Why Java is not Fully Object oriented language?

**Difference b/w Primitive data types and non primitive Data types?**

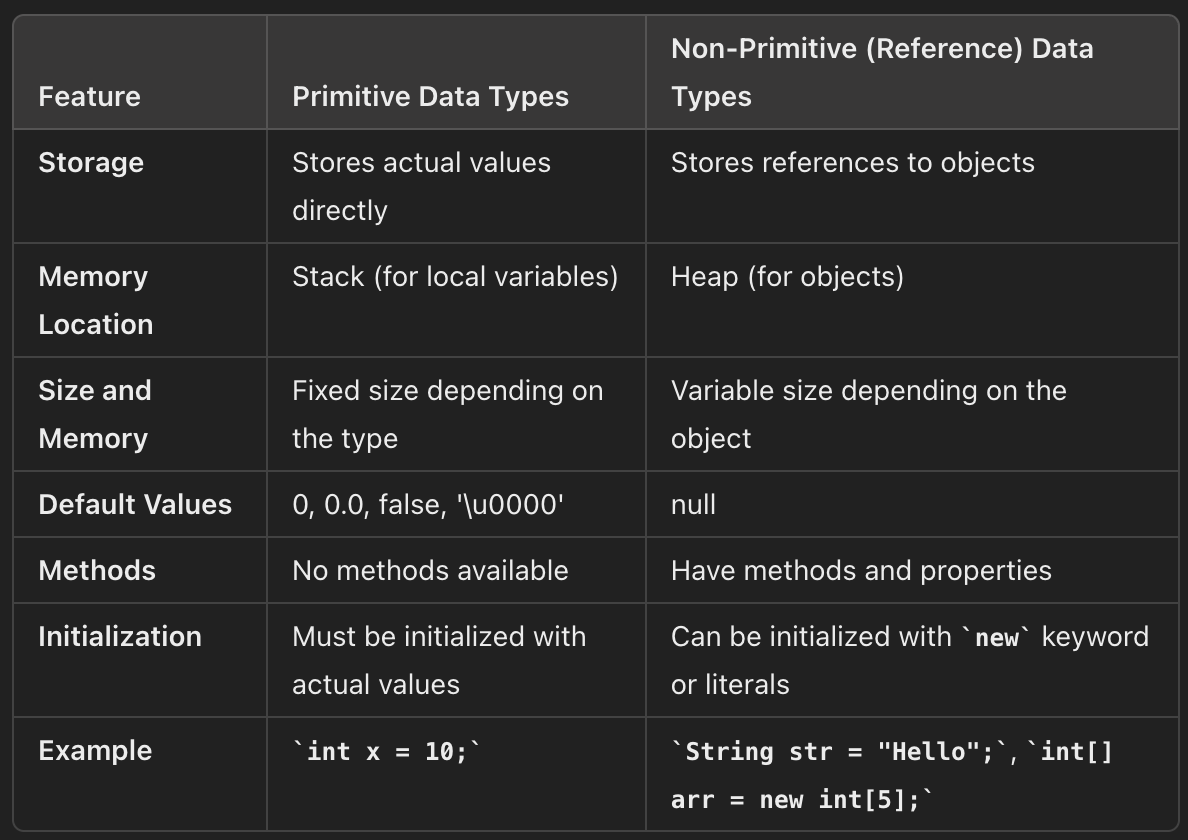
#### Primitive Data Types:

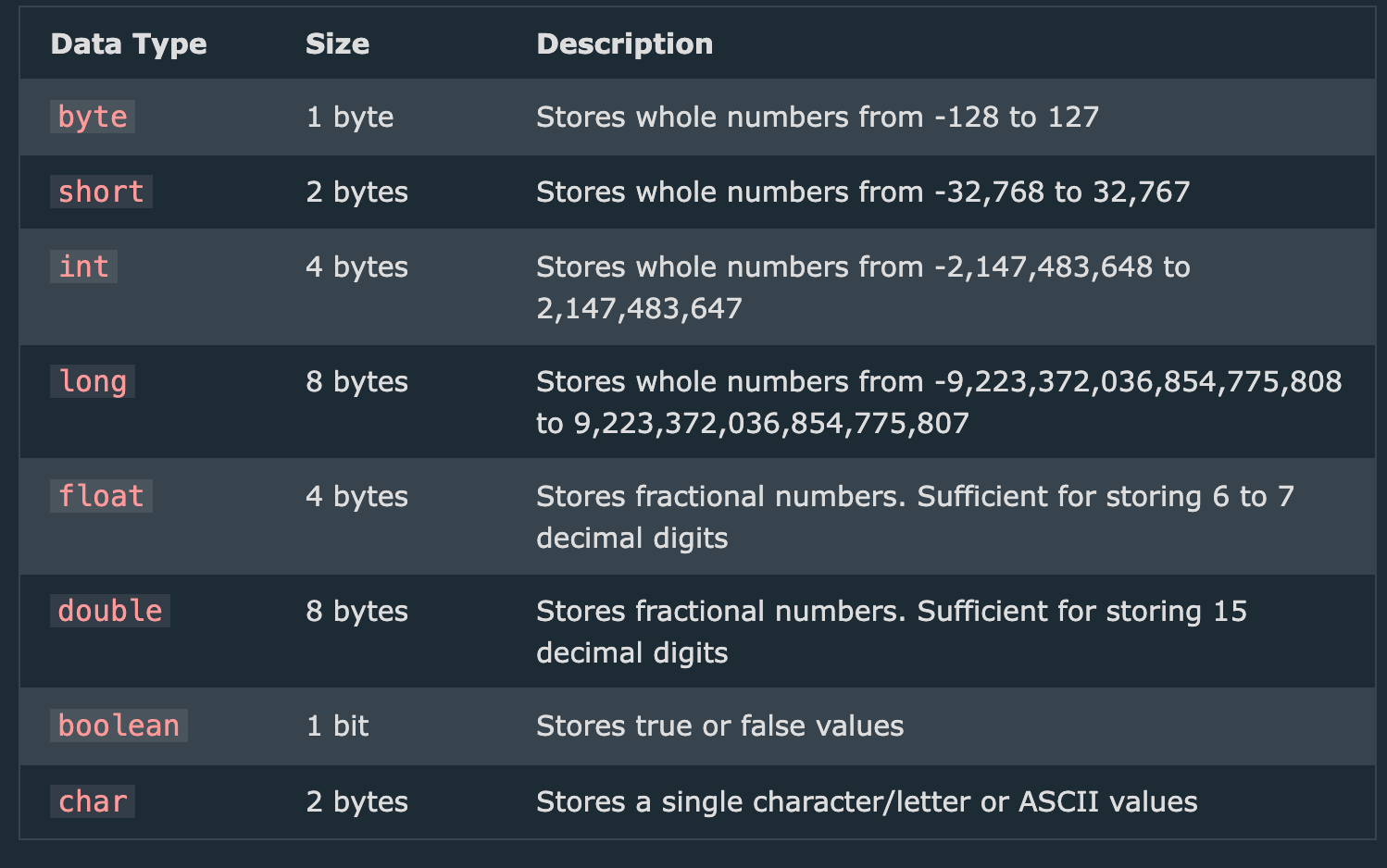
1. **Storage of Values**: Primitive data types store actual values directly in memory.
2. **Representation**: Each primitive data type has a fixed size and directly represents the actual data (e.g., an int variable stores the actual integer value).
3. **Examples**: byte, short, int, long, float, double, char, boolean.

#### **Non-Primitive (Reference) Data Types:**

1. **Storage of References**: Non-primitive data types store references (or addresses) to objects or arrays in memory. The actual data is stored elsewhere in the heap.
2. **Representation**: They represent objects or collections of data, which can vary in size and structure (e.g., a String variable stores a reference to a string object in memory).
3. **Memory Overhead**: Non-primitive types are generally less memory-efficient due to the overhead of storing references and additional object metadata.
4. **Default Values**: The default value for non-primitive types is null, indicating that the variable does not reference any object.
5. **Methods and Properties**: Non-primitive types can have methods and properties. For example, a String object has methods like length(), substring(), etc.

**Examples**: String, arrays (e.g., int[], String[]), custom objects (e.g., instances of classes), collections (e.g., ArrayList, HashMap).





Difference b/w identifiers and Variables?

· **Identifiers**:   
Used for naming any element in the program, such as classes, methods, variables, interfaces, and packages.

· Serve as unique names for various program components to ensure they can be referenced correctly.

·

· **Variables**:

Specifically used to name storage locations that hold data values.

Serve as containers for data that can be manipulated through operations and assignments.

What is Java Type Casting ?

Type casting in Java is the process of converting a variable from one data type to another. This can be necessary when you want to perform operations that involve different types, or when you need to store a value of one type in a variable of another type. There are two main types of type casting in Java:

There are two main types of type casting in Java:

1. **Implicit (Automatic) Casting**: Also known as widening conversion, where the conversion happens automatically because the source type is smaller or less precise than the target type.
2. onverting a smaller type to a larger type size  
   byte -> short -> char -> int -> long -> float -> double
3. **Explicit Casting**: Also known as narrowing conversion, where you must explicitly specify the conversion because the source type is larger or more precise than the target type.  
     
    converting a larger type to a smaller size type  
   double -> float -> long -> int -> char -> short -> byte

### Implicit (Automatic) Casting

Implicit casting occurs when the target type is larger or more precise than the source type, and Java automatically converts the data without any explicit instruction. This usually happens with primitive data types.

#### Example:

java

Copy code

public class ImplicitCastingExample {

public static void main(String[] args) {

int myInt = 100;

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

System.out.println("Integer value: " + myInt); // Outputs 100

System.out.println("Double value: " + myDouble); // Outputs 100.0

}

}

### Explicit Casting

Explicit casting is required when the target type is smaller or less precise than the source type. This is done by specifying the target type in parentheses before the variable.

#### Example:

java

Copy code

public class ExplicitCastingExample {

public static void main(String[] args) {

double myDouble = 9.78;

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

System.out.println("Double value: " + myDouble); // Outputs 9.78

System.out.println("Integer value: " + myInt); // Outputs 9

}

}

### Type Casting with Non-Primitive Data Types

Casting can also be performed with non-primitive data types like objects, typically involving inheritance and interfaces.

### **Upcasting**

### Upcasting is the process of converting a subclass type to a superclass type. It is always safe and is done implicitly by Java

### Downcasting

Downcasting is the process of converting a superclass type to a subclass type.

ex:-

class Animal {

public void makeSound() {

System.out.println("Animal makes a sound");

}

}

class Dog extends Animal {

@Override

public void makeSound() {

System.out.println("Dog barks");

}

public void fetch() {

System.out.println("Dog fetches");

}

}

public class CastingExample {

public static void main(String[] args) {

// Upcasting: Dog object is referenced by an Animal variable

Animal myAnimal = new Dog();

myAnimal.makeSound(); **// Outputs "Dog barks" due to polymorphism**

// Downcasting: Casting the Animal reference back to Dog

Dog myDog = (Dog) myAnimal;

myDog.makeSound(); **// Outputs "Dog barks"**

myDog.fetch();  **// Outputs "Dog fetches"**

// Incorrect downcasting example

Animal anotherAnimal = new Animal();

try {

Dog anotherDog = (Dog) anotherAnimal; // This will cause ClassCastException

anotherDog.makeSound();

} catch (ClassCastException e) {

System.out.println("Cannot cast Animal to Dog: " + e.getMessage());

}

}

}

**FOR EACH LOOP :-**

There is also a "for-each" loop, which is used exclusively to loop through elements in an [array](https://www.w3schools.com/java/java_arrays.asp):

**Syntax :-**

for (type variableName : arrayName) {

// code block to be executed

}

The following example outputs all elements in the cars array, using a "for-each" loop:

**Example**

String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};

for (String i : cars) {

System.out.println(i);

}

o/p:- Volvo, Ford, Mazda