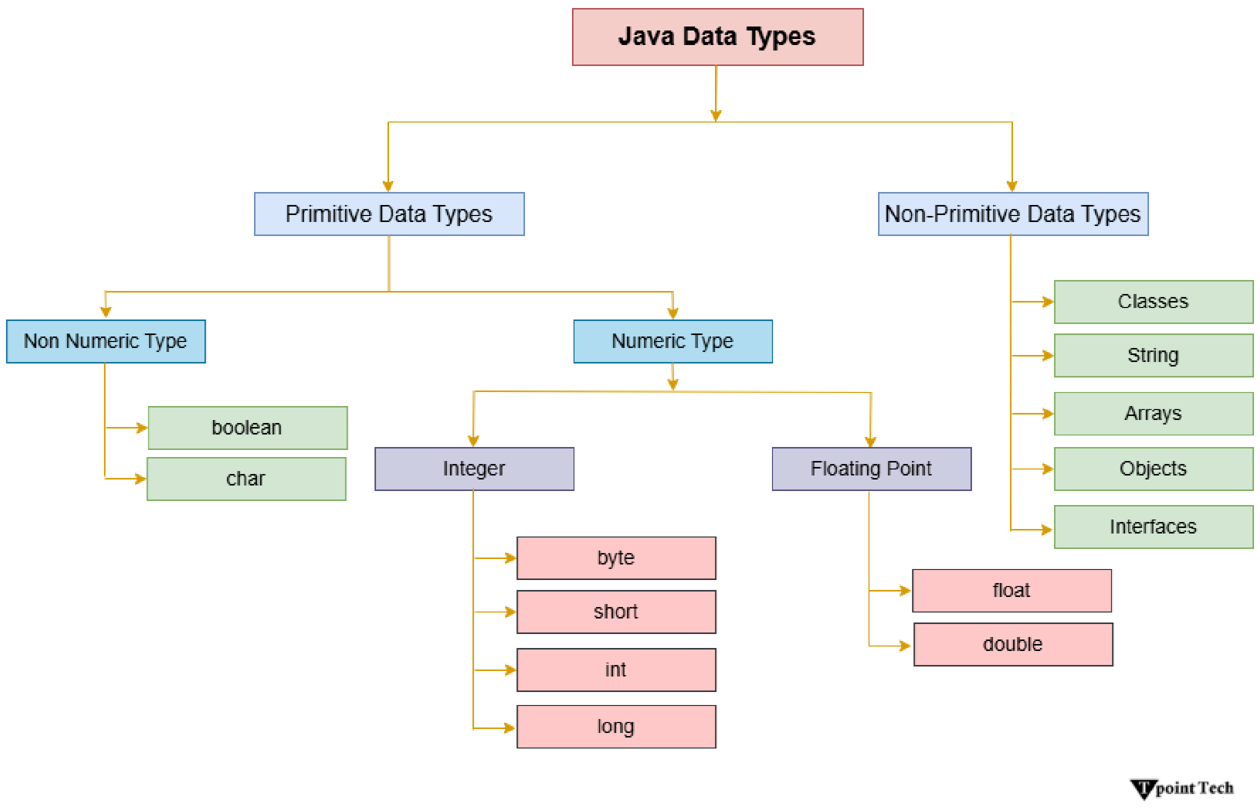
JAVA DATA TYPES

Data Types in Java

Last Updated : 24 Jan 2026

In programming languages, data types specify the different sizes and values that can be stored in the variable or constants. Each data type is predefined, which makes Java a statically and strongly typed language. There are the following two types of data types in Java.

1. Primitive Data Types: The primitive data types include boolean, char, byte, short, int, long, float and double.
2. Non-Primitive Data Types: The non-primitive data types include Classes, Interfaces, String, and Arrays.



Let's understand in detail.

Java Primitive Data Types

In Java, primitive data types are the building blocks of data manipulation. These are the basic data types.

In Java, there are mainly eight primitive data types which are as follows.

1. boolean data type
2. char data type
3. byte data type
4. short data type
5. int data type
6. long data type
7. float data type
8. double data type

Java is a statically typed programming language. It means all variables must be declared before their use. That is why we need to declare the variable's type and name. Let's discuss each data type one by one.

1. Boolean Data Type

In Java, the boolean data type represents a single bit of information with two possible states: true or false. The size of the Boolean data type is 1 byte (8 bits).

It is used to store the result of logical expressions or conditions. Unlike other primitive data types like int or double, boolean does not have a specific size or range. It is typically implemented as a single bit, although the exact implementation may vary across platforms.

Syntax:

```
boolean flag;
```

Example

```
Boolean a = false;
Boolean b = true;
```

2. Byte Data Type

The byte data type in Java is a primitive data type that represents an 8-bits signed two's complement integer. It has a range of values from -128 to 127. Its default value is 0.

The byte data type is commonly used when working with raw binary data or when memory conservation is a concern, as it occupies less memory than larger integer types like int or long.

Syntax:

```
byte size;
```

Example

```
byte a = 10;
byte b = -20;
```

3. Short Data Type

The short data type in Java is a primitive data type that represents a 16-bits signed two-complement integer. Its range of values is -32,768 to 32,767.

Similar to the byte data type, short is used when memory conservation is a concern, but more precision than byte is required. Its default value is 0.

Syntax:

```
short var;
```

Example

```
short a = 10000;
short b = -5000;
```

4. int Data Type

The int data type in Java is a primitive data type that represents a 32-bits signed two's complement integer. It has a range of values from -2,147,483,648 to 2,147,483,647.

The int data type is one of the most commonly used data types. It is typically used to store whole numbers without decimal points. Its default value is 0.

Syntax:

```
int myInt = 54;
```

In Java, int variables are declared using the int keyword. For example, int myInt = 54 ; declares an int variable named myInt and initializes it with the value 54. int variables can be used in mathematical expressions, assigned to other int variables, and used in conditional statements.

Remember: In Java SE 8 and later versions, we can use the int data type to represent an unsigned 32-bit integer. It 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

```
int a = 100000;
int b = -200000;
```

5. long Data Type

The long data type in Java is a primitive data type that represents a 64-bits signed two's complement integer. It has a wider range of values than int, ranging from - 9,223,372,036,854,775,808 to 9,223,372,036,854,775,807. Its default value is 0.0L or 0.0l.

The long data type is used when the int data type is not large enough to hold the desired value or when a larger range of integer values is needed.

Syntax:

```
long num = 15000000000L;
long num = 9,223,372,036,854,775l
```

The long data type is commonly used in applications where large integer values are required, such as in scientific computations, financial applications, and systems programming. It provides greater precision and a larger range than int, making it suitable for scenarios where int is insufficient.

Example

```
long a = 5000000L;
long b = -6000000L;
```

Note: In Java SE 8 and later versions, we can use the long data type to represent an unsigned 64-bit long number. It has a minimum value of 0 and a maximum value of 2^{64}-1.

6. float Data Type

The float data type in Java is a primitive data type that represents single-precision 32-bits IEEE 754 floating-point numbers. It can represent a wide range of decimal values, but it is not suitable for precise values such as currency. Its default value is 0.0f or 0.0F.

The float data type is useful for applications where a higher range of values is needed and precision is not critical.

Syntax:

```
float num = 67;
```

One of the key characteristics of the float data type is its ability to represent a wide range of values, both positive and negative, including very small and very large values. However, due to its limited precision (approximately 6-7 significant decimal digits), it is not suitable for applications where exact decimal values are required.

Example

```
float f = 234.5f;
```

7. double Data Type

The double data type in Java is a primitive data type that represents double-precision 64-bits IEEE 754 floating-point numbers. Its default value is 0.0. It provides a wider range of values and greater precision compared to the float data type, which makes it suitable for applications where accurate representation of decimal values is required.

Syntax:

```
double num = 75.658d;
```

One of the key advantages of the double data type is its ability to represent a wider range of values with greater precision compared to float. It can accurately represent values with up to approximately 15-16 significant decimal digits, making it suitable for applications that require high precision, such as financial calculations, scientific computations, and graphics programming.

Example

```
double d = 12.3;
```

Note: If accuracy is the most important concern, it is suggested not to use float and double data types; use the BigDecimal class instead.

8. char Data Type

The char data type in Java is a primitive data type that represents a single 16-bits Unicode character. It can store any character from the Unicode character set, which allows Java to support the internationalisation and representation of characters from various languages and writing systems.

Syntax:

```
char alphabets = 'J';
char a = 60, b = 61, c = 62;
```

The char data type is commonly used to represent characters, such as letters, digits, and symbols. It can also be used to perform arithmetic operations, as the Unicode values of characters can be treated as integers.

Example

```
char c = 'A';
```

For example, we can perform addition or subtraction operations on char variables to manipulate their Unicode values.

Type	Default Values	Size	Range of Values	Example

boolean	false	8 bits	true, false	true, false, 0, 1
byte	0	8 bits	-128 to 127	-
char	\u0000	16 bits	Characters Representation of ASCII values 0 to 255	'z', '\u0041', '\11', '\\', '\'', '\n'
short	0	16 bits	-32,768 to 32,767	-
int	0	32 bits	-2,147,483,648 to 2,147,483,647	-5,-1,0,6,8
long	0	64 bits	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	-7L,-2L,0L,1L,2L
float	0.0f	32 bits	up to 7 decimal digits	3.14f, 89.09F, -9.3F

double	0.0	64 bits	up to 16 decimal digits	6.98765e300d , -567899e-600 d , 2e2d

Java Primitive Data Type Example

File Name: Main.java

Example

```
public class Main
{
   public static void main(String args[])
   {
      boolean flag = true;
      char ch = 'z';
      int num = 1234;
      byte size = 2;
      short srt = 78;
      double value = 2.4546778;
      float temp = 3.8f;
      long val = 1888889;
      System.out.println("boolean: " + flag);
      System.out.println("char: " + ch);
      System.out.println("integer: " + num);
      System.out.println("byte: " + size);
      System.out.println("short: " + srt);
      System.out.println("float: " + value);
      System.out.println("double: " + temp);
      System.out.println("long: " + val);
   }
}
```

Output:

```
boolean: true
char: z
integer: 1234
byte: 2
```

```
short: 78
float: 2.4546778
double: 3.8
long: 1888889
```

Non-Primitive Data Types in Java

In Java, non-primitive data types are also known as reference data types. It is used to store complex objects rather than simple values. Reference data types store references or memory addresses that point to the location of the object in memory. This distinction is important because it affects how these data types are stored, passed, and manipulated in Java programs.

1. Class

One common non-primitive data type in Java is the class. Classes are used to create objects, which are instances of the class. A class defines the properties and behaviours of objects, including variables (fields) and methods.

For example, you might create a Person class to represent a person, with variables for the person's name, age, and address, and methods to set and get these values.

Syntax:

```
class Main
{
//code
}
```

2. Interface

Interfaces are another important non-primitive data type in Java. An interface defines a contract for what a class implementing the interface must provide without specifying how it should be implemented. Interfaces are used to achieve abstraction and multiple inheritance.

Syntax:

```
// interface
interface Shape {
  public void draw(); // interface method (does not have a body)
  public void color(); // interface method (does not have a body)
}
```

3. Arrays

Arrays are a fundamental non-primitive data type in Java that allows you to store multiple values of the same type in a single variable. Arrays have a fixed size, which is specified when the array is created and can be accessed using an index. Arrays are commonly used to store lists of values or to represent matrices and other multi-dimensional data structures.

Syntax:

```
int[] arr = { 1, 2, 3, 4, 5 };
```

4. String

In Java, a string is a sequence of characters. In simple words, we can define a string 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. Note that, unlike C/C++, Java strings are not terminated with a null character.

Syntax:

```
// string without using new operator
String s = "tpointtech";
// String using new operator
String str = new String("Austria");
```

5. enum

Java also includes other non-primitive data types, such as enums and collections. Enums are used to define a set of named constants, providing a way to represent a fixed set of values. Collections are a framework of classes and interfaces that provide dynamic data structures such as lists, sets, and maps, which can grow or shrink in size as needed.

Syntax:

```
enum Grade {
  FIRST,
  SECOND,
  THIRD
}
```

Overall, non-primitive data types in Java are essential for creating complex and flexible programs. They allow us to create and manipulate objects, define relationships between objects, and represent complex data structures.

Java Non-Primitive Data Type Example

File Name: Main.java

Example

```
enum Color {
    RED,
    GREEN,
    BLUE;
}
public class Main {
    public static void main(String[] args) {
        String str = "Hello";
        int[] arr = { 1, 2, 3, 4, 5 };
        Color clr = Color.BLUE;
        System.out.println(clr);
        System.out.println(str);
        for (int member: arr) {
        System.out.print(member+", ");
        }
    }
}
```

Output:

```
BLUE
Hello
1, 2, 3, 4, 5
```

Why char uses 2 bytes in Java, and what is \u0000 ?

It is because Java uses the Unicode system, not the ASCII code system. The \u0000 is the lowest range of the Unicode system. To get a detailed explanation of a Unicode, visit the next page.

1️⃣ What encoding was used *before Unicode* in Java?

Before Unicode became standard, systems mainly used **ASCII-based and platform-dependent encodings**.

Common pre-Unicode encodings:

- **ASCII** (7-bit)
- **Extended ASCII** (8-bit)
- **ISO-8859 (Latin-1, Latin-2, etc.)**
- **Platform-specific encodings**
    - Windows-1252 (Windows)
    - Shift-JIS (Japanese)
    - GB2312 (Chinese)

➡️ These were **not universal**.

2️⃣ How early Java handled characters?

Early programming languages (like C, C++) relied on:

- **1 byte = 1 character**
- Encoding depended on **operating system / locale**

This caused serious problems when moving programs between countries.

3️⃣ Limitations of pre-Unicode encodings ❌

🔴 1. Limited Character Set

- ASCII supports only **128 characters**
- Extended ASCII supports **256 characters**

➡️ Not enough for:

- Indian languages
- Chinese
- Arabic
- Japanese
- Emoji

🔴 2. Platform Dependency

Same byte value could mean different characters on different systems.

Example:

- Byte `0xE9`
    - é in ISO-8859-1

    - something else in another encoding

➡️ **“Write once, run anywhere” breaks**

🔴 3. No Global Language Support

One encoding = one language group.

➡️ Cannot represent multiple languages in one document easily.

🔴 4. Data Corruption

If file encoding mismatched:

- Characters appeared as `???` or garbage symbols

4️⃣ Why Java moved to Unicode ✅

Java was designed to be:

- **Platform-independent**
- **Global**
- **Internet-ready**

Unicode solved all earlier problems.

5️⃣ What is Unicode?

Unicode is a **universal character set** that assigns:

- A **unique number (code point)** to every character
- Across **all languages and symbols**

Examples:

- `A` → U+0041
- `अ` → U+0905
- `你` → U+4F60
- 😀 → U+1F600

6️⃣ Advantages of Unicode in Java 🌍

✅ 1. Global Language Support

Java can represent:

- English
- Hindi
- Chinese
- Arabic
- Emoji
  ➡️ All in **one program**

✅ 2. Platform Independence

Same character representation on:

- Windows
- Linux
- macOS

➡️ True **Write Once, Run Anywhere**

✅ 3. Fixed Internal Representation

Java internally uses **Unicode (UTF-16)** for `char` and `String`.

➡️ Predictable and consistent behavior.

✅ 4. Easy Internationalization (i18n)

Java supports:

- Multiple languages
- Locale-based formatting

7️⃣ Summary Table (Exam-friendly)

Before Unicode	Unicode in Java
ASCII / ISO-8859	UTF-16
Limited characters	Supports all languages

Platform dependent	Platform independent
One language at a time	Multiple languages together
Data corruption risk	No corruption

8️⃣ One-line interview answer 🎯

> “Before Unicode, Java relied on ASCII and platform-dependent encodings which supported limited characters and caused portability issues. Java adopted Unicode to support all world languages consistently and maintain platform independence.”

9️⃣ Short exam answer (3–4 lines)

> Before Unicode, systems used ASCII and extended ASCII encodings which supported limited characters and were platform dependent. These encodings could not support global languages. Java adopted Unicode to provide universal character representation, avoid data corruption, and ensure platform independence.

If you want, I can also:

- Explain **UTF-8 vs UTF-16 in Java**
- Show **Java String memory behavior**
- Provide **exam diagrams**

1️⃣ Why do we need Unicode *Transformation* Formats?

Unicode itself is **only a numbering system**.

Example:

- `A` → **U+0041**
- `अ` → **U+0905**
- 😀 → **U+1F600**

👉 These numbers **cannot be stored directly** in memory or files.
They must be **transformed into bytes**.

➡️ That transformation is done by **UTFs**.

2️⃣ What does "Unicode Transformation Format" mean?

Unicode Transformation Format (UTF) means:

> **A rule that defines how Unicode code points are converted into one or more bytes.**

- **Unicode** → character numbers
- **Transformation** → conversion
- **Format** → byte representation

3️⃣ Why are there *multiple* UTF formats?

Because different systems need:

- Memory efficiency

- Backward compatibility
- Fast processing
- Network friendliness

So Unicode provides **multiple encoding formats**.

4️⃣ Main Unicode Transformation Formats

🔹 **UTF-8**

- Uses **1 to 4 bytes**
- ASCII characters → 1 byte
- Most widely used on the web 🌐

Example:

- `A` → 1 byte
- `अ` → 3 bytes
- 😀 → 4 bytes

✅ Advantages:

- Backward compatible with ASCII
- Saves space for English text

🔹 **UTF-16**

- Uses **2 or 4 bytes**
- Java's internal `char` and `String` representation

Example:

- `A` → 2 bytes
- 😀 → 4 bytes (surrogate pair)

✅ Advantages:

- Faster processing in Java
- Good balance between size and speed

🔹 **UTF-32**

- Uses **fixed 4 bytes**
- Simple but memory heavy

Example:

- `A` → 4 bytes
- 😀 → 4 bytes

❌ Rarely used due to memory waste.

5️⃣ Simple comparison table

UTF Format	Bytes Used	Used Where
UTF-8	1–4 bytes	Web, files, APIs
UTF-16	2 or 4 bytes	Java internal
UTF-32	Always 4 bytes	Special systems

6️⃣ Real-world analogy 🧠

Unicode = **Universal ID numbers for people**
UTF-8 / UTF-16 / UTF-32 = **Different languages to write those IDs on paper**

Same person → written differently → same identity.

7️⃣ One-line exam definition ✅

> **Unicode Transformation Formats are encoding schemes that convert Unicode code points into byte sequences for storage and transmission.**

8️⃣ Why this is important in Java?

- Java uses **UTF-16 internally**
- Files, APIs, DBs often use **UTF-8**
- Encoding mismatch causes **garbled text (�)**

9️⃣ 10-second interview answer 🎯

“Unicode Transformation Formats define how Unicode characters are encoded into bytes. UTF-8, UTF-16, and UTF-32 are different formats designed for efficiency, compatibility, and performance.”