Java Datatypes Intro

Here is an introduction to data types in Java 21.

Data Types in Java 21:

Java is a statically typed language, meaning every variable must have a declared type known at compile time. Java 21 does introduce var for local type inference, but the underlying type system remains strongly typed.
Java's data types fall into two broad categories: primitive types and reference types.

Primitive Types:

Java has 8 primitive types, which hold raw values directly in memory (on the stack):

byte: An 8-bit signed integer, useful for saving memory in large arrays. Its values range from -128 to 127.

- **short**: A 16-bit signed integer, offering a balance between memory efficiency and range compared to byte and int. Its values range from -32,768 to 32,767.
- **int**: A 32-bit signed integer and one of the most commonly used data types. Its values range from -2,147,483,648 to 2,147,483,647.
- **long**: A 64-bit signed integer, used when int is not large enough to hold the required value. Its values range from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807.
- **float**: A single-precision 32-bit floating-point type for storing fractional numbers, sufficient for 6 to 7 decimal digits of precision.
- **double**: A double-precision 64-bit floating-point type, generally preferred for its higher precision (15 to 16 decimal digits) in mathematical and scientific computations.
- **char**: A single 16-bit Unicode character, which can store letters, numbers, or symbols.
- **boolean**: A logical data type that can only have two possible values: true or false. It is commonly used for conditional logic and flow control.

Examples:

```
      int age = 30;
      double price = 19.99;
      char grade = 'A';
      boolean isActive = true;
l     ong population = 8_000_000_000L;  // underscores aid readability
```

Reference Types:

Reference types store a reference (memory address) to an object on the heap, rather than the value itself.

Strings are the most common reference type and are immutable in Java:

```
String name = "Alice";
String greeting = "Hello, " + name; // "Hello, Alice"
```

Arrays hold multiple values of the same type:

```
int[] scores = {90, 85, 78};
String[] colors = new String[3];
```

Classes and Objects — any class you define (or use from the standard library) becomes a reference type:

```
ArrayList<String> names = new ArrayList<>();
names.add("Bob");
```

Type Inference with var (Java 10+, still relevant in Java 21):

Java 10 introduced var for local variables, letting the compiler infer the type. This is purely a compile-time feature — the variable still has a fixed, strong type:

```
var count = 42;          // inferred as int
var message = "Hello";   // inferred as String
var list = new ArrayList<String>(); // inferred as ArrayList<String>
```

You can only use var for local variables with an initializer — not for fields, method parameters, or return types.

Wrapper Classes:

In Java 21, wrapper classes are immutable classes that "wrap" Java's primitive data types (like int and double) into objects. They are essential for enabling primitives to be used in contexts that require objects, such as Java Collections and generics.

Core Wrapper Classes:

Each of the eight primitive types in Java has a corresponding wrapper class in the java.lang package.

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

The numeric wrapper classes (Byte, Short, Integer, Long, Float, Double) all extend the abstract class java.lang.Number.

Key Features and Usage:

Wrapper classes facilitate several core Java features:

Object Compatibility: They allow primitive values to be stored in collection frameworks (e.g., ArrayList<Integer>), which can only hold objects.

Autoboxing and Unboxing: Java automatically converts between primitives and their wrapper objects, reducing boilerplate code.

Autoboxing (primitive to wrapper): List<Integer> list = new ArrayList<>(); list.add(42); (the int 42 is autoboxed into an Integer object).

Unboxing (wrapper to primitive): int value = list.get(0); (the Integer object is unboxed into an int value).

Utility Methods: Wrapper classes provide a rich set of static methods for type conversion, string parsing (e.g., Integer.parseInt(String s)), and handling special values like NaN.

Null Values: Unlike primitives, wrapper class instances can be assigned a null value, useful for representing optional data.

Thread Safety: All wrapper classes are immutable, making them inherently thread-safe for use in concurrent applications.

Enhancements in Java 21:

In Java 21, wrapper classes integrate seamlessly with modern language features:

Pattern Matching: They work with enhanced pattern matching for switch statements and instanceof checks.

Virtual Threads: Their interaction is optimized with Java 21's new virtual threads system, ensuring good performance in high-throughput applications.

Best Practices:

Use valueOf(): The static factory method valueOf() is the recommended way to create wrapper objects, as it utilizes internal caching for common values (e.g., Integer values from -128 to 127) to improve memory efficiency.

Use .equals() for Comparison: The == operator compares object references, which can lead to unexpected results with cached vs non-cached wrapper objects. Use the .equals() method to compare the actual values.

Avoid null in Loops: Autoboxing and unboxing operations within performance-critical code or tight loops can create significant performance overhead due to memory usage and garbage collection; use primitives in such cases.

New in Java 21: Record Types & Pattern Matching:

Java 21 (an LTS release) brings a few type-related features worth knowing: Records are compact, immutable data carriers — a concise alternative to writing boilerplate POJOs:

```
javarecord Point(int x, int y) {}

Point p = new Point(3, 4);
System.out.println(p.x()); // 3
```

Pattern Matching for switch (finalized in Java 21) lets you branch on type elegantly:

```
javaObject obj = "Hello";

String result = switch (obj) {
    case Integer i -> "Int: " + i;
    case String s  -> "String: " + s;
    default        -> "Other";
};
```

Key Takeaways:

Primitives are fast, value-based, and have no methods.

Reference types are objects with methods, stored on the heap, and default to null.

Use **var** to reduce boilerplate for local variables when the type is obvious from context.

Wrapper classes bridge primitives and the object-oriented world.

Java 21 adds expressive features like records and pattern matching that build on the type system.