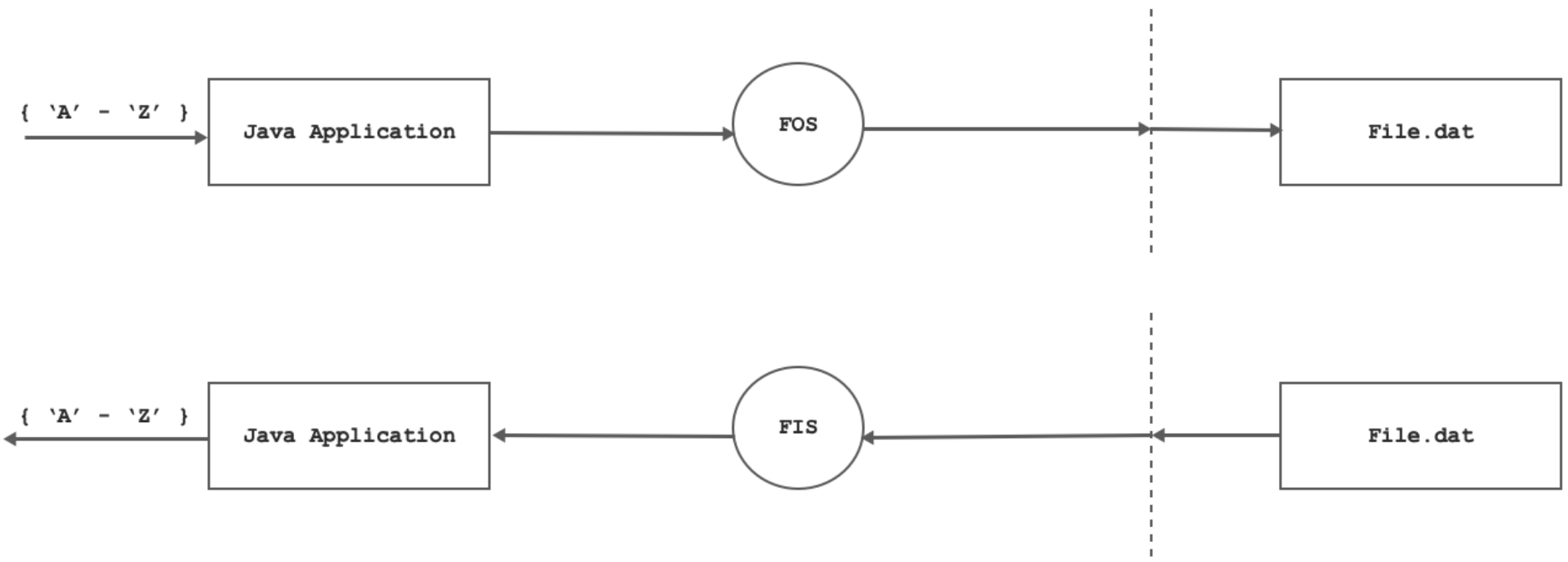
- Consider Writer hierarchy:

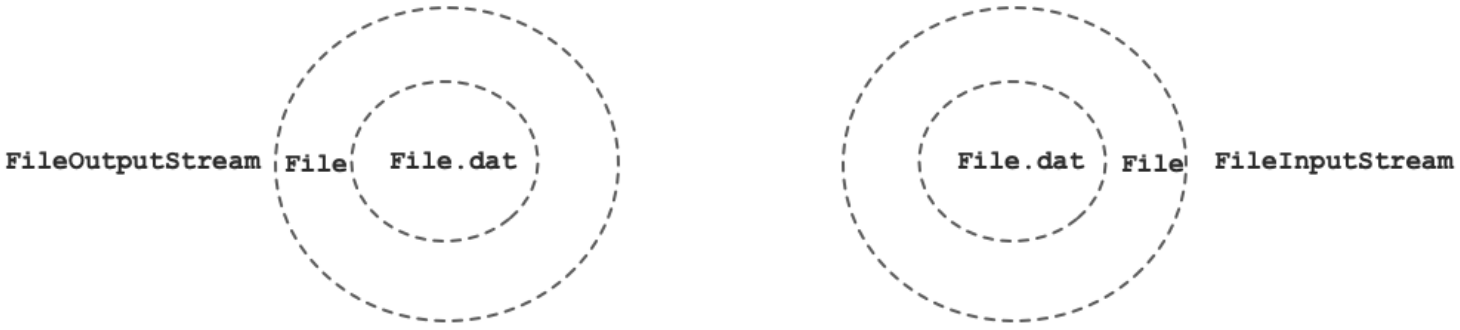


- Consider Reader hierarchy:

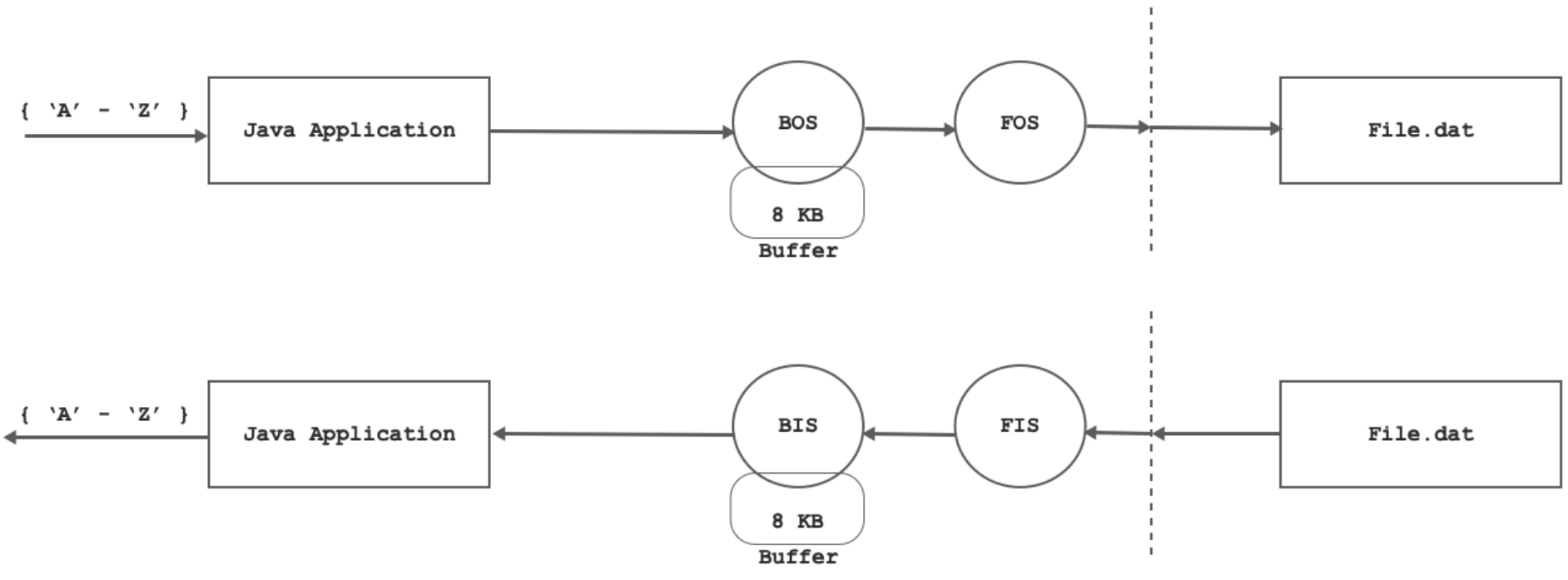


- How to read and write single byte data into file?

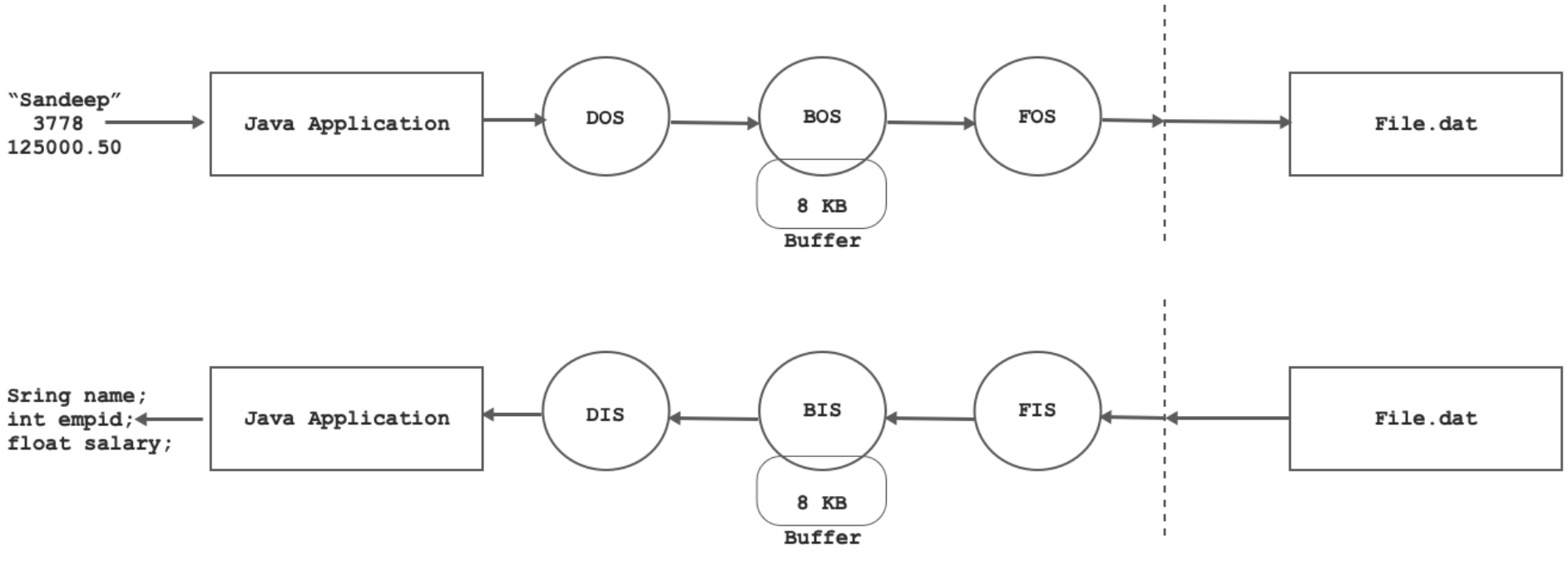




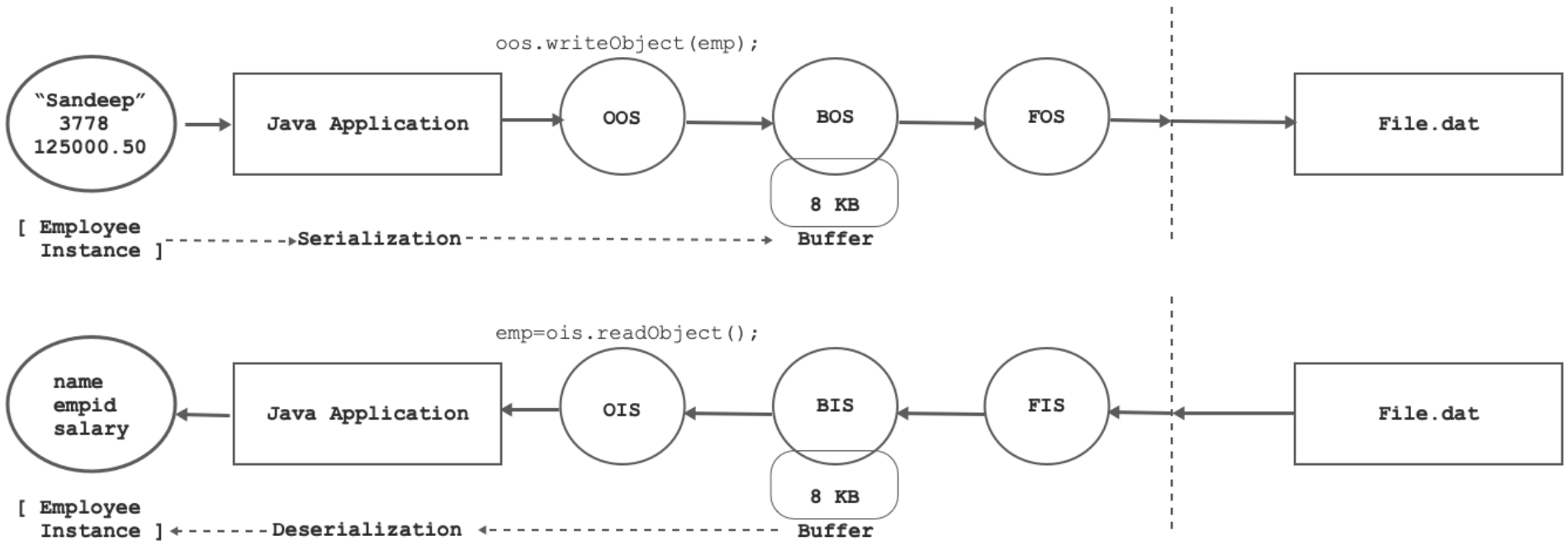
- How to improve performance of read/write operations?



- How to read/write primitive values and Strings to and from files?



- How to serialize / deserialize state of java Instance?



- Process of converting state of Java instance into binary data is called as serialization.
- To serialize Java instance type of instance must implement Serializable interface otherwise we get NotSerializableException.
- If class contains fields of non primitive type then its type must implement Serializable interface. Consider following example:

```
class Date implements Serializable{
}
class Address implements Serializable{
}
class Person implements Serializable{
  private String name;  //Final class. Already implemented Serializable
  private Address address;
  private Date birthDate;
}
```

- transient is modifier in Java. If we declare any field transient then JVM do not serialize its state.
- Process of converting binary data into Java instance is called as deserialization.

SerialVersionUID

- In Java, the **serialVersionUID** is a special static variable that is used to control the serialization and deserialization process of objects. It is a version number associated with a serialized class, and it serves as a unique identifier for the class.
- When an object is serialized, its state is converted into a byte stream. The **serialVersionUID** is included in this byte stream. During deserialization, the JVM checks if the **serialVersionUID** of the serialized object matches the **serialVersionUID** of the corresponding class in the local environment. If the **serialVersionUID** values match, the deserialization process proceeds successfully. However, if the **serialVersionUID** values don't match, a **InvalidClassException** is thrown, indicating a version mismatch between the serialized object and the class definition.

- The **serialVersionUID** is used for versioning purposes to ensure that the serialized object and the class definition are compatible. It helps to maintain compatibility between different versions of a class when objects are serialized and deserialized.
- Here's an example that demonstrates the usage of **serialVersionUID**:

```java
import java.io.*;
class Sample implements Serializable {
    private static final long serialVersionUID = 1L;
    private int data;

    public Sample(int data) {
        this.data = data;
    }

    public int getData() {
        return this.data;
    }
}
```

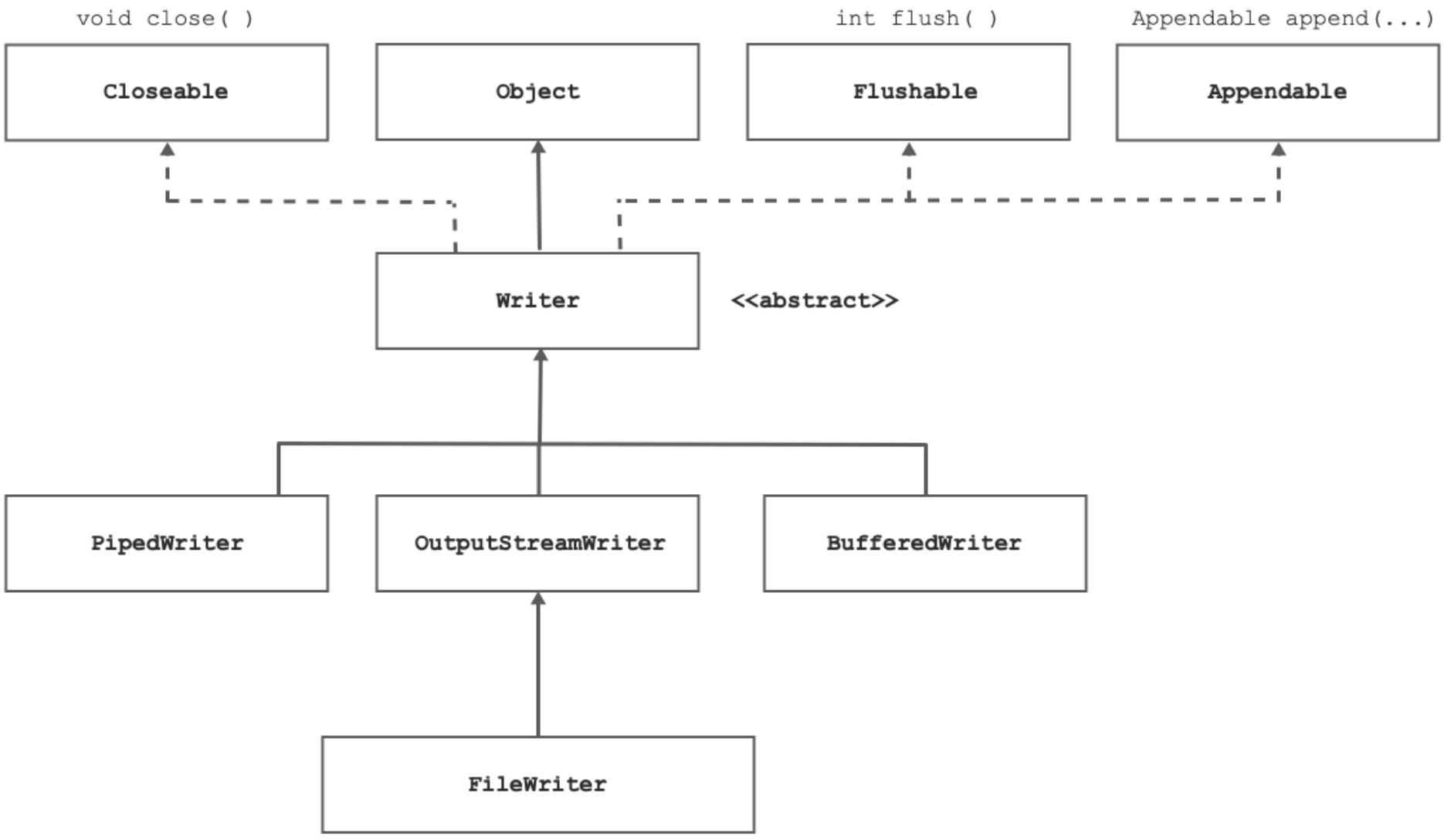
```java
public class Program {
    public static void main(String[] args) {
        Sample sample = new Sample(42);

        // Serialize the object to a file
        try ( ObjectOutputStream outputStream = new ObjectOutputStream(new FileOutputStream("object.ser"))) {
            outputStream.writeObject(sample);
            System.out.println("Object serialized successfully.");
        } catch (IOException e) {
            e.printStackTrace();
        }

        // Deserialize the object from the file
        try (
            ObjectInputStream inputStream = new ObjectInputStream(new FileInputStream("object.ser"))){
            Sample sample = (Sample) inputStream.readObject();
            System.out.println("Deserialized object data: " + sample.getData());
        } catch (IOException | ClassNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```

- In the above example, the Sample implements the Serializable interface, indicating that its objects can be serialized. The class also defines a seri**serialVersionUID**alVersionUID as 1L. When an object of Sample is serialized, the **serialVersionUID** is included in the serialized byte stream. During deserialization, the **serialVersionUID** is checked to ensure compatibility between the serialized object and the class definition.
- It's important to note that if you make any changes to a serialized class, such as adding or removing fields or changing their types, you should update the **serialVersionUID** accordingly to maintain compatibility between different versions of the class.

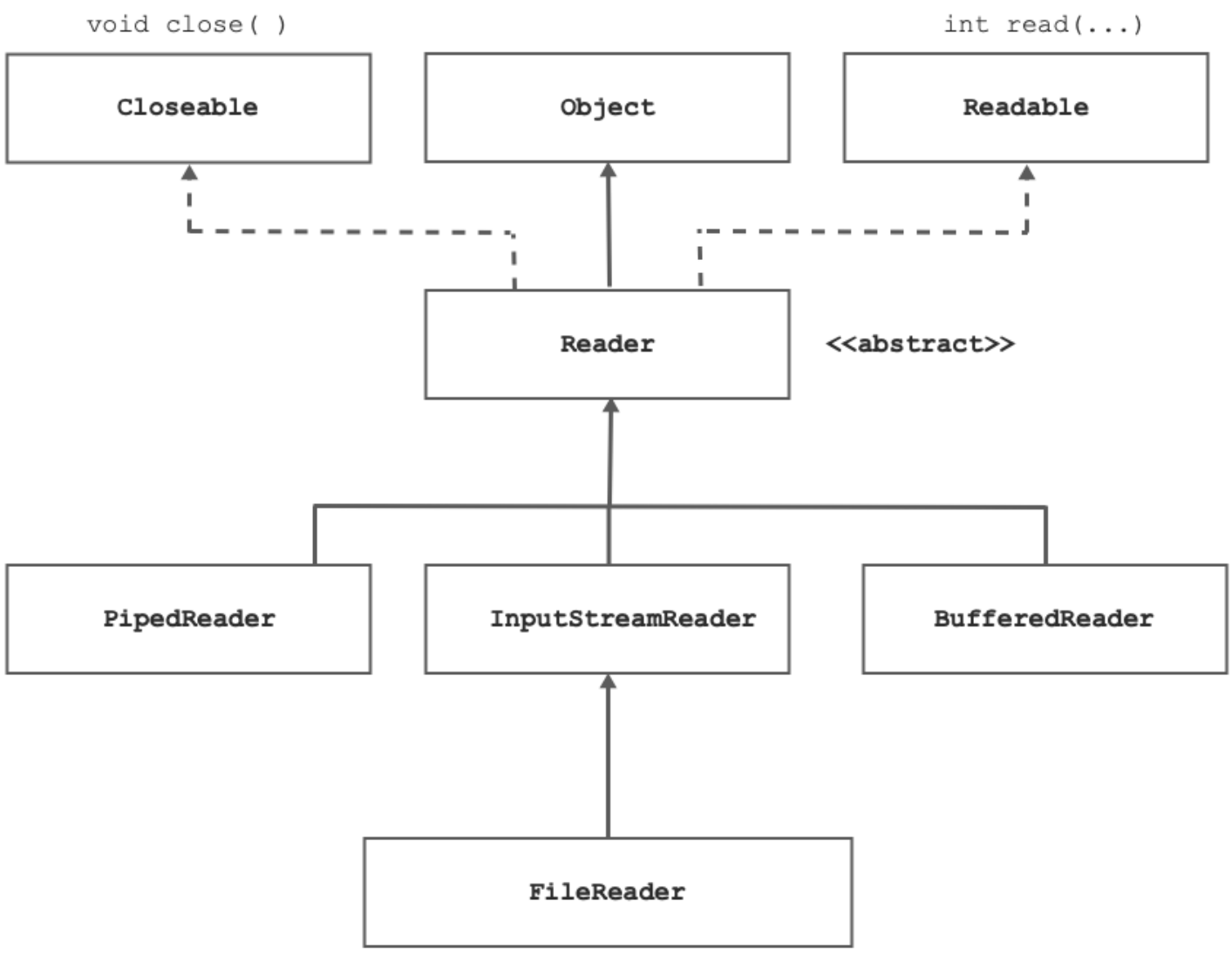
Text file manipulation

- If we/you want to manipulate text file then we should use Reader/writer classes and their sub classes.

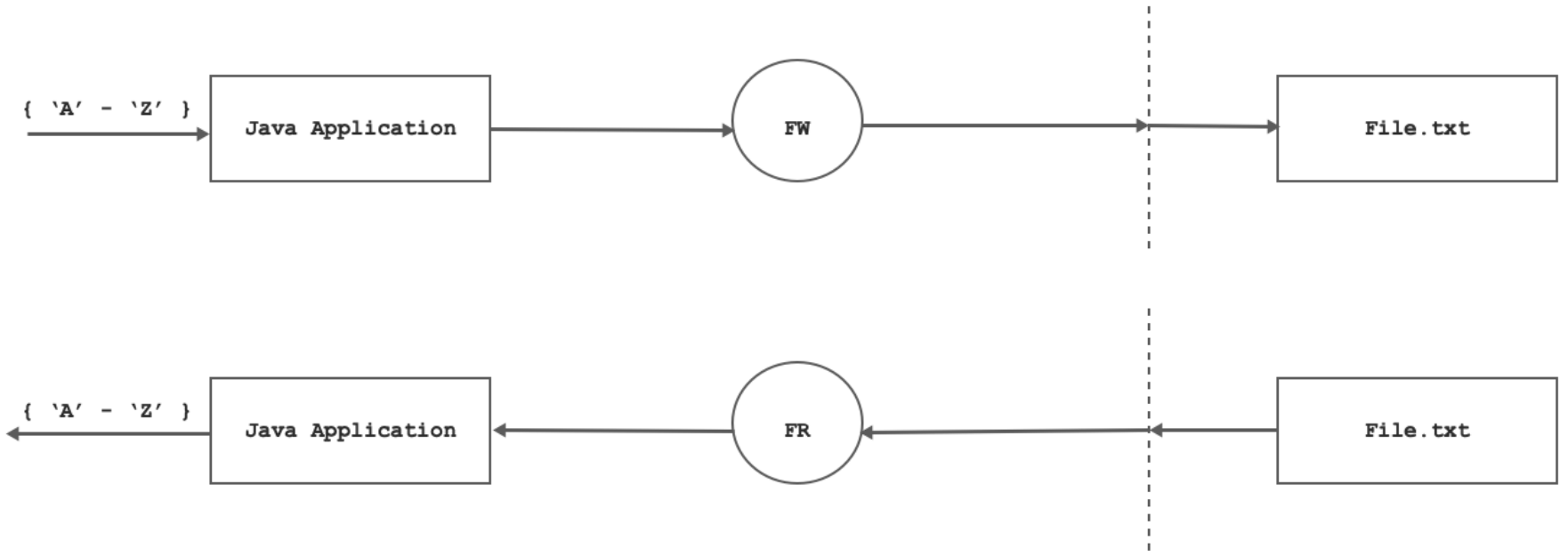
- Consider Writer hierarchy:



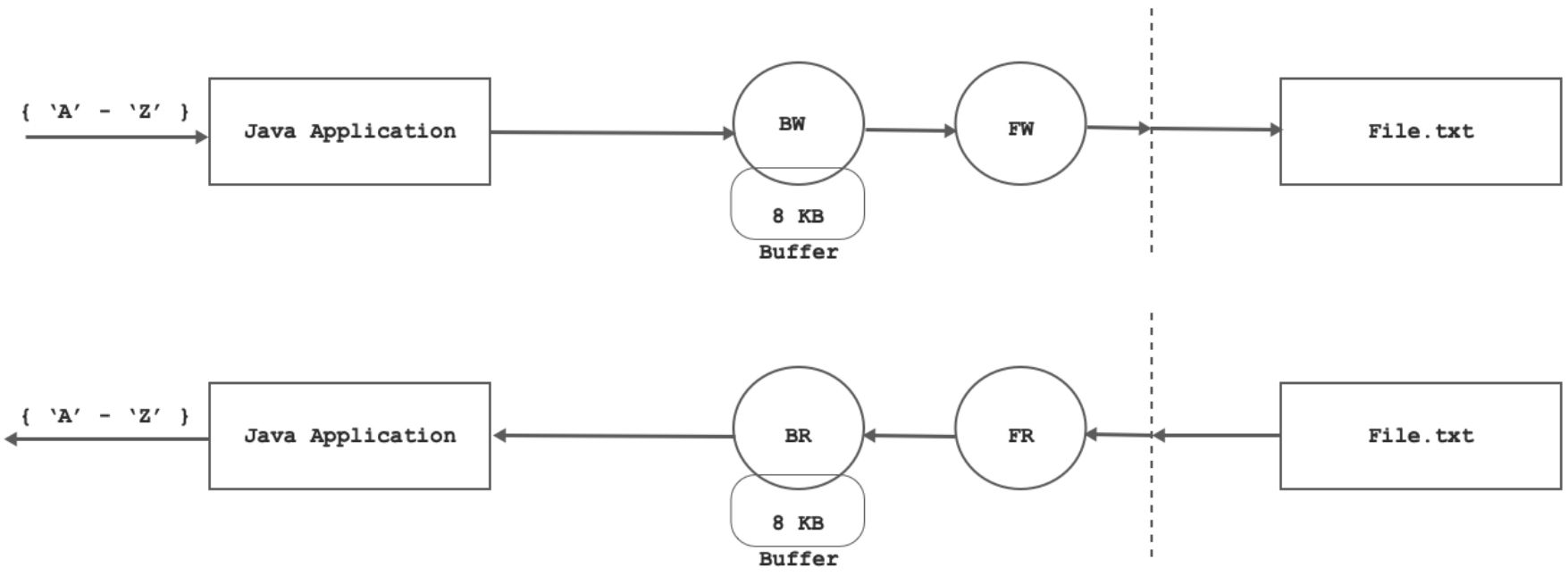
- Consider Reader hierarchy:



- The **FileWriter** class is used to write characters to a file. It extends the Writer class and provides methods to write characters or character arrays to a file. It handles the underlying low-level operations required for writing characters to a file, such as opening the file, writing the data, and closing the file.
- The **FileReader** class is used to read characters from a file. It extends the Reader class and provides methods to read characters into a buffer from a file. It handles the low-level operations required for reading characters from a file, such as opening the file, reading the data, and closing the file.



- The **BufferedWriter** class is used to write characters to a character stream with buffering capabilities. It wraps an existing Writer and improves the performance of writing characters by buffering them in memory before writing them to the underlying stream.
- The **BufferedReader** class is used to read characters from a character stream with buffering capabilities. It wraps an existing Reader and improves the performance of reading characters by buffering them in memory before accessing the underlying stream.



- In Java, InputStreamReader and OutputStreamWriter are classes that provide a bridge between byte streams and character streams. They are used to convert bytes to characters (for input) and characters to bytes (for output) while reading from or writing to streams.
- The InputStreamReader class is used to read bytes from an InputStream and decode them into characters using a specified character encoding. It converts a stream of bytes into a stream of characters.
- Consider following code:

```java
import java.io.*;
public class Program {
    public static void main(String[] args) {
        String inputFile = "input.txt";
        // Reading from a file using InputStreamReader
        try (FileInputStream fileInputStream = new FileInputStream(inputFile);
            InputStreamReader inputStreamReader = new InputStreamReader(fileInputStream)) {
            char[] buffer = new char[1024];
            int length;
            while ((length = inputStreamReader.read(buffer)) != -1) {
                System.out.println(new String(buffer, 0, length));
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

- The OutputStreamWriter class is used to write characters to an output stream of bytes. It wraps an existing OutputStream and provides methods to write characters to the output stream using a specified character encoding.
- Consider following example:

```
import java.io.*;
public class Program {
    public static void main(String[] args) {
        String outputFile = "output.txt";
        // Writing to a file using OutputStreamWriter
        try (FileOutputStream fileOutputStream = new FileOutputStream(outputFile);
            OutputStreamWriter outputStreamWriter = new OutputStreamWriter(fileOutputStream)) {
            String data = "Hello, World!";
            outputStreamWriter.write(data);
            System.out.println("Data written to the file.");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

- How to serialize and deserialize state of Java instance into text file?

```
class Employee implements Serializable {
  private static final long serialVersionUID = 1L;

  private String name;
  private int age;
  private String department;

  public Employee(String name, int age, String department) {
      this.name = name;
      this.age = age;
      this.department = department;
  }
  public String getName() {
    return this.name;
  }
  public int getAge() {
    return this.age;
  }
  public String getDepartment() {
    return this.department;
  }
  public String toString() {
        return "Name: " + this.name + ", Age: " + this.age + ", Department: " + this.department;
    }
}
```

```
public class Program {
  public static void writeRecord( String pathname ) throws Exception{
    try ( BufferedWriter bufferedWriter = new BufferedWriter(new FileWriter(pathname))) {
      Employee employee = new Employee("Sandeep Kulange", 39, "Engineering");
        String str = new StringBuilder()
            .append(employee.getName()).append(",")
            .append(employee.getAge()).append(",")
            .append(employee.getDepartment())
            .toString();
        bufferedWriter.write(str);
        System.out.println("Employee object serialized and written to the "+pathname+" file.");
    }
  }
  private static Employee deserializeEmployee(String str) {
    String[] parts = str.split(",");
    String name = parts[0].trim();
    int age = Integer.parseInt(parts[1].trim());
    String department = parts[2].trim();
    return new Employee(name, age, department);
  }
  public static void readRecord( String pathname ) throws Exception{
    try (BufferedReader bufferedReader = new BufferedReader(new FileReader(pathname))) {
      String str = bufferedReader.readLine();
```

```
            Employee emp = deserializeEmployee(str);
            System.out.println(emp.toString());
        }
      }
      public static void main(String[] args) {
        try {
          String fileName = "employees.txt";
          Program.writeRecord(fileName);
          Program.readRecord(fileName);
        } catch (Exception e) {
          e.printStackTrace();
        }
      }
    }
```