Java Basics

In Java, the **main** method is the entry point of a program, where the execution of the program begins. It is a special method, and every Java application must have a **main** method in order to run. The syntax of the **main** method is always the same:

```java
package com.example;

public class Main {
    public static void main(String[] args) {
        // Main body
    }
}
```

Variables

In Java, data types can be categorized into three main types. The first type includes numbers, which can be divided into integers and floating-point numbers. Integers in Java are represented by the **`byte`**, **`short`**, **`int`**, and **`long`** types, while floating-point numbers are represented by **`float`** and **`double`**. For example, an integer can be declared as:

```java
byte smallNumber = 100;
short mediumNumber = 20000;
int age = 25;
long population = 7800000000L;
float price = 19.99f;
double distance = 256.4789;
```

Data Type	Range	Size (in bytes)
byte	-128 to 127	1
short	-32,768 to 32,767	2
int	-2^{31} to $2^{31} - 1$	4
long	-2^{63} to $2^{63} - 1$	8
float	-2^{-149} to $(2 - 2^{-23}) * 2^{127}$	4
double	-2^{-1074} to $(2 - 2^{-52}) * 2^{1023}$	8
boolean	true or false	1 bit to 1 byte
char	0 to $2^{16} - 1$	2

The second category involves characters and strings. The **`char`** type represents a single character, and the **`String`** type represents a sequence of characters. Here's how you can use them:

```
char letter = 'A';
String greeting = "Hello, World!";
```

Finally, the third category is booleans, which can only hold one of two possible values: **`true`** or **`false`**. It is commonly used in decision-making structures:

```
boolean isJavaFun = true;
boolean isRaining = false;
```

Conditional Statements

Conditional statements in Java, such as **`if`**, **`if-else`**, and **`switch-case`**, allow us to control the flow of execution based on certain conditions. These statements evaluate expressions and execute specific blocks of code depending on whether the conditions are true or false.

For instance, the **`if`** statement allows for conditional execution of code when a certain condition is met:

```
int age = 18;
if (age >= 18) {
    System.out.println("You are an adult.");
}
```

The **`if-else`** statement provides an alternative block of code that runs when the condition is false:

```
int age = 16;
if (age >= 18) {
    System.out.println("You are an adult.");
} else {
    System.out.println("You are a minor.");
}
```

When there are multiple conditions to check, the **`switch-case`** statement is often more efficient than using multiple **`if-else`** statements. It works by matching an expression against several possible values:

```java
int day = 3;
switch (day) {
    case 1:
        System.out.println("Monday");
        break;
    case 2:
        System.out.println("Tuesday");
        break;
    case 3:
        System.out.println("Wednesday");
        break;
    default:
        System.out.println("Invalid day");
}
```

Loops

Loops in Java allow us to repeat a block of code multiple times, and the type of loop we use depends on how many iterations we need and the conditions we are working with. The **`for`** loop is typically used when the number of iterations is known in advance, providing a concise way to loop through a range of values.

```java
for (int i = 0; i < 5; i++) {
    System.out.println("Iteration " + i);
}
```

On the other hand, the **`while`** loop is used when the number of iterations is not known beforehand and we want to continue looping as long as a certain condition is true:

```java
int count = 0;
while (count < 5) {
    System.out.println("Count: " + count);
    count++;
}
```

The **`do-while`** loop is similar to the **`while`** loop but ensures that the loop body executes at least once, even if the condition is false initially:

```java
int count = 0;
do {
    System.out.println("Count: " + count);
    count++;
} while (count < 5);
```

Lastly, the **`for-each`** loop is used when working with arrays, allowing us to iterate over each element without needing to manage the index explicitly:

```java
int[] numbers = {1, 2, 3, 4, 5};
for (int num : numbers) {
    System.out.println("Number: " + num);
}
```

Arrays

Arrays in Java are used when we need to store multiple values of the same data type in a single variable. They provide an efficient way to manage large amounts of data, as each element in the array is stored in a contiguous block of memory. For example, if we want to store the ages of several people, we can use an array:

```java
int[] ages = {25, 30, 22, 28, 35};
System.out.println("First person's age: " + ages[0]);
```

Arrays in Java can also be multidimensional, such as **two-dimensional arrays**. However, two-dimensional arrays are less frequently used and are typically only needed in more complex situations. A two-dimensional array can be thought of as an array of arrays:

```
int[][] matrix = {
    {1, 2, 3},
    {4, 5, 6},
    {7, 8, 9}
};
System.out.println("Element at position (1, 1): " + matrix[1][1]);
```

One common issue when working with arrays is the **Index Out of Bounds (IOB)** error, which occurs when trying to access an element outside the bounds of the array. To avoid this error, we should ensure that we don't exceed the array's length during iteration:

```
int[] numbers = {10, 20, 30};
for (int i = 0; i < numbers.length; i++) {
    System.out.println(numbers[i]);
}
```

Another issue is the **NullPointerException (NPE)**, which happens when we try to use an object reference that is null. We can avoid this by adding null checks before performing operations on an object:

```
String[] names = null;
if (names != null) {
    System.out.println(names[0]);
} else {
    System.out.println("Names array is null");
}
```

String

The **`String`** data type in Java is used to store a sequence of characters and comes with a variety of built-in methods that make it easier to work with text. These methods allow us to manipulate, compare, and transform strings efficiently. For example, we can use the

`length()` method to get the length of a string, or the **`toUpperCase()`** method to convert a string to uppercase:

```
String message = "hello world";
System.out.println("Length of message: " + message.length());
System.out.println("Uppercase message: " + message.toUpperCase());
```

Sometimes, we need to perform operations on strings that involve frequent modifications. In these cases, **`StringBuilder`** is a more efficient choice than **`String`**, as it is mutable, meaning it can be modified without creating new objects each time:

```
StringBuilder sb = new StringBuilder("hello");
sb.append(" world");
System.out.println(sb.toString());
```

Java also uses a **String pool**, a special area in memory where identical string values are stored to save memory. When you create a string with a literal value, Java checks if the same string already exists in the pool; if it does, it reuses the existing string, rather than creating a new one:

```
String str1 = "hello";
String str2 = "hello";
// true, because both point to the same string in the pool
System.out.println(str1 == str2);
```

When comparing strings, it's important to use the **`equals()`** method instead of the **`==`** operator. The **`==`** operator checks for reference equality (i.e., whether both variables point to the same memory location), while **`equals()`** checks for content equality (i.e., whether the strings have the same characters):

```java
String str1 = "hello";
String str2 = new String("hello");
 // true, because the contents are the same
System.out.println(str1.equals(str2));
// false, because they reference different objects
System.out.println(str1 == str2);
```