Assignment 1

Ritesh kumar

ritesh282017@gmail.com

1.

Primitive data types are predefined in Java and set of basic data types from which all other data types are constructed.

In Java, there are eight foundational data types known as primitives.

1. byte: This data type is an 8-bit integer that's signed using two's complement notation. Its value range is from -128 to 127, making it a compact choice for storing large arrays of numerical data where the size of each element is a concern.

2. short: Represented by a 16-bit signed integer also using two's complement notation, short's value range extends from -32,768 to 32,767. It's useful for saving memory in large arrays where the magnitude of the numbers isn't too great.

3. int: The int data type is a 32-bit signed integer, following two's complement notation, with a value range from -2^31 to 2^31-1. This is the standard choice for numerical values in Java, unless there's a specific need for a different size or type.

4. long : For larger integer values, the long data type provides a 64-bit signed integer, again using two's complement notation. Its extensive range, from -2^63 to 2^63-1, accommodates the need for large numerical values.

5. float: This 32-bit floating point follows IEEE 754 standards for floating-point arithmetic. It's chosen over double when memory efficiency is more crucial than precision, such as in large arrays of floating-point numbers.

6. double: With double-precision, this 64-bit floating point also adheres to IEEE 754 standards and offers a balance of precision and range, making it suitable for accurate calculations.

7. boolean: The boolean data type is simple, representing truth values with only two possible states: “true” or “false”. It's commonly used for flags and conditional statements.

8. char : This type stores a single 16-bit Unicode character, ranging from '\u0000' to '\uffff', or 0 to 65,535. It's used for representing individual characters in a text.

These primitive types are the building blocks for data manipulation in Java, providing a range of options for numerical, logical, and textual data. Unlike objects, these primitives do not belong to the object hierarchy and are defined directly by the language for efficiency and performance.

2.

A name in java program is called "identifier"
The name can be a classname ,
interfacename,enumname,varaiblename,methodname
and label name.

Rules for Identifiers

Rule1: The allowed characters in java identifiers are
a to z, A to Z,0 to 9, _ ,$

Rule2: If we use any other character, then the program would result in "CompileTime Error".

Rule3: Identifiers should not start with digits

eg: Student123---> valid
123Student---> invalid
Rule4: java identifiers are case sensitive,as such Java language only is case sensitive(JVM).

Rule5: There is no constraint of the length of the java identifiers,but it is not recomended to take the length more than 15.
Eg. int studentplayingcricketwithnationalteam = 100;

Rule6: Reserve words or builtin words can't be used as "java identifiers".
if we try to use then it would result in "CompiletimeError".
.

3.

- A final class in Java is a class that can't be extended or inherited by another class.

- It's like a locked box that can't be opened or changed.

- Declaring a class as final ensures its behavior cannot be altered through inheritance.

- Useful for maintaining the original functionality of the class without modification.

4.

Using the final Keyword: Declare the class as final by using the **final** keyword in the class definition. This is the most straightforward method.

For example:

```
public final class MyFinalClass {
    // class body
}
```

Private Constructors and Factory Methods: Another way to effectively make a class final, without using the **final** keyword, involves making all of the class's constructors private and providing a static method to create instances. This approach prevents inheritance since other classes cannot

call the superclass constructor, which is a requirement for extending a class.

```
public class MyClass {
    // private constructor
    private MyClass() {
        // constructor body
    }

    // static factory method
    public static MyClass createInstance() {
        return new MyClass();
    }
}
```

5. Yes, we can create an instance of a final class in another class in Java. A final class is simply a class that cannot be extended, meaning no other class can inherit from it

1. Final Class Definition: A final class in Java is a class that cannot be extended. No other class can inherit from a final class.

2. Purpose of Final Class: The final modifier is used to ensure the immutability of the class, keeping its implementation secure and unchanged.

3. Instantiation Allowed: Despite the restriction on inheritance, creating instances (objects) of a final class within another class is perfectly allowed.

4. Instantiation Process: The process of creating an instance of a final class is identical to that of any other class. It does not require any special syntax or method.

5. Usage Example: To instantiate a final class named `FinalClass` in another class, you would use the standard instantiation syntax: `FinalClass instance = new FinalClass();`.

6. Practical Implication: This feature allows developers to maintain the integrity of their class's behavior while still utilizing its functionalities across different parts of their application.

7. Security and Consistency: The ability to instantiate final classes while preventing inheritance ensures that the core logic and behavior of these classes remain secure and consistent across different uses.

6.

The `volatile` keyword in Java is used to indicate that a variable's value may be changed by multiple threads that are executing concurrently. When a field is declared as volatile, it ensures that any read or write operations on that variable are directly performed on the main memory, not just in the thread's local cache. This guarantees visibility of changes to the variable across threads.

1. Visibility Guarantee : Ensures that any thread that reads a volatile variable will see the most recent write to that variable by any thread.

2. No Caching : Prevents the caching of volatile variables by threads, so changes made in one thread are immediately visible to other threads.

3. Use Cases: Often used for variables that may be changed by multiple threads, such as a flag to stop a thread.

4. Limits: While `volatile` guarantees visibility, it does not ensure atomicity. For atomic operations beyond simple read/write, consider using `synchronized` blocks or `java.util.concurrent.atomic` package classes.

5. Performance Consideration: Accessing volatile variables is slower than accessing non-volatile variables because of the lack of caching and the requirement to always read from and write to the main memory.

6. Synchronization Alternative: In some cases, `volatile` can be a lighter alternative to synchronization for ensuring data visibility, with less overhead than `synchronized` blocks for certain simple operations.\

7.

The `transient` keyword in Java is used to indicate that a field should not be serialized. Serialization is the process of converting an object's state into a byte stream so that the object can be easily saved to a storage medium or transmitted over a network. However, not all parts of an object may be suitable or necessary for serialization. This is where the `transient` keyword comes into play.

1. Non-Serialization: Fields marked as `transient` are skipped by the serialization mechanism, meaning they are not included in the serialized form of an object.

2. Sensitive Data: It is particularly useful for fields that contain sensitive information that should not be saved or transmitted, such as passwords or security tokens.

3. Temporary State: The `transient` keyword can also be used for fields that represent a temporary state which does not make sense to persist, such as thread states or database connection handles.

4. Default Values: When an object is deserialized, any `transient` fields are not restored from the serialized data. Instead, they are set to their default values (null for objects, 0 for numeric primitives, and false for boolean).

5. Selective Serialization: Using `transient` allows for more selective and efficient serialization, enabling developers to exclude unnecessary data from the serialization process, reducing the size of the serialized object.

6. Usage: To use the `transient` keyword, simply prefix the field declaration with `transient`. For example: `private transient int counter;`.

8.

Type casting is the process in which the compiler automatically converts one data type in a program to another one.

Implicit Casting (Widening Conversion):

This type of casting is done automatically by the Java compiler.

Occurs when the source type has a smaller range than the destination type.There's no risk of data loss.

Example: Converting an int to a long.

int myInt = 100;

long myLong = myInt; // Implicit casting from int to long

System.out.println(myLong); // Outputs: 100

Explicit Casting (Narrowing Conversion):

This type of casting requires a manual operation by the programmer.

Occurs when the source type has a larger range than the destination type.

There's a risk of data loss because the target type may not be large enough to hold the original value.

Example: Converting a double to an int.

double myDouble = 9.78;

int myInt = (int) myDouble; // Explicit casting from double to int

System.out.println(myInt); // Outputs: 9 (fractional part is lost)

9.

Boxing:

This is when you take a basic data type (e.g., an int) and turn it into an object (e.g., an Integer).

Java does this automatically for you.

For example, if you have a number like 5 and you need it to be an object so you can use it in a list, boxing makes that happen.

int basicNumber = 5;

Integer boxedNumber = basicNumber; // This is boxing

Unboxing:

This does the opposite of boxing. It takes an object (e.g., an Integer) and turns it back into a basic data type (e.g., an int).

Java also does this automatically.

So, if you have an object that's a number and you need to do some math with it, unboxing turns it back into a basic number type.

Integer boxedNumber = Integer.valueOf(5);

int basicNumber = boxedNumber; // This is unboxing

Why Do We Need Boxing and Unboxing?

Working with Collections: Java's collections like lists and maps can only store objects, not basic data types. Boxing lets you store basic types in these collections.

Using Methods: Some methods require objects as inputs. Boxing allows you to use basic types with these methods.

Be Careful: While boxing and unboxing are helpful, they can slow down your program if used a lot because creating and converting objects takes extra time.

10.

Keywords:

Keywords are predefined words in Java that have a special meaning to the compiler.

They are used to define the structure and syntax of the code.

For example, class, public, static, and if are all keywords.

You cannot use keywords as names for variables, methods, or classes because they are reserved by the Java language.

Identifiers:

Identifiers are the names given to elements in a program, such as classes, methods, and variables.

They are created by the programmer and can be almost any combination of letters, numbers, $, and _, but they cannot start with a number and must not be a keyword.

Identifiers are used to identify the various elements uniquely within the code.

For example, myVariable, calculateSalary, and EmployeeClass are identifiers.

Literals:

Literals represent fixed values in the code that are not changed during the execution of the program.

They are directly written into the code.

There are different types of literals, including integer literals (e.g., 10), floating-point literals (e.g., 10.01), character literals (e.g., 'a'), string literals (e.g., "Hello"), and boolean literals (e.g., true or false).

Literals are used to initialize variables or use as constants throughout the code.