JAVA Core 4

4. Variables and identifiers
    1. Variables having primitive data type
    2. Convention for making variables and methods in java

4. Variables and Identifiers

An identifier is a name given to a program element, while a variable is a specific kind of program element, a storage location identified by an identifier, it is used to store data during the program's execution.

Identifiers are not limited to variables, they can also be used for classes, methods, etc.

Variables are a type of identifier used specifically for storing and managing data in a program.

4.1 Variables having primitive data type

As described above variables are the memory location where a certain type of data is stored during the programs execution. The Primitive data type variable indicates that the data that is to be stored is primitive in nature and can be store using literals and raw values.

That is why this type of data can be read from as it is from any readable sources, that source can be

- **Command Line**
- **Text Files**
- **Databases**
- **Data Streams**

Java makes the I/O operations using streams (which is defined as a sequence of elements), for reading or writing data from any of the above sources java uses 2 types of Streams, Byte Stream and Character Stream.

The most commonly used Byte Stream in java are :

Byte Stream Class	Description
`FileInputStream`	Reads bytes from a file. Commonly used for file input operations.
`FileOutputStream`	Writes bytes to a file. Commonly used for file output operations.
`ByteArrayInputStream`	Reads bytes from a byte array. Useful for reading in-memory byte data.
`ByteArrayOutputStream`	Writes bytes to a byte array. Useful for capturing output in-memory.
`BufferedInputStream`	Adds buffering to an input stream, improving performance by reducing I/O operations.
`BufferedOutputStream`	Adds buffering to an output stream, improving performance by reducing I/O operations.
`DataInputStream`	Reads primitive Java data types from an input stream.
`DataOutputStream`	Writes primitive Java data types to an output stream.
`ObjectInputStream`	Reads objects from an input stream. Commonly used for serialization and deserialization.
`ObjectOutputStream`	Writes objects to an output stream. Commonly used for serialization and deserialization.
`PrintStream`	Provides print methods for writing formatted text. Commonly used for console output.
`RandomAccessFile`	Supports random access to a file's contents. Allows reading and writing at any position in a file.

The most commonly used Character Stream in java are :

Character Stream Class	Description
`FileReader`	Reads characters from a file. Commonly used for file input operations.
`FileWriter`	Writes characters to a file. Commonly used for file output operations.
`CharArrayReader`	Reads characters from a character array. Useful for reading in-memory character data.
`CharArrayWriter`	Writes characters to a character array. Useful for capturing output in-memory.
`BufferedReader`	Adds buffering to a character input stream, improving performance by reducing I/O operations. Often used with `FileReader` or other `Reader` classes.
`BufferedWriter`	Adds buffering to a character output stream, improving performance by reducing I/O operations. Often used with `FileWriter` or other `Writer` classes.
`StringReader`	Reads characters from a string.
`StringWriter`	Writes characters to a string.
`PrintWriter`	Provides print methods for writing formatted text to a file or another output stream. Offers a higher-level API than `FileWriter`.
`Scanner`	Parses primitive types and strings using regular expressions. It can read from various sources, including files and strings.

While working with streams consider the following :

1. **Select the Appropriate Stream Class:**
   - **Byte Streams (`InputStream` / `OutputStream`):**
     - For reading binary data from files: `FileInputStream`.
     - For reading from byte arrays: `ByteArrayInputStream`.
     - For writing binary data to files: `FileOutputStream`.
     - For writing to byte arrays: `ByteArrayOutputStream`.
   - **Character Streams (`Reader` / `Writer`):**
     - For reading characters from files: `FileReader`.
     - For reading characters from strings: `StringReader`.
     - For writing characters to files: `FileWriter`.
     - For writing characters to strings: `StringWriter`.
2. **Use Try-With-Resources:**
   - Always use try-with-resources to ensure that the streams are properly closed, even if an exception occurs. This is essential for preventing resource leaks.
3. **Buffering for Efficiency:**

- Consider using buffered streams (`BufferedInputStream` / `BufferedOutputStream` or `BufferedReader` / `BufferedWriter`) for improved performance. Buffered streams reduce the number of I/O operations by reading or writing data in chunks.

4. **Handle Exceptions:**
    - Handle exceptions appropriately. This may involve catching specific exceptions or propagating them up the call stack. Logging or reporting errors is also crucial for debugging.

In Java there is a class `Scanner` which is easy to use for reading data from various sources :

1. **Reading from console**

```
import java.util.Scanner;

public class ConsoleInputExample {
    public static void main(String[] args) {
    //  passing the Stream from where we want to read.
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter something: ");
        String userInput = scanner.nextLine();
        System.out.println("You entered: " + userInput);
    }
}
```

2. **Reading from files**

```
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

public class FileInputExample {
    public static void main(String[] args) {
        try {
        // creating a stream Object
            File file = new File("path/to/your/file.txt");
        // Passing a file object that will act as a stream
            Scanner scanner = new Scanner(file);

            while (scanner.hasNextLine()) {
                String line = scanner.nextLine();
                System.out.println(line);
            }
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```

JAVA allows to store objects of any class directly with the memory for the fields and methods, this is done by java Serialization API which is an advance topic itself.

4.2 Convention for making variables and other in java

In Java, there are widely accepted conventions for naming variables, methods, and classes. Following these conventions enhances code readability and helps developers understand the structure of your code. Here are some common conventions:

Variables:

1. **Camel Case:** Start with a lowercase letter and capitalize the first letter of each subsequent concatenated word. Example: `myVariable`, `totalAmount`, `employeeName`.
2. **Descriptive Names:** Choose meaningful and descriptive names that convey the purpose of the variable. Avoid single-letter names unless the variable is a loop control variable.

Methods:

1. **Camel Case:** Follow the same camel case convention as variables. Example: `calculateTotal`, `getUserInfo`, `processData`.
2. **Verb-Noun Pairing:** Choose method names that reflect actions or behaviors. Use verb-noun pairs to describe what the method does. Example: `calculateTotal`, `printReport`, `getUserInfo`.

Classes:

1. **Pascal Case:** Class names should follow Pascal case with the first letter capitalized. Example: `MyClass`, `EmployeeDetails`, `BankAccount`.
2. **Noun:** Class names should be nouns and represent an entity or a concept. Example: `Person`, `Car`, `Customer`.
3. **Avoid Underscores:** While underscores are allowed in class names, it's a common convention to avoid them and use camel case instead.

Constants:

1. **Uppercase:** Constants should be written in uppercase letters with underscores separating words. Example: `MAX_SIZE`, `PI_VALUE`, `DEFAULT_TIMEOUT`.
2. **Declare Constants as `final`:** If possible, declare constants using the `final` keyword to indicate that their values cannot be changed.

Packages:

1. **Lowercase:** Package names should be in lowercase. Example: `com.example.project`.
2. **Use Reverse Domain Name:** Consider using the reverse domain name as a prefix for your package to ensure uniqueness. Example: `com.example.project`.