Java Data Types

Java is statically typed programming language which means variable types are known at the compile time.

- In Java, compiler knows exactly what types each variable holds and enforces correct usage during compilation. For example "`int x = "GfG";`" gives compiler error in Java as we are trying to store string in an integer type.
- Data types in Java are of different sizes and values that can be stored in a variable that is made as per convenience and circumstances to handle different scenarios or data requirements.

Why Data Types Matter in Java?

Data types matter in Java because of the following reasons, which are listed below:

- **Memory Efficiency:** Choosing the right type (byte vs int) saves memory.
- **Performance:** Proper types reduce runtime errors.
- **Code Clarity:** Explicit typing makes code more readable.

Java Data Type Categories

Java has two categories in which data types are segregated

1. Primitive Data Type: These are the basic building blocks that store simple values directly in memory. Examples of primitive data types are

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

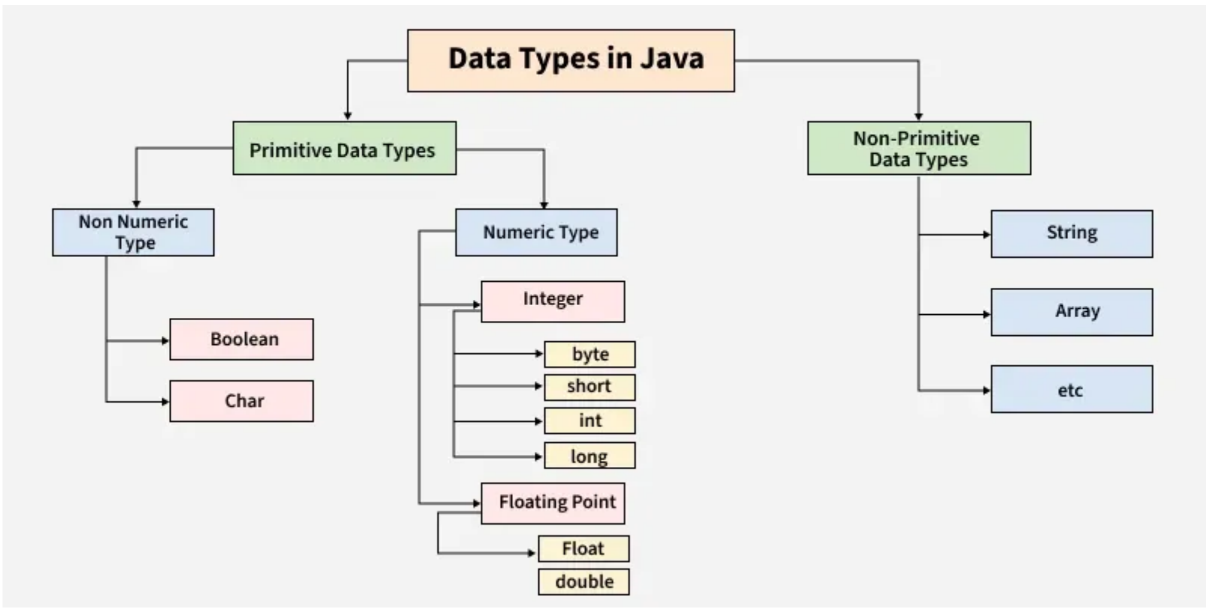
Note: *The Boolean with uppercase B is a wrapper class for the primitive boolean type.*

2. Non-Primitive Data Types (Object Types): These are reference types that store memory addresses of objects. Examples of Non-primitive data types are

- **String**
- **Array**

- Class
- Interface
- Object

The below diagram demonstrates different types of primitive and non-primitive data types in Java.



Primitive Data Types in Java

Primitive data store only single values and have no additional capabilities. There are **8 primitive data types**. They are depicted below in tabular format below as follows:

Type	Description	Default	Size	Example Literals	Range of values
boolean	true or false	false	JVM-dependent (typically 1 byte)	true, false	true, false
byte	8-bit signed integer	0	1 byte	(none)	-128 to 127
char	Unicode character(16 bit)	\u0000	2 bytes	'a', '\u0041', '\101', '\\', '\', '\n', 'β'	0 to 65,535 (unsigned)
short	16-bit signed integer	0	2 bytes	(none)	-32,768 to 32,767

Type	Description	Default	Size	Example Literals	Range of values
int	32-bit signed integer	0	4 bytes	-2,0,1	-2,147,483,648 to 2,147,483,647
long	64-bit signed integer	0L	8 bytes	-2L,0L,1L	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
float	32-bit IEEE 754 floating-point	0.0f	4 bytes	3.14f, -1.23e-10f	~6-7 significant decimal digits
double	64-bit IEEE 754 floating-point	0.0d	8 bytes	3.1415d, 1.23e100d	~15-16 significant decimal digits

1. boolean Data Type

The boolean data type represents a logical value that can be either true or false. Conceptually, it represents a single bit of information, but the actual size used by the virtual machine is implementation-dependent and typically at least one byte (eight bits) in practice. Values of the boolean type are not implicitly or explicitly converted to any other type using casts. However, programmers can write conversion code if needed.

Syntax:

boolean booleanVar;

Size : Virtual machine dependent (typically 1 byte, 8 bits)

Example: This example, demonstrating how to use boolean data type to display true/false values.

```java
// Demonstrating boolean data type
public class Gla {
    public static void main(String[] args) {
        boolean b1 = true;
        boolean b2 = false;

        System.out.println("Is Java fun? " + b1);
        System.out.println("Is fish tasty? " + b2);
    }
}
```

Output

```
Is Java fun? true

Is fish tasty? false
```

2. byte Data Type

The byte data type is an 8-bit signed two's complement integer. The byte data type is useful for saving memory in large arrays.

Syntax:

byte byteVar;

Size : 1 byte (8 bits)

Example: This example, demonstrating how to use byte data type to display small integer values.

```java
// Demonstrating byte data type
public class Gla {
   public static void main(String[] args) {
      byte a = 25;
      byte t = -10;

      System.out.println("Age: " + a);
      System.out.println("Temperature: " + t);
   }
}
```

Output

```
Age: 25

Temperature: -10
```

3. short Data Type

The short data type is a 16-bit signed two's complement integer. Similar to byte, a short is used when memory savings matter, especially in large arrays where space is constrained.

Syntax:

short shortVar;

Size : 2 bytes (16 bits)

Example: This example, demonstrates how to use short data type to store moderately small integer value.

```java
// Demonstrating short data types
```

```
public class Gla {
   public static void main(String[] args) {
      short num = 1000;
      short t = -200;

      System.out.println("Number of Students: " + num);
      System.out.println("Temperature: " + t);
   }
}
```

Output

```
Number of Students: 1000

Temperature: -200
```

4. int Data Type

It is a 32-bit signed two's complement integer.

Syntax:

int intVar;

Size : 4 bytes (32 bits)

Remember: *In Java SE 8 and later, we can use the int data type to represent an unsigned 32-bit integer, which has a value in the range [0, 2^{32} -1]. Use the Integer class to use the int data type as an unsigned integer.*

Example: This example demonstrates how to use int data type to display larger integer values.

```
// Demonstrating int data types
public class Gla {
   public static void main(String[] args) {
      int p = 2000000;
      int d = 150000000;

      System.out.println("Population: " + p);
      System.out.println("Distance: " + d);
   }
}
```

Output

```
Population: 2000000

Distance: 150000000
```

5. long Data Type

The long data type is a 64-bit signed two's complement integer. It is used when an int is not large enough to hold a value, offering a much broader range.

Syntax:

long longVar;

Size : 8 bytes (64 bits)

Remember: *In Java SE 8 and later, you can use the long data type to represent an unsigned 64-bit long, which has a minimum value of 0 and a maximum value of 2 ^{64} -1. The Long class also contains methods like comparing Unsigned, divide Unsigned, etc to support arithmetic operations for unsigned long.*

Example: This example demonstrates how to use long data type to store large integer value.

```
// Demonstrating long data type
public class Gla {
   public static void main(String[] args) {
      long w = 7800000000L;
      long l = 9460730472580800L;

      System.out.println("World Population: " + w);
      System.out.println("Light Year Distance: " + l);
   }
}
```

Output

```
World Population: 7800000000

Light Year Distance: 9460730472580800
```

6. float Data Type

The float data type is a single-precision 32-bit IEEE 754 floating-point. Use a float (instead of double) if you need to save memory in large arrays of floating-point numbers. The size of the float data type is 4 bytes (32 bits).

Syntax:

float floatVar;

Size : 4 bytes (32 bits)

Example: This example demonstrates how to use float data type to store decimal value.

```java
// Demonstrating float data type
public class Gla {
   public static void main(String[] args) {
      float pi = 3.14f;
      float gravity = 9.81f;

      System.out.println("Value of Pi: " + pi);
      System.out.println("Gravity: " + gravity);
   }
}
```

Output

```
Value of Pi: 3.14

Gravity: 9.81
```

7. double Data Type

The double data type is a double-precision 64-bit IEEE 754 floating-point. For decimal values, this data type is generally the default choice. The size of the double data type is 8 bytes or 64 bits.

Syntax:

double doubleVar;

Size : 8 bytes (64 bits)

Note: *Both float and double data types were designed especially for scientific calculations, where approximation errors are acceptable. If accuracy is the most prior concern then, it is recommended not to use these data types and use BigDecimal class instead.*

It is recommended to go through rounding off errors in java.

Example: This example demonstrates how to use double data type to store precise decimal value.

```java
// Demonstrating double data type
public class Gla {
   public static void main(String[] args) {
      double pi = 3.141592653589793;
      double an = 6.02214076e23;

      System.out.println("Value of Pi: " + pi);
      System.out.println("Avogadro's Number: " + an);
   }
}
```

Output

```
Value of Pi: 3.141592653589793

Avogadro's Number: 6.02214076E23
```

8. char Data Type

The char data type is a single 16-bit Unicode character with the size of 2 bytes (16 bits).

Syntax:

char charVar;

Size : 2 bytes (16 bits)

Example: This example, demonstrates how to use char data type to store individual characters.

```
// Demonstrating char data type
public class Gla{
   public static void main(String[] args) {
      char g = 'A';
      char s = '$';

      System.out.println("Grade: " + g);
      System.out.println("Symbol: " + s);
   }
}
```

Output

```
Grade: A

Symbol: $
```

Why is the Size of char 2 bytes in Java?

Unlike languages such as C or C++ that use the **ASCII character** set, Java uses the Unicode character set to support internationalization. Unicode requires more than 8 bits to represent a wide range of characters from different languages, including Latin, Greek, Cyrillic, Chinese, Arabic, and more. As a result, Java uses 2 bytes to store a char, ensuring it can represent any **Unicode** character.

Example: Here we are demonstrating how to use various primitive data types.

```
// Java Program to Demonstrate Char Primitive Data Type
class Gla
{
   public static void main(String args[])
```

```
  {

     // Creating and initializing custom character
     char a = 'G';

     // Integer data type is generally
     // used for numeric values
     int i = 89;

     // use byte and short
     // if memory is a constraint
     byte b = 4;

     // this will give error as number is
     // larger than byte range
     // byte b1 = 7888888955;

     short s = 56;

     // this will give error as number is
     // larger than short range
     // short s1 = 87878787878;

     // by default fraction value
     // is double in java
     double d = 4.355453532;

     // for float use 'f' as suffix as standard
     float f = 4.7333434f;

     // need to hold big range of numbers then we need
     // this data type
     long l = 12121;

     System.out.println("char: " + a);
     System.out.println("integer: " + i);
     System.out.println("byte: " + b);
     System.out.println("short: " + s);
     System.out.println("float: " + f);
     System.out.println("double: " + d);
     System.out.println("long: " + l);
  }
}
```

Output

```
char: G

integer: 89
```

```
byte: 4

short: 56

float: 4.7333436

double: 4.355453532

long: 12121
```

Non-Primitive (Reference) Data Types

The **Non-Primitive (Reference) Data Types** will contain a memory address of variable values because the reference types won't store the variable value directly in memory. They are strings, objects, arrays, etc.

1. Strings

Strings are defined as an array of characters. The difference between a character array and a string in Java is, that the string is designed to hold a sequence of characters in a single variable whereas, a character array is a collection of separate char-type entities. Unlike C/C++, Java strings are not terminated with a null character.

Syntax: Declaring a string

<String_Type> <string_variable> = "<sequence_of_string>";

Example: This example demonstrates how to use string variables to store and display text values.

```
// Demonstrating String data type
public class Gla {
    public static void main(String[] args) {
        String n = "Gla1";
        String m = "Hello, World!";

        System.out.println("Name: " + n);
        System.out.println("Message: " + m);
    }
}
```

Output

```
Name: Gla1

Message: Hello, World!
```

Note: String cannot be modified after creation. Use StringBuilder for heavy string manipulation

2. Class

A Class is a user-defined blueprint or prototype from which objects are created. It represents the set of properties or methods that are common to all objects of one type. In general, class declarations can include these components, in order:

- **Modifiers** : A class can be public or has default access. Refer to access specifiers for classes or interfaces in Java
- **Class name:** The name should begin with an initial letter (capitalized by convention).
- **Superclass(if any):** The name of the class's parent (superclass), if any, preceded by the keyword extends. A class can only extend (subclass) one parent.
- **Interfaces(if any):** A comma-separated list of interfaces implemented by the class, if any, preceded by the keyword implements. A class can implement more than one interface.
- **Body:** The class body is surrounded by braces, { }.

Example: This example demonstrates how to create a class with a constructor and method, and how to create an object to call the method.

```java
// Demonstrating how to create a class
class Car {
   String model;
   int year;

   Car(String model, int year) {
      this.model = model;
      this.year = year;
   }

   void display() {
      System.out.println(model + " " + year);
   }
}

public class Gla {
   public static void main(String[] args) {
      Car myCar = new Car("Toyota", 2020);
      myCar.display();
   }
}
```

Output

```
Toyota 2020
```

3. Object

An Object is a basic unit of Object-Oriented Programming and represents real-life entities. A typical Java program creates many objects, which as you know, interact by invoking methods. An object consists of :

- **State**: It is represented by the attributes of an object. It also reflects the properties of an object.
- **Behavior**: It is represented by the methods of an object. It also reflects the response of an object to other objects.
- **Identity**: It gives a unique name to an object and enables one object to interact with other objects.

Example: This example demonstrates how to create the object of a class.

```java
// Define the Car class
class Car {
   String model;
   int year;

   // Constructor to initialize the Car object
   Car(String model, int year) {
      this.model = model;
      this.year = year;
   }
}

// Main class to demonstrate object creation
public class Gla {
   public static void main(String[] args) {
      // Create an object of the Car class
      Car myCar = new Car("Honda", 2021);

      // Access and print the object's properties
      System.out.println("Car Model: " + myCar.model);
      System.out.println("Car Year: " + myCar.year);
   }
}
```

Output

```
Car Model: Honda

Car Year: 2021
```

4. Interface

Like a class, an interface can have methods and variables, but the methods declared in an interface are by default abstract (only method signature, no body).

- Interfaces specify what a class must do and not how. It is the blueprint of the class.
- An Interface is about capabilities like a Player may be an interface and any class implementing Player must be able to (or must implement) move(). So it specifies a set of methods that the class has to implement.
- If a class implements an interface and does not provide method bodies for all functions specified in the interface, then the class must be declared abstract.
- A Java library example is Comparator Interface. If a class implements this interface, then it can be used to sort a collection.

Example: This example demonstrates how to implement an interface.

```java
// Demonstrating the working of interface
interface Animal {
   void sound();
}

class Dog implements Animal {
   public void sound() {
      System.out.println("Woof");
   }
}

public class InterfaceExample {
   public static void main(String[] args) {
      Dog myDog = new Dog();
      myDog.sound();
   }
}
```

Output

```
Woof
```

5. Array

An Array is a group of like-typed variables that are referred to by a common name. Arrays in Java work differently than they do in C/C++. The following are some important points about Java arrays.

- In Java, all arrays are dynamically allocated. (discussed below)
- Since arrays are objects in Java, we can find their length using member length. This is different from C/C++ where we find length using size.
- A Java array variable can also be declared like other variables with [] after the data type.
- The variables in the array are ordered and each has an index beginning with 0.
- Java array can also be used as a static field, a local variable, or a method parameter.
- The **size** of an array must be specified by an int value and not long or short.
- The direct superclass of an array type is Object.
- Every array type implements the interfaces Cloneable and java.io.Serializable.

Example: This example demonstrates how to create and access elements of an array.

```
// Demonstrating how to create an array
public class Gla {
   public static void main(String[] args) {
      int[] num = {1, 2, 3, 4, 5};
      String[] arr = {"Gla1", "Gla2", "Gla3"};

      System.out.println("First Number: " + num[0]);
      System.out.println("Second Fruit: " + arr[1]);
   }
}
```

Output

```
First Number: 1

Second Fruit: Gla2
```

Primitive vs Non-Primitive Data Types

The table below demonstrates the difference between Primitive and Non-Primitive Data types

Aspect	Primitive	Non-Primitive
Memory	Stored on the stack	Stored on the heap

Aspect	Primitive	Non-Primitive
Speed	Primitive data types are faster	Non-primitive data types are slower
Example	int x = 5;	String s = "Gla";

Understanding Java’s data types is fundamental to efficient programming. Each data type has specific use cases and constraints, making it essential to choose the right type for the task at hand. This ensures optimal memory usage and program performance while leveraging Java’s strong typing system to catch errors early in the development process.