### ****Java Data Types & Wrapper Classes****

### ****Java Data Types****

In Java, data types define the kind of data that can be stored in variables. Each data type determines the size, range, and operations that can be performed on the data. Java has two main categories of data types:

1. **Primitive Data Types**
2. **Reference (Non-Primitive) Data Types**

#### ****1. Primitive Data Types:****

Primitive data types are the most basic data types in Java. They are predefined by the language and are not objects. These data types are directly supported by the Java programming language.

| **Data Type** | **Size** | **Default Value** | **Range** |
| --- | --- | --- | --- |
| byte | 1 byte | 0 | -128 to 127 |
| short | 2 bytes | 0 | -32,768 to 32,767 |
| int | 4 bytes | 0 | --2,147,483,648 to 2,147,483,647 |
| long | 8 bytes | 0L | -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 |
| float | 4 bytes | 0.0f | 6-7 decimal digits precision |
| double | 8 bytes | 0.0d | 15-16 decimal digits precision |
| char | 2 bytes | '\u0000' (null) | 0 to 65,535 (Unicode characters) |
| boolean | 1 bit | false | true or false |

#### ****2. Reference Data Types:****

Reference data types are used to store the memory address (or reference) of objects and arrays. These types can hold complex data and are not directly stored in memory like primitive types. Reference types include classes, interfaces, arrays, and enums.

### ****Example of Primitive Data Types:****

public class DataTypeExample {

public static void main(String[] args) {

// Primitive Data Types

byte b = 100;

short s = 10000;

int i = 100000;

long l = 1000000000L;

float f = 10.5f;

double d = 20.99;

char c = 'A';

boolean isJavaFun = true;

System.out.println("Byte Value: " + b);

System.out.println("Short Value: " + s);

System.out.println("Int Value: " + i);

System.out.println("Long Value: " + l);

System.out.println("Float Value: " + f);

System.out.println("Double Value: " + d);

System.out.println("Char Value: " + c);

System.out.println("Boolean Value: " + isJavaFun);

}

}

### ****Example of Reference Data Types:****

public class ReferenceTypeExample {

public static void main(String[] args) {

// Reference Data Types

String name = "John Doe"; // String (Class)

int[] numbers = {1, 2, 3, 4}; // Array

Person person = new Person("Alice", 30); // Class (Person is a user-defined class)

System.out.println("Name: " + name);

System.out.println("Array: " + numbers[0] + ", " + numbers[1]);

System.out.println("Person: " + person.getName() + ", Age: " + person.getAge());

}

}

class Person {

private String name;

private int age;

// Constructor

public Person(String name, int age) {

this.name = name;

this.age = age;

}

// Getter methods

public String getName() {

return name;

}

public int getAge() {

return age;

}

}

### ****Wrapper Classes in Java:****

In Java, **wrapper classes** are part of the java.lang package and provide object representations for the primitive data types. They allow primitive types (like int, char, boolean, etc.) to be treated as objects, enabling them to be used in contexts that require objects, such as in collections (e.g., ArrayList, HashMap), generics, and other object-oriented structures.

Each primitive data type in Java has a corresponding wrapper class:

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

#### ****Features of Wrapper Classes:****

1. **Boxing and Unboxing:**
   * **Autoboxing** is the automatic conversion that the Java compiler makes between primitive types and their corresponding wrapper classes.
   * **Unboxing** is the reverse process, where a wrapper object is automatically converted to its corresponding primitive type.

Example:

Integer obj = 10; // Autoboxing: primitive int to Integer

int num = obj; // Unboxing: Integer to primitive int

1. **Utility Methods:** Wrapper classes come with utility methods for converting between different data types, parsing strings, and other operations:
   * parseInt(), parseDouble() for parsing strings into primitive values.
   * valueOf() to convert a primitive into a wrapper object.
   * toString() to convert a wrapper object to a string.

Example:

String str = "123";

int x = Integer.parseInt(str); // Parsing a string to an int

Double d = Double.valueOf("45.67"); // Converting String to Double

1. **Constants:** Many wrapper classes provide **constants** for the minimum and maximum values that the primitive type can hold:
   * Byte.MIN\_VALUE, Byte.MAX\_VALUE
   * Integer.MIN\_VALUE, Integer.MAX\_VALUE
   * Double.MIN\_VALUE, Double.MAX\_VALUE

Example:

System.out.println("Integer min: " + Integer.MIN\_VALUE);

System.out.println("Integer max: " + Integer.MAX\_VALUE);

1. **Nullability:** Unlike primitive types, wrapper classes can be assigned the value null, which is useful in cases where a variable might not be initialized or when working with databases that require object references.

Example:

Integer x = null; // Wrapper class can be null

1. **Immutability:** Wrapper classes are **immutable**, meaning once an object is created, its value cannot be changed.

#### ****Example of Wrapper Class Usage:****

public class WrapperClassExample {

public static void main(String[] args) {

// Autoboxing

Integer intObj = 100; // int primitive to Integer object

Double doubleObj = 45.67; // double primitive to Double object

// Unboxing

int num = intObj; // Integer object to int primitive

double value = doubleObj; // Double object to double primitive

// Using utility methods

String str = "123";

int parsedInt = Integer.parseInt(str); // Parsing string to int

// Printing

System.out.println("Parsed integer: " + parsedInt);

System.out.println("Integer object: " + intObj);

System.out.println("Double object: " + doubleObj);

// Constants

System.out.println("Max value of Integer: " + Integer.MAX\_VALUE);

System.out.println("Min value of Integer: " + Integer.MIN\_VALUE);

}

}

#### ****Advantages of Using Wrapper Classes:****

1. **Compatibility with Collections:** Since Java collections (like ArrayList, HashMap) work only with objects, wrapper classes allow you to store primitive values as objects in these collections.

Example:

ArrayList<Integer> list = new ArrayList<>();

list.add(10); // Autoboxing: primitive int to Integer

1. **Utility Methods:** Wrapper classes provide several useful methods for converting between types, comparing values, and manipulating data, such as compareTo(), equals(), and toString().
2. **Nullability:** Wrapper classes allow null to represent an uninitialized state or missing value, unlike primitive types that cannot be assigned null.

#### ****Disadvantages of Using Wrapper Classes:****

1. **Performance Overhead:** Since wrapper classes involve creating objects, they can introduce additional memory and performance overhead compared to using primitive types.
2. **Unnecessary Object Creation:** When only simple values are needed, using wrapper classes might not be necessary and could lead to excessive object creation, which can affect performance.

### ****Read Input in Java****

In Java, there are several ways to read input from the user. The most common methods include using the Scanner class, BufferedReader class, Console class, and the DataInputStream class. Each has its own use cases and characteristics.

Here’s a detailed overview of the different ways to read input in Java:

### ****1. Using the**** Scanner ****Class****

The Scanner class is the most commonly used method to read input in Java. It provides methods to read various types of data like strings, integers, floats, and more.

#### ****Steps to use**** Scanner ****class:****

* Import the Scanner class from java.util package.
* Create an instance of the Scanner class.
* Use appropriate methods like nextInt(), nextLine(), nextDouble(), etc., to read different types of input.

#### ****Example:****

import java.util.Scanner;

public class ScannerExample {

public static void main(String[] args) {

// Create a Scanner object

Scanner scanner = new Scanner(System.in);

// Read a string

System.out.print("Enter your name: ");

String name = scanner.nextLine();

// Read an integer

System.out.print("Enter your age: ");

int age = scanner.nextInt();

// Read a float

System.out.print("Enter your height (in meters): ");

float height = scanner.nextFloat();

// Print the values entered

System.out.println("Hello " + name + ", you are " + age + " years old and " + height + " meters tall.");

// Close the scanner object to prevent resource leak

scanner.close();

}

}

#### ****Methods available in**** Scanner****:****

* nextLine(): Reads a full line of input (including spaces).
* nextInt(): Reads an integer.
* nextDouble(): Reads a double.
* nextFloat(): Reads a float.
* nextBoolean(): Reads a boolean value.

### ****2. Using**** BufferedReader ****and**** InputStreamReader

BufferedReader is another way to read input, and it's often used when reading large chunks of text (e.g., reading a file or input in a loop). It's more efficient than Scanner for reading larger inputs because it buffers input data.

#### ****Steps to use**** BufferedReader****:****

* Import the BufferedReader and InputStreamReader classes.
* Create an instance of BufferedReader.
* Use the readLine() method to read a line of input.

#### ****Example:****

import java.io.BufferedReader;

import java.io.InputStreamReader;

import java.io.IOException;

public class BufferedReaderExample {

public static void main(String[] args) throws IOException {

// Create a BufferedReader object

BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));

// Read a string

System.out.print("Enter your name: ");

String name = reader.readLine();

// Read an integer

System.out.print("Enter your age: ");

int age = Integer.parseInt(reader.readLine());

// Read a float

System.out.print("Enter your height (in meters): ");

float height = Float.parseFloat(reader.readLine());

// Print the values entered

System.out.println("Hello " + name + ", you are " + age + " years old and " + height + " meters tall.");

// Close the reader object

reader.close();

}

}

#### ****Advantages of**** BufferedReader****:****

* More efficient for reading large inputs because it buffers the input data.
* Can be used to read input from files as well.
* Requires manual parsing of data (e.g., converting strings to integers).

#### ****Methods available in**** BufferedReader****:****

* readLine(): Reads an entire line of text.

### ****3. Using**** Console ****Class****

The Console class provides a way to read input from the user, but it only works in console-based environments and might not work in some IDEs (e.g., Eclipse, IntelliJ). It's primarily used in command-line applications.

#### ****Steps to use**** Console****:****

* Create a Console object by calling System.console().
* Use readLine() to read strings, and readPassword() to read passwords.

#### ****Example:****

public class ConsoleExample {

public static void main(String[] args) {

// Get the console object

Console console = System.console();

if (console == null) {

System.out.println("No console available");

return;

}

// Read a string

String name = console.readLine("Enter your name: ");

// Read a password (hidden input)

char[] passwordArray = console.readPassword("Enter your password: ");

// Print the values entered

System.out.println("Hello " + name + ", you entered a password with length: " + passwordArray.length);

}

}

#### ****Methods available in**** Console****:****

* readLine(): Reads a line of text.
* readPassword(): Reads a password (input is hidden).

#### ****Note:****

The Console class might not work in certain IDEs because it requires a terminal/console environment.

### ****4. Using**** DataInputStream

DataInputStream allows you to read primitive data types directly (like int, float, long, etc.) from the input stream. It's typically used for reading data from files or binary streams but can also be used for reading input from the console.

#### ****Steps to use**** DataInputStream****:****

* Create an instance of DataInputStream.
* Use methods like readInt(), readDouble(), readByte(), etc., to read different primitive types.

#### ****Example:****

import java.io.DataInputStream;

import java.io.IOException;

public class DataInputStreamExample {

public static void main(String[] args) throws IOException {

// Create a DataInputStream object

DataInputStream dataInputStream = new DataInputStream(System.in);

// Read an integer

System.out.print("Enter your age: ");

int age = dataInputStream.readInt();

// Read a double

System.out.print("Enter your height (in meters): ");

double height = dataInputStream.readDouble();

// Print the values entered

System.out.println("You are " + age + " years old and " + height + " meters tall.");

// Close the DataInputStream

dataInputStream.close();

}

}

#### ****Methods available in**** DataInputStream****:****

* readInt(): Reads an integer.
* readDouble(): Reads a double.
* readByte(): Reads a byte.
* readBoolean(): Reads a boolean.