DATA TYPES

Primitive Data Types

Java has **8 primitive** data types, which are the most basic data types available in the language. They are not objects and hold pure, simple values of a data type.

Primitive data types in Java are directly allocated memory in the stack. When you declare a variable of a primitive data type, Java reserves a fixed amount of memory for that variable based on its type. This memory is allocated on the stack when a method is invoked and is deallocated when the method exits. **Because of this direct allocation,** accessing and manipulating primitive data types is fast and efficient.

Each primitive type has a predefined size:

1. **byte:**
   - Size: 8-bit (1 byte)
   - Range: -128 to 127
   - Use: Can be used in place of `int` to save memory when handling small values.
2. **short:**
   - Size: 16-bit (2 bytes)
   - Range: -32,768 to 32,767
   - Use: Also used to save memory in large arrays, when the memory savings actually matters.
3. **int:**
   - Size: 32-bit (4 bytes)
   - Range: -2^31 to 2^31-1
   - Use: Generally used as the default data type for integral values unless there is a concern about memory.
4. **long:**
   - Size: 64-bit (8 bytes)
   - Range: -2^63 to 2^63-1
   - Use: Used when a wider range than `int` is needed.
5. **float:**
   - Size: 32-bit (4 bytes)
   - Range: Approximately ±3.40282347E+38F (6-7 significant decimal digits)

- Use: Used to save memory in large arrays of floating-point numbers. Not used for precise values, such as currency.

6. **double:**
   - Size: 64-bit (8 bytes)
   - Range: Approximately ±1.79769313486231570E+308 (15 significant decimal digits)
   - Use: The default choice for decimal values, more precise than `float`.

7. **boolean:**
   - Size: Not precisely defined, but generally 1 bit of information (true or false)
   - Use: Used for simple flags that track true/false conditions.

8. **char:**
   - Size: 16-bit (2 bytes), allowing for Unicode characters
   - Range: 0 to 65,535
   - Use: Used to store any character

ORDER AND RANKING

In Java, primitive data types **don't have a hierarchy** in the sense of an inheritance hierarchy like classes do in object-oriented programming. However, there is a concept known as "type promotion" or "type conversion" that implies a sort of ordering when it comes to how primitive types are automatically converted or promoted to other types in expressions. This isn't a hierarchy in the traditional sense, but it does establish a kind of rank or order among the primitive types based on their size and precision.

➢ **Remember:** ***"Big Strong Cats In Large Fancy Dens"***

- byte
- short
- char
- int
- Long
- Float
- Double

Note: `boolean` is not part of this promotion path because it represents true/false values and does not have a numeric size.

It's also worth noting that explicit casting can be used to convert from a type higher in this order to a lower one, but this can lead to data loss and must be done with care.

Type Promotion in Java:

From smaller to larger: This conversion happens automatically because there is no risk of data loss. For example, a **`float`** can be automatically promoted to a **`double`**. This is because **`double`** is a 64-bit data type, which is larger than the 32-bit **`float`** data type, and can accommodate all **`float`** values without loss of precision.

```
float floatValue = 10.5f;
double doubleValue = floatValue; // Automatic promotion from
float to double
```

From larger to smaller: Converting a larger data type like **`double`** to a smaller data type like **`float`** requires explicit casting. This is because **`double`** has a higher precision and range compared to **`float`**, and converting a **`double`** to **`float`** could potentially lead to loss of precision or overflow.

```
double doubleValue = 10.5;
float floatValue = (float) doubleValue; // Explicit casting from
double to float
```

This concept is crucial to understand when working with different data types in Java, ensuring that data is handled correctly and without unintended consequences.

Conversion Rules:

- Converting from a type **higher in order to a lower** one requires **explicit casting** because it might lead to data loss.
- Converting from **lower to higher is implicit** because **there's no risk of data loss**, which is why a `float` can be implicitly converted to a `double`, but converting a `double` to a `float` requires explicit casting due to the potential loss in precision.

Some Basic Coding Exercises:

Byte

Exercise: Write a Java program that takes a byte value as input and prints its square.

```
byte a = 2;
System.out.println("Square value is: " + a * a);
```

Short

Exercise: Create a Java program that initializes a short variable with any value, then converts this short value to a byte and prints both values.

```
short c = 125;
byte bytevalue = (byte) c;
System.out.println("Converted byte value is: " + bytevalue);
```

Int

Exercise: Write a Java program to calculate and print the result of integer division and the remainder of two int variables.

```
int dividend = 25;
int divisor = 4;
int quotient = dividend / divisor;
int remainder = dividend % divisor;

System.out.println("Given dividend: " + dividend + " and divisor: " +
divisor);
System.out.println("Quotient (Result of integer division): " + quotient);
System.out.println("Remainder: " + remainder);
```

Long

Exercise: Write a Java program that initializes a long variable with a very large number (beyond the int range), then multiplies it by another long variable and prints the result.

```
short c = 125;
byte bytevalue = (byte) c;
System.out.println("Converted byte value is: " + bytevalue);
```

Float

Exercise: Write a Java program to calculate the area of a circle. The radius should be a float value.

```
float r = 2f;
System.out.println("The area of the circle is: " + (Math.PI * r * r));
```

Double

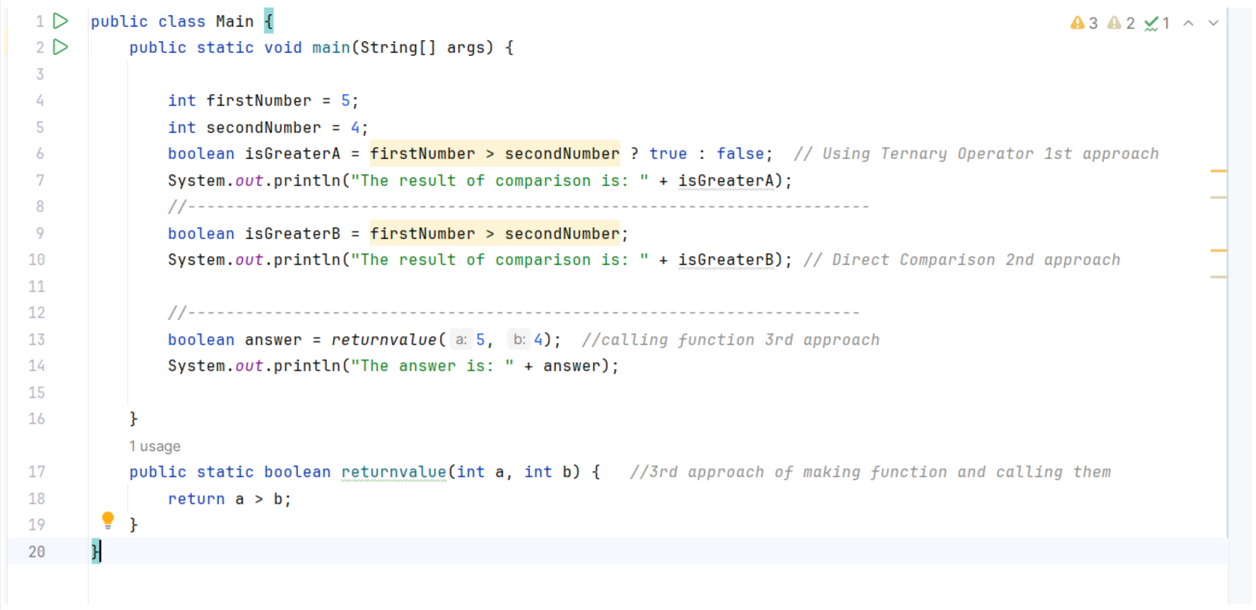
Exercise: Create a Java program that calculates and prints the square root of a double value. Use `Math.sqrt()` method.

```
double m = 9;
System.out.println("The sqrt of m is: " + Math.sqrt(m));
```

Boolean

Exercise: Write a Java program that compares two integers and prints a Boolean value indicating if the first is greater than the second.

This can be done in various ways:



```java
public class Main {
    public static void main(String[] args) {

        int firstNumber = 5;
        int secondNumber = 4;
        boolean isGreaterA = firstNumber > secondNumber ? true : false;  // Using Ternary Operator 1st approach
        System.out.println("The result of comparison is: " + isGreaterA);
        //-----------------------------------------------------------------------
        boolean isGreaterB = firstNumber > secondNumber;
        System.out.println("The result of comparison is: " + isGreaterB); // Direct Comparison 2nd approach

        //----------------------------------------------------------------------
        boolean answer = returnvalue( a: 5,  b: 4);  //calling function 3rd approach
        System.out.println("The answer is: " + answer);

    }
    1 usage
    public static boolean returnvalue(int a, int b) {   //3rd approach of making function and calling them
        return a > b;
    }
}
```

Result: The result of comparison is: true

The result of comparison is: true

The answer is: true

Char

Exercise: Write a Java program that initializes a char variable with an alphabet and prints its corresponding ASCII value.

```java
char i = 'A';
int asciiValue = i;
System.out.println("ASCII value of 'A' is: " + asciiValue);
```

WHAT TO KNOW NEXT !!

Reference Data Types or Object Types

Reference types refer to objects and hence are called reference types. These types do not store the value directly within the variable itself; instead, they store a reference (or address) to the memory location where the actual data or object is stored. This distinction is important because it affects how the data is managed and manipulated within your program. Unlike primitive types, reference types can hold null values or references to objects. The size of a reference type is not fixed, as it depends on the instance of the object it refers to.

For Example:

1. **Classes:**
   - Every class in Java is a reference type, including predefined classes like `String`, `System`, etc., and user-defined classes.
2. **Arrays:**
   - Arrays in Java are a group of like-typed variables that are referred to by a common name. Arrays can be of primitive data types or reference types.
3. **Strings:**
   - The `String` class is not a primitive data type but a very commonly used class in Java. It represents a sequence of characters.
4. **Interfaces:**
   - An interface in Java is a reference type, similar to a class, and is a collection of abstract methods.

Understanding these data types, their sizes, ranges, and uses helps in choosing the right type for your variables, leading to more efficient and effective Java programs. We will talk about these in detail in a later session.

Characteristics of Reference (Non-Primitive) Data Types:

- **Memory Allocation:** Memory for reference types is allocated on the heap, and they can be used to invoke methods since they refer to objects.
- **Default Value:** The default value for reference types is `null`, indicating that they point to no object initially.
- **Size:** The size of a reference variable is not fixed and depends on the architecture of the Java Virtual Machine (JVM), not on the size of the object it refers to.
- **Examples:** Classes, Interfaces, Arrays, Strings, etc.

Key Reference Types:

1. **Classes:** User-defined or predefined (like `String`, `Integer`, etc.), classes are blueprints for creating objects. Each object created from a class can have its own state and behavior as defined by its class.
2. **Arrays:** Arrays in Java are objects that hold multiple variables of the same type. They can hold both primitive and reference types. An array itself is treated as an object.
3. **Strings:** Even though `String` appears to be a primitive type due to its extensive support and syntax within Java, it is **actually a class** and therefore a reference type. Strings are immutable in Java, meaning once created, their values cannot be changed.
4. **Interfaces:** Interfaces in Java define a contract (a set of methods) that implementing classes must adhere to. An interface itself is an abstract data type, and variables of an interface type can reference any object that implements the interface.
5. **Enums:** Enumerations or Enums in Java are special classes that represent a group of constants (unchangeable variables, like final variables).

When working with reference types, it's crucial to understand concepts such as object creation (`new` keyword), memory management, and garbage collection, as these aspects are fundamental to effective and efficient Java programming. Reference types offer the flexibility and power of object-oriented programming, allowing for complex data structures, inheritance, polymorphism, encapsulation, and more. These all will be discussed later in detail.

------------------------**THE END OF CHAPTER 1**--------------------------------------

Note:

The Next chapter will be in **Syntax Basics:** Learning about Java syntax rules, naming conventions for identifiers (variables, methods, classes), and commenting your code.