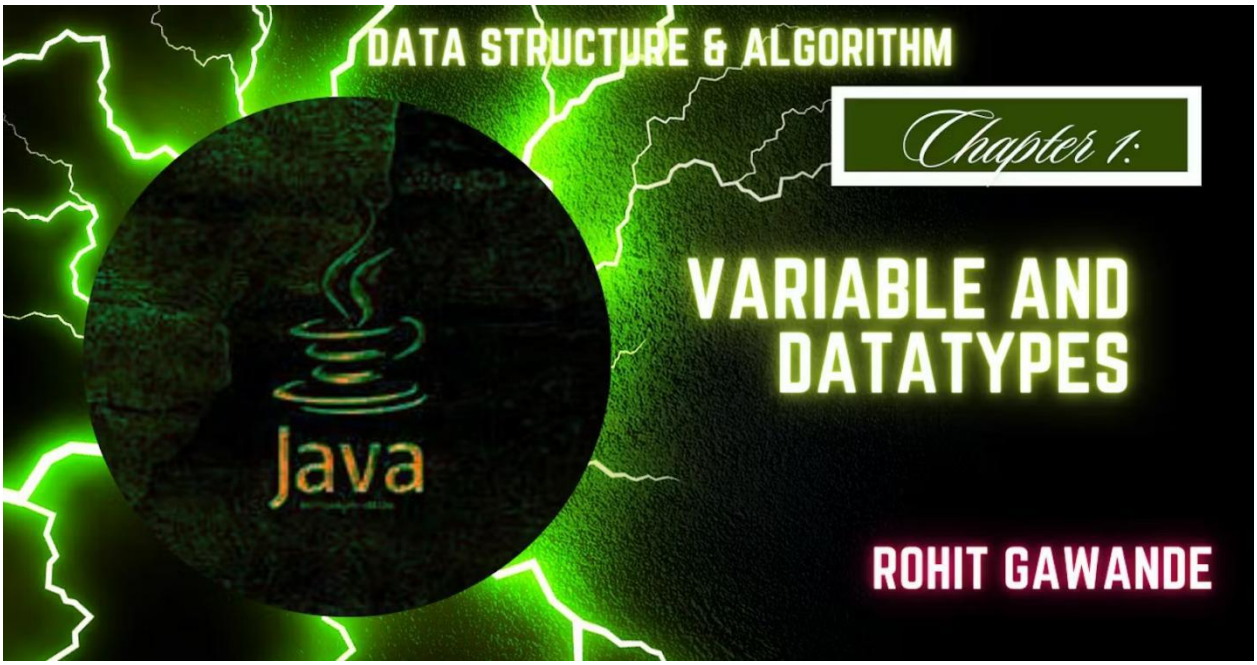
Java With DSA by Apna College: Chapter 1

rohit253.hashnode.dev/01java-basics-unveiled-a-comprehensive-guide-to-variables-data-types-and-basic-operations

Rohit Gawande



Java is a powerful programming language widely used for building various applications. In this chapter, we will explore some fundamental concepts such as creating a Java file, understanding boilerplate code, and working with variables and data types. We'll also cover basic input/output operations, mathematical operations, and assignments.

1. Creating a Java File

To start coding in Java, the first step is to create a Java file. A Java file is simply a text file with the `.java` extension. The file name should match the class name. For example, if your class is named `Hello_World`, the file should be saved as `Hello_World.java`.

2. Boilerplate Code

Java programs require a specific structure or "boilerplate code" to run. This includes defining a class and the main method, which serves as the entry point of the program.

```
public class Hello_World {
    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}
```

- **Class Declaration**: The `public class Hello_World` line declares a class named `Hello_World`.
- **Main Method**: The `public static void main(String[] args)` method is where the program execution begins. The `System.out.println("Hello World!");` statement prints "Hello World!" to the console.

3. Output in Java

The `System.out.println()` function is used to print text to the console. For example:

```
System.out.println("Hello World!");
```

This will output `Hello World!` on the console.

4. Print a Pattern

You can use the `System.out.println()` function to print patterns. Here's an example of printing a triangle pattern:

```
public class PrintTriangle {
    public static void main(String[] args) {
        System.out.println("****");
        System.out.println("***");
        System.out.println("**");
        System.out.println("*");
    }
}
```

This code will output:

```
****
***
**
*
```

5. Variables in Java

Variables in Java

In Java, variables are fundamental constructs used to store data that can be manipulated throughout your program. Understanding how to declare and use variables is crucial for any Java programmer. Here’s a comprehensive overview of variables in Java:

5.1. What is a Variable?

A variable in Java is a named storage location in memory that holds data. Variables can store different types of data, and the type of data a variable can hold is determined by its data type.

5.2. Declaring Variables

Before using a variable, you must declare it by specifying its type and name. The syntax is:

```
type variableName;
```

- `type`: Specifies the data type of the variable (e.g., `int`, `String`, `float`).
- `variableName`: The name you give to the variable.

Example:

```
int age;
String name;
```

5.3. Initializing Variables

After declaring a variable, you can assign a value to it. This is called initialization.

Syntax:

```
variableName = value;
```

Example:

```
age = 20;
name = "Rohit Gawande ";
```

You can also declare and initialize a variable in a single line:

```
int age = 20;
String name = "Rohit Gawande ";
```

5.4. Variable Types

Java has several types of variables based on the data they hold:

Primitive Data Types:

- `byte`: 8-bit integer

- `short`: 16-bit integer
- `int`: 32-bit integer
- `long`: 64-bit integer
- `float`: Single-precision 32-bit floating-point
- `double`: Double-precision 64-bit floating-point
- `char`: 16-bit Unicode character
- `boolean`: True or false value

Non-Primitive (Reference) Data Types:

- `String`: Represents a sequence of characters
- **Arrays**: Store multiple values of the same type
- **Objects**: Instances of user-defined classes

5.5. Example Usage

Here's a simple Java program demonstrating the use of different types of variables:

```
public class VariablesExample {
    public static void main(String[] args) {
        // Primitive Data Types
        int age = 20;
        float height = 5.5f;
        char initial = 'R';
        boolean isStudent = true;

        // Non-Primitive Data Type
        String name = "Rohit Gawande";

        // Output the values
        System.out.println("Name: " + name);
        System.out.println("Age: " + age);
        System.out.println("Height: " + height);
        System.out.println("Initial: " + initial);
        System.out.println("Is Student: " + isStudent);
    }
}
```

5.6. Variable Scope

The scope of a variable defines where it can be accessed within your code:

- **Local Variables**: Declared inside a method or block and accessible only within that method or block.
- **Instance Variables**: Declared inside a class but outside any method. They are accessible to all methods within the class.
- **Class Variables**: Declared with the `static` keyword inside a class. They are shared among all instances of the class.

```
public class Variable {
    public static void main(String[] args) {
        int a = 5;
        int b = 9;
        System.out.println(a); // Output: 5
        String name = "Rohit Gawande";
        System.out.print(name); // Output: Rohit Gawande
    }
}
```

5.7. Best Practices

- Use meaningful variable names to make your code more readable.
- Follow naming conventions: variables should start with a lowercase letter and use camelCase for multiple words (e.g., `totalAmount`).
- Initialize variables before using them to avoid errors.

6. Data Types in Java

Java is a statically typed language, meaning each variable must have a data type.Java has various data types to store different kinds of data:

- **Primitive Data Types**: These include `int` (integer), `float` (floating-point number), `char` (character), `boolean` (true/false), etc.
- **Non-Primitive Data Types**: These include `String`, arrays, classes, etc.

Here's a summary of primitive data types:

Data Type	Description	Example
`int`	Integer	`5`
`float`	Floating-point number	`5.99f`
`char`	Character	`'A'`
`boolean`	True/False	`true`

7. Sum of Two Numbers

Let's write a simple program to calculate the sum of two numbers:

```
public class Sum {
    public static void main(String[] args) {
        int a = 8;
        int b = 7;
        int sum = a + b;
        System.out.println(sum); // Output: 15
    }
}
```

8. Comments in Java

Comments are used to explain code and are ignored by the compiler. There are two types:

- **Single-line comments**: Start with `//`
- **Multi-line comments**: Enclosed in `/* */`

Example:

```
// This is a single-line comment
/* This is a
   multi-line comment */
```

9. Input in Java

Handling input in Java is a common requirement, especially when building interactive programs where users provide data. Java provides several ways to take input from the user, with the most common being through the `Scanner` class, which is part of the `java.util` package. Here's a detailed explanation of how to use input in Java:

1. Using the Scanner Class

The `Scanner` class provides methods to read input of various data types, such as integers, floating-point numbers, and strings. To use the `Scanner` class, you first need to import it:

```
import java.util.Scanner;
```

2. Creating a Scanner Object

Before reading input, you need to create an instance of the `Scanner` class. This object will be used to capture user input:

```
Scanner sc = new Scanner(System.in);
```

> `System.in` is an input stream that reads data from the standard input (usually the keyboard).

3. Reading Different Types of Input

Once you have created a `Scanner` object, you can use it to read various types of input from the user:

i) Reading an Integer:

```
System.out.println("Enter an integer:");
int number = sc.nextInt();
System.out.println("You entered: " + number);
```

> `sc.nextInt()` reads an integer value from the user.

ii) Reading a Float:

```
System.out.println("Enter a float:");
float f = sc.nextFloat();
System.out.println("You entered: " + f);
```

> `sc.nextFloat()` reads a floating-point number.

iii) Reading a String:

```
System.out.println("Enter a string:");
String str = sc.next();
System.out.println("You entered: " + str);
```

> `sc.next()` reads a single word (up to the first space or newline).

iv) Reading a Line of Text:

```
System.out.println("Enter a full line:");
String line = sc.nextLine();
System.out.println("You entered: " + line);
```

`sc.nextLine()` reads the entire line of text, including spaces.

4. Example Program: Sum of Two Numbers

Here's a simple example that reads two numbers from the user and prints their sum:

```
import java.util.Scanner;

public class InputSum {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        // Reading two integers from the user
        System.out.print("Enter first number: ");
        int num1 = sc.nextInt();

        System.out.print("Enter second number: ");
        int num2 = sc.nextInt();

        // Calculating the sum
        int sum = num1 + num2;

        // Printing the result
        System.out.println("The sum is: " + sum);
    }
}
```

Conclusion

The `Scanner` class in Java provides a simple and effective way to handle user input. Whether you're reading integers, floating-point numbers, strings, or characters, understanding how to use `Scanner` will allow you to create interactive and user-friendly programs. Remember to always validate input to avoid exceptions and ensure a smooth user experience!

10. Product of Two Numbers

Similarly, you can calculate the product of two numbers using input from the user:

```
import java.util.Scanner;

public class Prod {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter a:");
        int a = sc.nextInt();
        System.out.println("Enter b:");
        int b = sc.nextInt();
        int mul = a * b;
        System.out.println("The product of a and b: " + mul);
    }
}
```

11. Area of a Circle

To calculate the area of a circle, you can use the formula `πr^2` where `r` is the radius:

```
import java.util.Scanner;

public class CircleArea {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter radius:");
        float radius = sc.nextFloat();
        float area = 3.14f * radius * radius;
        System.out.println("Area of Circle: " + area);
    }
}
```

12. Type Conversion in Java

Type conversion in Java refers to converting a variable from one data type to another. This process can be either automatic (implicit) or manual (explicit), depending on the situation.

i) Automatic Type Conversion (Implicit Conversion)

Java automatically converts a smaller data type to a larger data type. This is known as implicit type conversion or widening conversion, and it happens when there is no loss of information. For example, converting an `int` to a `float`.

Example:

```
int a = 10;
float b = a; // Automatic type conversion from int to float
System.out.println("Value of b: " + b);
```

In this example, the `int` value `a` is automatically converted to a `float` and stored in `b`. Since `float` can hold larger values and has a decimal point, no data is lost during this conversion.

When Does Automatic Type Conversion Occur?

- **When both data types are compatible:** For example, `int` to `float`, `float` to `double`.
- **When the target data type is larger than the source data type:** For example, `byte` to `int`, `short` to `long`.

Examples of Automatic Type Conversion:

- **byte → short → int → long → float → double**
- **char → int → long → float → double**

Example:

```
byte smallNumber = 25;
int largerNumber = smallNumber; // byte to int conversion
System.out.println("Larger Number: " + largerNumber);
```

In this example, a `byte` value is automatically converted to an `int` because an `int` has a larger capacity than a `byte`.

Conclusion

Type conversion is a fundamental concept in Java that allows you to work with different data types seamlessly. While automatic type conversion simplifies programming, explicit type conversion provides more control but requires caution to avoid data loss. Understanding when and how to use these conversions is crucial for efficient Java programming.

13. Type Casting in Java

Type casting in Java is the process of explicitly converting one data type into another. Unlike automatic type conversion, which is handled by the compiler, type casting requires manual intervention. It is generally used when you want to convert a larger data type into a smaller one, which may lead to data loss.

i) Narrowing Conversion (Explicit Casting)

Narrowing conversion, also known as explicit casting, involves converting a larger data type into a smaller one. This type of conversion may result in the loss of data or precision, so it must be done explicitly by the programmer.

Syntax:

```
type variableName = (type) value;
```

Example:

```
public class TypeCasting {
    public static void main(String[] args) {
        int a = 10;
        float b = 5.0f;
        int c = (int) b; // Narrowing conversion from float to int
        System.out.println(c); // Output: 5
    }
}
```

In this example, the `float` value `b` is explicitly cast to an `int`. Since `int` cannot hold decimal values, the fractional part of `b` is discarded, and only the integer part is stored in `c`. This is a narrowing conversion because you're converting from a larger data type (`float`) to a smaller one (`int`).

ii) When to Use Type Casting?

- **When you need precision:** If you want to control how data is handled, especially when dealing with calculations.
- **When converting incompatible types:** For example, converting from a `double` to an `int`.
- **When you know the value won't overflow:** Ensure that the conversion won't result in unexpected behavior, like truncating values.

Examples of Type Casting:

- **double → float**
- **float → int**
- **long → int**
- **int → byte**

Example:

```
double x = 9.78;
int y = (int) x; // Narrowing conversion from double to int
System.out.println("Value of y: " + y); // Output: 9
```

In this example, the `double` value `x` is cast to an `int`, and the decimal part is truncated.

Conclusion

Type casting in Java allows you to convert data types explicitly, giving you control over how your data is processed. However, use it carefully, as improper casting can lead to data loss or unexpected results. Understanding both narrowing and widening conversions will help you manage data types effectively in your Java programs.

14. Type Promotion in Expressions

Type promotion in Java is a crucial aspect of how expressions are evaluated when multiple data types are involved. It ensures consistency, precision, and compatibility across different operations, especially when smaller primitive types interact with larger ones. Understanding how and why type promotion occurs is fundamental for writing efficient and bug-free code in Java.

How Type Promotion Works

When you perform arithmetic or other operations in Java, the operands (values being operated on) may have different data types. Java will automatically promote smaller data types to larger ones based on a hierarchy to match the type of the expression's largest operand. This process is known as **type promotion**.

The Hierarchy of Data Types

Java has a predefined hierarchy of primitive data types that it follows during type promotion. The order, from smallest to largest, is as follows:

1. `byte`
2. `short`
3. `char`
4. `int`

5. `long`
6. `float`
7. `double`

When an expression contains mixed data types, Java will promote each smaller type to match the largest type in the expression, following this hierarchy.

Rules for Type Promotion

1. **Byte, Short, and Char:**
    - In expressions, `byte`, `short`, and `char` are first promoted to `int` before any further evaluation. This happens even if there's no `int` in the expression.
    - For example, when a `byte` is added to a `short`, both are promoted to `int` before the operation occurs.
2. **Int and Higher Data Types:**
    - If any operand is a `long`, the entire expression will be promoted to `long`.
    - If any operand is a `float`, the entire expression will be promoted to `float`.
    - If any operand is a `double`, the entire expression will be promoted to `double`.
3. **Mixed Data Types:**

    When two different data types are combined, Java promotes them to the larger of the two. For example, if you combine an `int` and a `float`, the result is promoted to a `float` since it has higher precision than `int`.

Example in Detail

Let's revisit the example code:

```
public class TypePromotion {
    public static void main(String[] args) {
        byte b = 42;
        char c = 'a';
        short s = 1024;
        int i = 50000;
        float f = 5.67f;
        double d = 0.1234;
        double result = (f * b) + (i / c) - (d * s);
        System.out.println(result);
    }
}
```

Breakdown of the Expression

The expression is: `(f * b) + (i / c) - (d * s)`

1. `(f * b)`:
    - `f` is a `float` and `b` is a `byte`.
    - The `byte` is promoted to `float`, and the result of this operation is a `float`.
2. `(i / c)`:
    - `i` is an `int` and `c` is a `char`.
    - The `char` is promoted to `int`, and the result of this division remains an `int`.
3. `(d * s)`:
    - `d` is a `double` and `s` is a `short`.
    - The `short` is promoted to `double`, making the result of this operation a `double`.
4. **Adding and Subtracting**:
    - When the results of these sub-expressions are combined, the largest data type (`double`) determines the overall result of the expression.
    - `(f * b)` (a `float`) and `(i / c)` (an `int`) are both promoted to `double` when combined with `(d * s)` (already a `double`).

Why Does Java Promote Types?

Type promotion helps prevent data loss, rounding errors, and other precision-related issues. If smaller types were not promoted, there would be a risk of overflow or incorrect results due to limited storage capacity. For example:

- **Overflow in Integer Operations**: If Java did not promote `byte` or `short` to `int`, their limited ranges (−128 to 127 for `byte` and −32,768 to 32,767 for `short`) could lead to overflow during arithmetic operations.
- **Precision in Floating-Point Operations**: When using `float` and `double`, Java promotes smaller types like `int` or `long` to these floating-point types to maintain precision. Without this, calculations involving floating-point numbers might lose their accuracy.

15. How Does Java Code Run?

Understanding how Java code runs is crucial for grasping the fundamentals of programming in Java. The execution process of Java code involves several stages, from writing the code to finally running it on a machine. Here's a detailed look at how Java code is executed:

i) Writing Code

The first step in any Java program is writing the source code. This is done in a `.java` file using a text editor or an Integrated Development Environment (IDE). The code is written in plain text and follows the syntax and rules of the Java programming language.

Example:

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

This simple program prints "Hello, World!" to the console.

ii) Compilation

Once the code is written, the next step is to compile it. Compilation is the process of converting the human-readable Java code into a machine-readable format. This is done using the Java compiler (`javac`), which translates the `.java` file into bytecode, a platform-independent intermediate representation of the code.

Steps:

1. **Command to Compile:**

```
javac HelloWorld.java
```

2. **Output:**

   After successful compilation, the compiler generates a `.class` file, such as `HelloWorld.class`. This file contains the bytecode, which can be executed on any platform that has a Java Virtual Machine (JVM).

iii) Execution

The final step is executing the compiled bytecode. This is where the Java Virtual Machine (JVM) comes into play. The JVM is responsible for loading the `.class` file and executing the bytecode on the host machine.

Steps:

1. **Command to Run:**

```
java HelloWorld
```

2. **Execution:**

   The JVM interprets the bytecode and translates it into machine code that the operating system understands. The program is then executed, and you see the output on the console.

Example:

```
Hello, World!
```

This is the output of the `HelloWorld` program.

Conclusion

The process of running a Java program involves writing the code, compiling it into bytecode, and then executing it using the JVM. This multi-step process ensures that Java code is platform-independent, meaning the same `.class` file can be executed on any device that has a JVM installed. Understanding these stages will help you appreciate how Java manages to be both powerful and versatile as a programming language.

Assignments

1. **Calculate the Average of Three Numbers**:

```
import java.util.Scanner;

public class Q1 {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter a:");
        int a = sc.nextInt();
        System.out.println("Enter b:");
        int b = sc.nextInt();
        System.out.println("Enter c:");
        int c = sc.nextInt();
        int average = (a + b + c) / 3;
        System.out.println("The Average of 3 numbers is: " + average);
    }
}
```

2. **Calculate the Area of a Square**:

```
import java.util.Scanner;

public class Q2 {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter side:");
        float side = sc.nextFloat();
        float area = side * side;
        System.out.println("Area of Square is: " + area);
    }
}
```

3. **Calculate the Total Cost of Items**:

```
import java.util.Scanner;

public class Q3 {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter the price of Pen:");
        float pen = sc.nextFloat();
        System.out.println("Enter the price of Pencil:");
        float pencil = sc.nextFloat();
        System.out.println("Enter the price of Eraser:");
        float eraser = sc.nextFloat();
        float total = pen + pencil + eraser;
        System.out.println("Bill: " + total);
        float totalGST = total + (0.18f * total);
        System.out.println("Bill with GST: " + totalGST);
    }
}
```

Conclusion:

In this chapter, we covered the basics of Java, including creating Java files, understanding the boilerplate code, working with variables and data types, and solving basic problems using these concepts. Mastering these fundamentals is essential as you move forward in your journey of learning Java with DSA.

Stay tuned for the next chapter, where we'll dive deeper into more complex topics and continue building our Java programming skills. Happy coding!

Related Posts in My Series:

DSA in Java Series:

- **Chapter 2: Operators in Java** – Learn about the different types of operators in Java, including arithmetic, relational, logical, and assignment operators, along with examples to understand their usage.
- **Chapter 33: Trie in Java** – Explore the implementation and use of Trie data structure in Java, a powerful tool for handling strings, with practical coding examples.

Other Series:

- **Full Stack Java Development** – Master the essentials of Java programming and web development with this series. Topics include object-oriented programming, design patterns, database interaction, and more.

- **Full Stack JavaScript Development** – Become proficient in JavaScript with this series covering everything from front-end to back-end development, including frameworks like React and Node.js.

Connect with Me

Stay updated with my latest posts and projects by following me on social media:

- **LinkedIn**: Connect with me for professional updates and insights.
- **GitHub**: **Explore my repositories** and contributions to various projects.
- **LeetCode**: **Check out my coding practice** and challenges.

Your feedback and engagement are invaluable. Feel free to reach out with questions, comments, or suggestions. Happy coding!

Rohit Gawande
Full Stack Java Developer | Blogger | Coding Enthusiast