Data Types in Java

Java is a statically typed language, meaning every variable must be declared with a data type. Data types are categorized into two main groups: **Primitive** and **Non-Primitive**.

A. Primitive Data Types

These are the most basic data types built into the language. They hold simple values directly in memory. There are 8 primitive types:

Type	Size	Description	Example
byte	1 byte	Whole numbers from -128 to 127	`byte b = 100;`
short	2 bytes	Whole numbers from -32,768 to 32,767	`short s = 5000;`
int	4 bytes	Standard integer numbers (Approximately +/- 2 Billion)	`int i = 100000;`
long	8 bytes	Large integer numbers (suffix 'L') (Approximately +/- 9 Quintillion)	`long l = 100000L;`
float	4 bytes	Fractional numbers (suffix 'f')	`float f = 5.75f;`
double	8 bytes	Fractional numbers (more precise)	`double d = 19.99;`
boolean	1 bit*	Represents true or false	`boolean isFun = true;`
char	2 bytes	Single character (Unicode)	`char c = 'A';`

B. Non-Primitive (Reference) Data Types

These refer to objects. They do not store the value directly but store a reference (memory address) to where the object is located.

- **Examples:** `String`, `Arrays`, `Classes`, `Interfaces`.
- **Key Difference:** Non-primitive types can be `null`, whereas primitives cannot.

Type Conversions (Casting)

Type conversion is the process of converting a value from one data type to another.

A. Implicit Conversion (Widening Casting)

This happens automatically when passing a smaller size type to a larger size type. It is safe because there is no loss of data.

- **Order:** `byte` -> `short` -> `char` -> `int` -> `long` -> `float` -> `double`

```
int myInt = 9;
double myDouble = myInt; // Automatic casting: int to double

System.out.println(myDouble); // Output: 9.0
```

B. Explicit Conversion (Narrowing Casting)

This must be done manually by placing the type in parentheses `()` in front of the value. It is required when converting a larger type to a smaller type because data might be lost (e.g., losing the decimal part).

```
double myDouble = 9.78;
int myInt = (int) myDouble; // Manual casting: double to int

System.out.println(myInt); // Output: 9 (0.78 is lost)
```

Wrapper Classes

Java is an object-oriented language, but primitive data types (like `int`, `char`) are not objects. Wrapper classes provide a way to use primitive data types as objects. This is crucial when using Collection frameworks (like `ArrayList`, `HashMap`) which only store objects, not primitives.

Each primitive type has a corresponding wrapper class:

Primitive Type	Wrapper Class
byte	**Byte**
short	**Short**
int	**Integer**
long	**Long**
float	**Float**
double	**Double**
boolean	**Boolean**
char	**Character**

Autoboxing and Unboxing

Since Java 5, the compiler automatically converts between primitives and their wrapper classes.

A. Autoboxing

The automatic conversion of a **primitive type** into its corresponding **wrapper class**.

```
int a = 20;
Integer val = a; // Autoboxing: int converted to Integer automatically
// Behind the scenes, the compiler does: Integer val = Integer.valueOf(a);
```

B. Unboxing

The automatic conversion of a **wrapper class** object back into its corresponding **primitive type**.

```
Integer val = new Integer(10);
int a = val; // Unboxing: Integer object converted to int
// Behind the scenes, the compiler does: int a = val.intValue();
```

Real-world Example of Autoboxing/Unboxing:

```
import java.util.ArrayList;

public class Main {
    public static void main(String[] args) {
        // ArrayLists can only store Objects (Wrapper Classes)
        ArrayList<Integer> numbers = new ArrayList<Integer>();

        // Autoboxing: passing primitive 5, converted to Integer object
        numbers.add(5);

        // Unboxing: getting Integer object, converted to primitive int
        int num = numbers.get(0);
    }
}
```

String Conversions

While `String` is not a primitive, converting between Strings and data types is a daily task for developers.

- **String to Primitive:**
  - `int a = Integer.parseInt("123");`
  - `double d = Double.parseDouble("12.34");`
- **Primitive to String:**
  - `String s = String.valueOf(100);`

- `String s = Integer.toString(100);`
- `String s = 100 + "";` (The lazy way, but commonly used).

Default Values (Local vs. Instance)

- **Instance Variables (Fields):** Have default values (int = 0, boolean = false, Object = null).
- **Local Variables (Inside methods):** Do **not** have default values. You must initialize them before use, or the code won't compile.

```
public void myMethod() {
    int x;
    System.out.println(x); // COMPILER ERROR: Variable x might not have been initialized
}
```

Primitive Data Types vs Wrapper Classes

Feature	Primitive Types	Wrapper Classes
Example	`int`, `double`	`Integer`, `Double`
Stored as	Raw values	Objects
Memory usage	Very low	Higher (object overhead)
Speed	Faster	Slower
Null allowed	❌ No	☑ Yes
Used in collections	❌ No	☑ Yes

Why primitives are faster

☑ 0. Memory Consumption (The "Weight" Difference)

Primitives are extremely lightweight because they store only the raw value. Wrapper classes are full-blown Objects, which carry a lot of extra "baggage" called object overhead.

Type	Memory Used (Approx.)	What's being stored?
int (Primitive)	4 bytes	Just the number.
Integer (Wrapper)	16 - 24 bytes	Object header, metadata, and the 4-byte value.

☑ 1. No object creation

```java
int x = 10;          // stored directly
Integer y = 10;    // object created (autoboxing)
```

Wrapper classes involve:

- Object creation
- Heap allocation
- Garbage collection

All of these **cost CPU time and memory**.

☑ 2. No Autoboxing / Unboxing overhead

```java
Integer a = 10;
Integer b = 20;
int sum = a + b;   // unboxing happens
```

Behind the scenes:

```java
int sum = a.intValue() + b.intValue();
```

➡ This extra work slows execution, especially in loops.

☑ 3. Better cache & CPU friendliness

- Primitives are stored **contiguously**
- CPU cache usage is more efficient
- Wrappers are **scattered objects** in heap

Example: Performance Difference

```java
long start = System.nanoTime();

int sum = 0;
for (int i = 0; i < 1_000_000; i++) {
    sum += i;
}

long end = System.nanoTime();
System.out.println(end - start);
```

vs

```
long start = System.nanoTime();

Integer sum = 0;
for (Integer i = 0; i < 1_000_000; i++) {
    sum += i; // boxing & unboxing every iteration
}

long end = System.nanoTime();
System.out.println(end - start);
```

📌 The wrapper version can be **several times slower** and consume more memory.

But wrappers have advantages

Use **wrapper classes** when you need:

- Collections (`ArrayList<Integer>`)
- `null` values
- Generics
- Utility methods (`Integer.parseInt()`)
- Synchronization (`AtomicInteger`)

Summary

> Primitive data types are faster and more memory-efficient than wrapper classes because they store values directly without object overhead. Wrapper classes introduce extra cost due to object creation, heap allocation, garbage collection, and boxing/unboxing.

Recommendation

- **Use primitives** for:
  - Loops
  - Calculations
  - Performance-critical code
- **Use wrappers** for:
  - Collections
  - APIs
  - When `null` is required