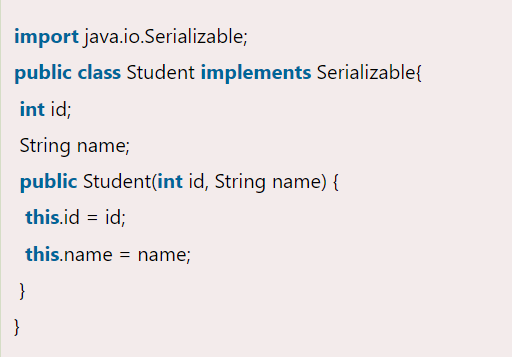
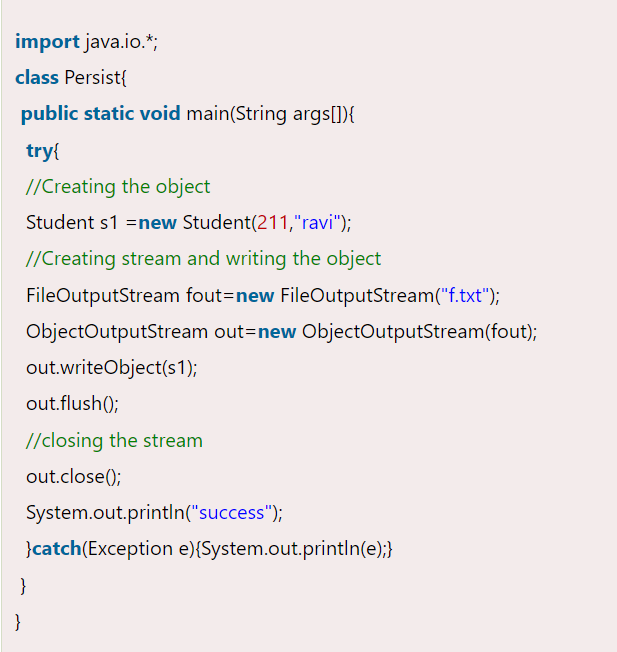
1. Threading
2. JRE JDK JVM
3. JDBC
4. Collections
5. OOPS
6. Singleton
7. Java 8 features
8. Comparator vs comparable
9. Files
10. Synchronization vs Asynchronization
11. StringBuilder vs StringBuffer
12. Equals and == (Heap memory, String pool)
13. Date Time
14. Java Regular Expression
15. Exception Handling
16. Enum
17. Inner class
18. Typecasting
19. Access modifiers
20. Java IO
21. Primitive vs Non primitive
22. Wrapper class Auto Boxing vs Un Boxing
23. Design Pattern
24. Serialization

Serialization is the process of converting an object into a stream of bytes, which can then be stored in a file, sent over a network, or persisted in a database. The reverse process, i.e., reconstructing the object from the serialized bytes, is called deserialization. Serialization is primarily used for data persistence and communication between different Java applications or systems.

In Java, serialization is achieved by implementing the **Serializable** interface. Here's an example:

Student.java





Deserialization is the process of reconstructing the object from the serialized state. It is the reverse operation of serialization. Let's see an example where we are reading the data from a deserialized object.

Deserialization is the process of reconstructing the object from the serialized state. It is the reverse operation of serialization. Let's see an example where we are reading the data from a deserialized object.

In Java, the **flush()** method is used in the context of streams, such as output streams or buffered streams. Here's a simple explanation:

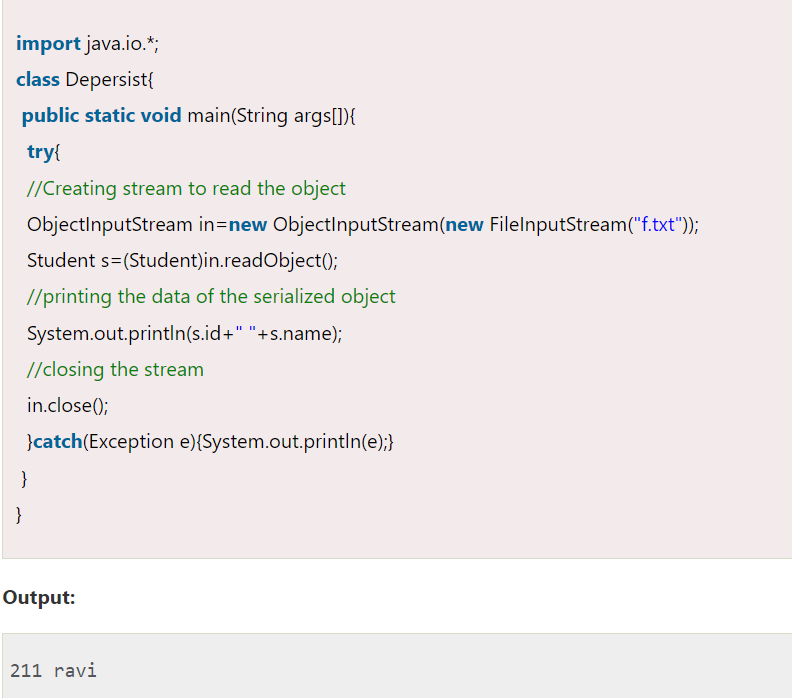
When you write data to a stream (like writing to a file or sending data over a network), Java doesn't immediately send or write the data out. Instead, it stores the data temporarily in a buffer. This buffering helps improve performance because writing data to a file or network can be slow compared to writing to memory.

Now, the **flush()** method comes into play. When you call **flush()** on a stream, you're telling Java to empty that buffer and send out any data that's been stored there immediately. It's like saying, "Okay, I'm done for now. Send out whatever is in the buffer."

Why would you want to do this? Well, sometimes you might want to make sure that data is sent or written right away, instead of waiting for the buffer to fill up naturally. For example, if you're sending data over a network connection and you want to ensure that the other end receives the data promptly, you might call **flush()** to force it to be sent immediately.

In summary, **flush()** is a method used with streams in Java to force any buffered data to be sent or written immediately, instead of waiting for the buffer to fill up naturally. It's handy when you want to ensure data is sent or written promptly.

Top of Form



1. Why Strings are immutable

Strings are immutable in many programming languages, including Java, Python, and C#. There are several reasons why strings are designed to be immutable:

1. \*Thread Safety\*: Immutability ensures that strings cannot be modified after creation. This property makes strings inherently thread-safe because multiple threads can safely read the same string without worrying about concurrent modifications.

2. \*Caching\*: Since strings are immutable, they can be cached and reused. This optimization can improve performance and reduce memory usage, especially in scenarios where the same string is used multiple times.

3. \*Security\*: Immutable strings prevent unintended modifications, which can be crucial in security-sensitive applications. For example, if strings representing sensitive data (such as passwords or cryptographic keys) were mutable, they could be inadvertently modified, compromising security.

4. \*Hashing\*: Immutable strings allow for efficient hashing and comparison operations. Once a string is hashed, its hash value remains constant, simplifying hash table implementations and string comparison algorithms.

5. \*Predictability\*: Immutability leads to predictable behavior. Since strings cannot change, developers can rely on their values remaining constant throughout the program's execution, which reduces complexity and potential bugs.

6. \*Optimizations\*: Immutable strings enable compiler and runtime optimizations. For example, string interning techniques can be used to optimize memory usage by reusing identical string literals.

Overall, immutability provides several benefits in terms of performance, thread safety, security, and predictability, making it a common design choice for strings in many programming languages.