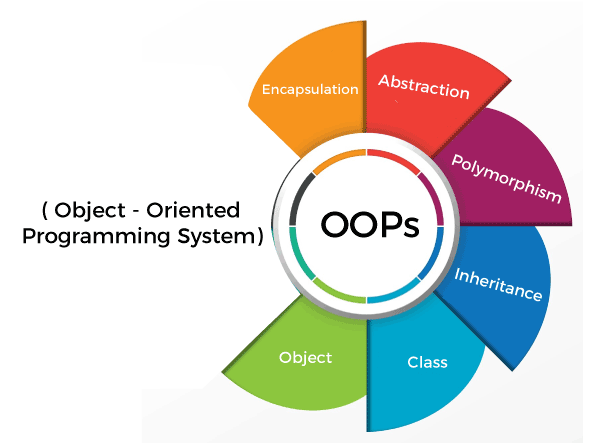
OOP

**OOP**

OOP stands for object oriented programming language,the main purpose of OOP is to deal with real world entity using programming language. A programming model that organizes software design around objects, rather than functions and logic.



**OOPS Features:**

**01.** Object

**02.** Class

**03.** inheritance

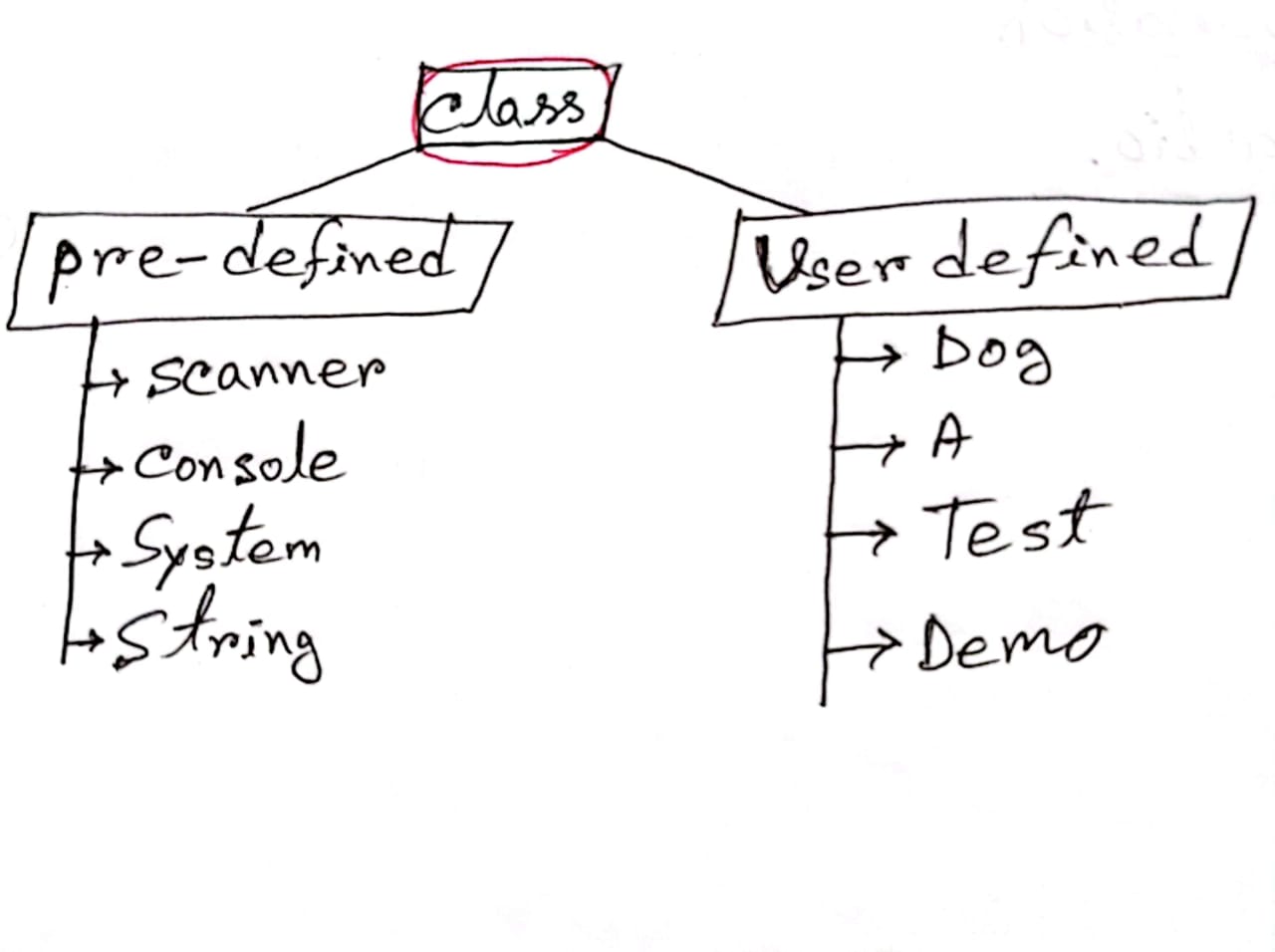
**04.** Polymorphism

**05.** EnCapsulation

**06.** Abstraction

**Class**

Class is a collection of Data and methods,it doesn’t take any space on memory. Class is tem-plate that defines the methods and variables of a specific type of object. Class are user-defined data types that act as blueprints for objects. a class is a blueprint for creating objects.



**User-defined class:**

A class whivh is created by ~~java~~ programmer is called user-defined class.

Ex:-

**Object**

In object-oriented programming (OOP), objects are the fundamental (building blocks) unit of computer programs. Object is an instance veriable of class thar executes the class.

**Method**

In Object-Oriented Programming (OOP), a method is a function that is defined within a class and describes a behavior or action that objects (or instances) of the class can perform.

**Constructor**

Constructor is a special type of method whose name is same as class name.

**Key Features of Constructors**

1. Same Name as the Class: A constructor has the same name as the class it belongs to.

2. No Return Type: Constructors don’t have a return type, not even `void`.

3. Automatic Invocation: A constructor is called automatically when an object is created using the new keyword.

4. A constructor is automatically called the object.

**Types of Constructors:**

**Default Constructor:** A constructor which does not have any parameter is called default constructor.

**1. Default Constructor**

If you don't define any constructor, Java provides a default, no-argument constructor that initializes objects with default values (like `null` for objects, `0` for numeric types, and `false` for `boolean`).

**Example:**

public class MyClass {

int value;

// No explicit constructor means Java provides a default constructor.

}

MyClass obj = new MyClass(); // Calls the default constructor

**02. Parameterized Constructor:** A constructor through which we can pass one or more parameters is called parametrized constructor.

2. Parameterized Constructor

A parameterized constructor allows you to pass values at the time of object creation, giving you control over how each object is initialized.

Example:

public class MyClass {

int value;

// Parameterized constructor

public MyClass(int value) {

this.value = value;

}

}

MyClass obj = new MyClass(10); // Calls the parameterized constructor with a value

3. Copy Constructor (Manual)

Java doesn’t have a built-in copy constructor, but you can manually define one to create a new object by copying an existing one.

Example:

public class MyClass {

int value;

// Parameterized constructor

public MyClass(int value) {

this.value = value;

}

// Copy constructor

public MyClass(MyClass other) {

this.value = other.value;

}

}

MyClass obj1 = new MyClass(10);

MyClass obj2 = new MyClass(obj1); // Creates a copy of obj1

**Special Features of Constructors in OOP**

**Constructor Overloading:** Like regular methods, constructors can be overloaded, meaning you can have multiple constructors in the same class with different parameter lists.

**Constructor Chaining:** Constructors can call other constructors within the same class (using `this()`) or from the superclass (using `super()`).

**Inheritance and Constructors**: A subclass does not inherit the constructor of its superclass, but it can invoke the superclass constructor using `super()`.

Example: Constructor in OOP

public class Car {

String model;

int year;

// Default constructor

public Car() {

this.model = "Unknown";

this.year = 0;

}

// Parameterized constructor

public Car(String model, int year) {

this.model = model;

this.year = year;

}

// Copy constructor

public Car(Car otherCar) {

this.model = otherCar.model;

this.year = otherCar.year;

}

}

// Creating objects

Car car1 = new Car(); // Calls the default constructor

Car car2 = new Car("Toyota", 2020); // Calls the parameterized constructor

Car car3 = new Car(car2); // Calls the copy constructor

**Summary**

- Constructors are vital for object initialization in OOP.

- They help in setting up initial values and ensuring that objects start in a valid state.

- Constructors can be overloaded to offer flexible initialization options.

**Static**

In Java, the static keyword is widely used in Object-Oriented Programming (OOP) for defining variables, methods, and even nested classes that belong to a class rather than to instances of the class.

**1. Static Variables (Class Variables)**

A static variable is shared among all instances of a class. Instead of being part of the individual objects, it is part of the class itself.

Syntax:

public class MyClass {

static int sharedVariable = 10;

}

Usage: All instances of MyClass will share sharedVariable. Any change to sharedVariable by one instance affects all other instances.

**2. Static Methods (Class Methods)**

A static method can be called on the class itself, without creating an instance. Static methods typically operate on static variables or perform utility tasks that do not depend on instance variables.

Syntax:

public class MyClass {

static void display() {

System.out.println("Hello from static method!");

}

}

Usage: You can call MyClass.display() directly, without needing an object of MyClass.

**3. Static Blocks**

A static block is a block of code that runs only once when the class is loaded, often used for initializing static variables.

Syntax:

public class MyClass {

static int staticVariable;

static {

staticVariable = 5; // This runs once when the class is loaded

}

}

**4.** Static Nested Classes

A static nested class is a class defined within another class but is not associated with an instance of the outer class. You can create objects of a static nested class without creating an instance of the outer class.

Syntax:

public class OuterClass {

static class NestedClass {

void display() {

System.out.println("Inside static nested class");

}

}

}

Usage: You can create an instance of NestedClass without an OuterClass instance:

**Instance Variable:**

This is a term often used in programming. In object-oriented programming, an instance variable is a variable defined within a class that is unique to each instance (or object) of that class.

**Global or Local Variable:**

If you mean a variable that is accessible throughout a program or within a specific function instantly after it's declared.

In Object-Oriented Programming (OOP) in Java, variables are essential as they store data that objects can use. There are different types of variables used in Java OOP, each serving a unique purpose depending on the context and scope. Here’s a breakdown:

**Instance Variables (Non-Static Fields)**

Definition: These variables are defined inside a class but outside any method, constructor, or block. They represent the properties or attributes of an object.

Characteristics:

- Each instance of a class has its own copy of the instance variables.

- Their values are unique to each object.

Usage: Typically used to maintain the state of an object.

Example:

public class Person {

String name; // Instance variable

int age; // Instance variable

}

2. Class Variables (Static Fields)

Definition: Defined with the `static` keyword inside a class, but outside methods or constructors.

Characteristics:

- Shared among all instances of a class.

- Only one copy exists, regardless of how many objects are created.

Usage: Often used for constants or properties shared by all objects.

Example:

public class Employee {

static int companyCount = 0; // Class variable

}

```

3.Local Variables

Definition: Variables declared inside a method, constructor, or block.

Characteristics:

- Only accessible within the method, constructor, or block where they are declared.

- Not accessible outside that specific method or block.

Usage: Used for temporary storage and computation inside methods.

Example:

public void display() {

int temp = 10; // Local variable

System.out.println(temp);

}

```

4. Parameters

Definition: Variables passed to methods or constructors.

Characteristics:

- Used to pass values to methods.

- Local to the method, they are only accessible within the method.

Usage: Used for passing data when calling a method.

Example:

public void setName(String name) { // 'name' is a parameter

this.name = name;

}

These variable types allow Java programs to manage data effectively, giving each object a structure for holding information and a way to perform operations using that data.

In Java, both \*\*instance blocks\*\* and \*\*static blocks\*\* are used for initializing code, but they serve different purposes and have different behaviors. Here’s a breakdown of the differences:

**1.Instance Block**

- \*\*Definition\*\*: A block of code within a class that does not use any modifiers like `static`.

- \*\*Execution\*\*: Executes every time an instance of the class is created, right after the constructor is called.

- \*\*Usage\*\*: Used to initialize instance variables, especially when the initialization logic is complex or should happen every time an object is created.

- \*\*Syntax\*\*:

```java

{

// instance block code

}

```

- \*\*Order of Execution\*\*: Runs before the constructor but after the superclass constructor, if any.

\*\*Example\*\*:

```java

public class MyClass {

int x;

// Instance block

{

x = 10;

System.out.println("Instance block executed");

}

public MyClass() {

System.out.println("Constructor executed");

}

}

```

**2.Static Block**

- \*\*Definition\*\*: A block of code within a class that is marked with the `static` keyword.

- \*\*Execution\*\*: Executes only once when the class is loaded into memory, before any instances of the class are created and even before the main method runs if it’s in the same class.

- \*\*Usage\*\*: Typically used for static variable initialization or any setup code that should only run once for the entire class, not for each instance.

- \*\*Syntax\*\*:

```java

static {

// static block code

}

```

- \*\*Order of Execution\*\*: Runs when the class is loaded, before any instance blocks or constructors are called.

\*\*Example\*\*:

```java

public class MyClass {

static int y;

// Static block

static {

y = 20;

System.out.println("Static block executed");

}

public static void main(String[] args) {

System.out.println("Main method executed");

MyClass obj = new MyClass(); // Instance block will run now

}

}

```

### Key Differences

| Aspect | Instance Block | Static Block |

|-----------------------|------------------------------------------------------|----------------------------------------------------|

| \*\*Keyword\*\* | None | `static` |

| \*\*Execution\*\* | Each time an instance is created | Once, when the class is loaded |

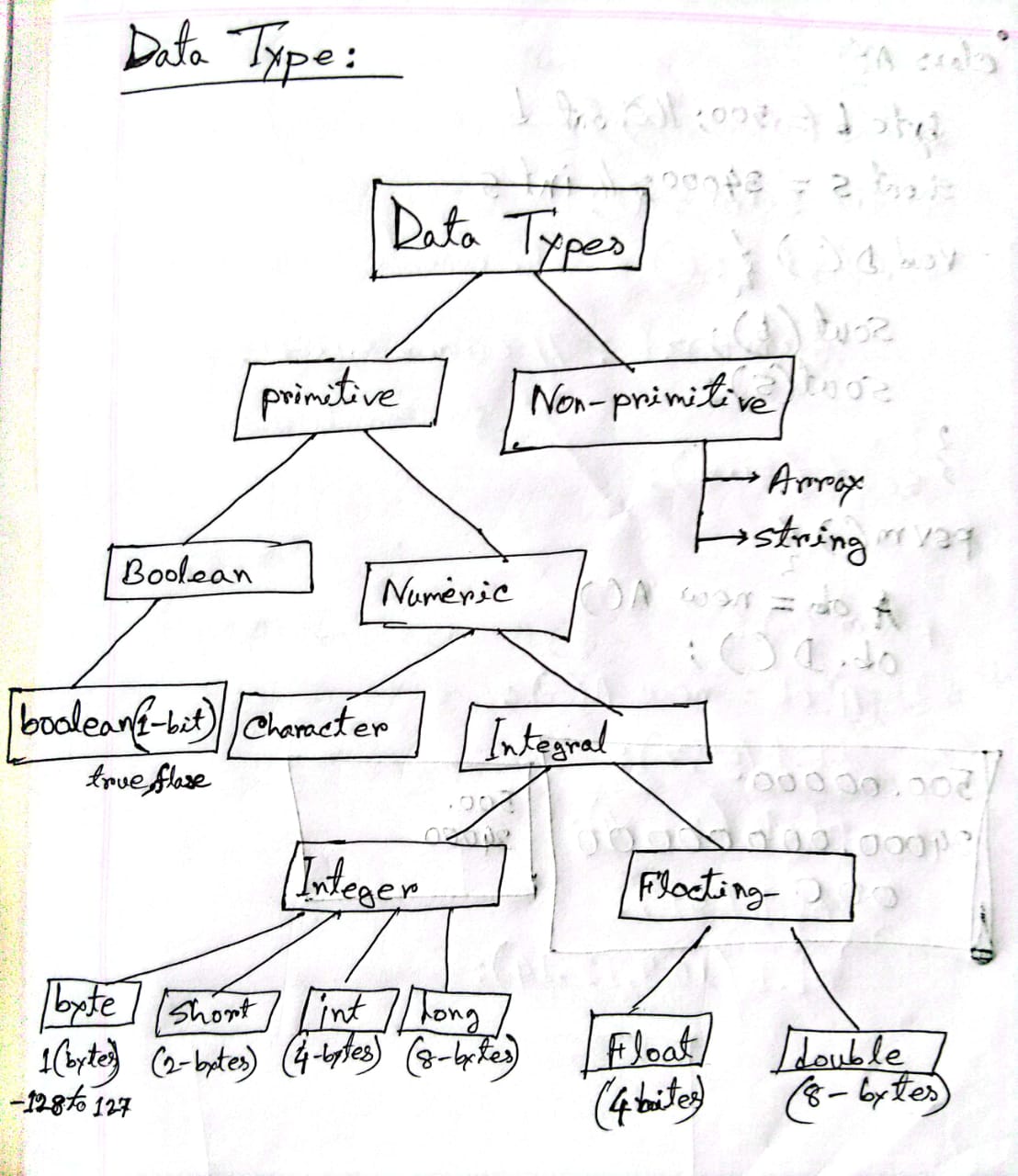
| \*\*Purpose\*\* | Initialize instance variables for each object | Initialize static variables or one-time setup |

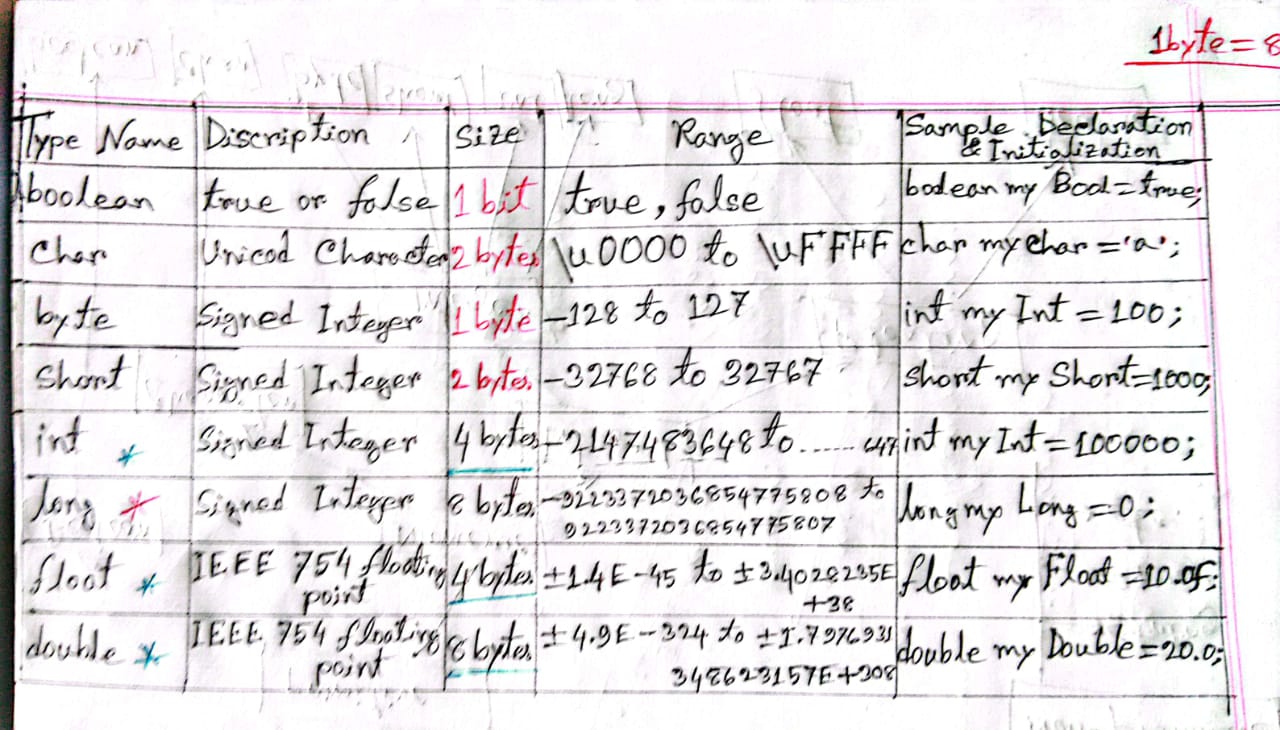
| \*\*Timing\*\* | Before constructor for each object | Before the main method and any instance creation |

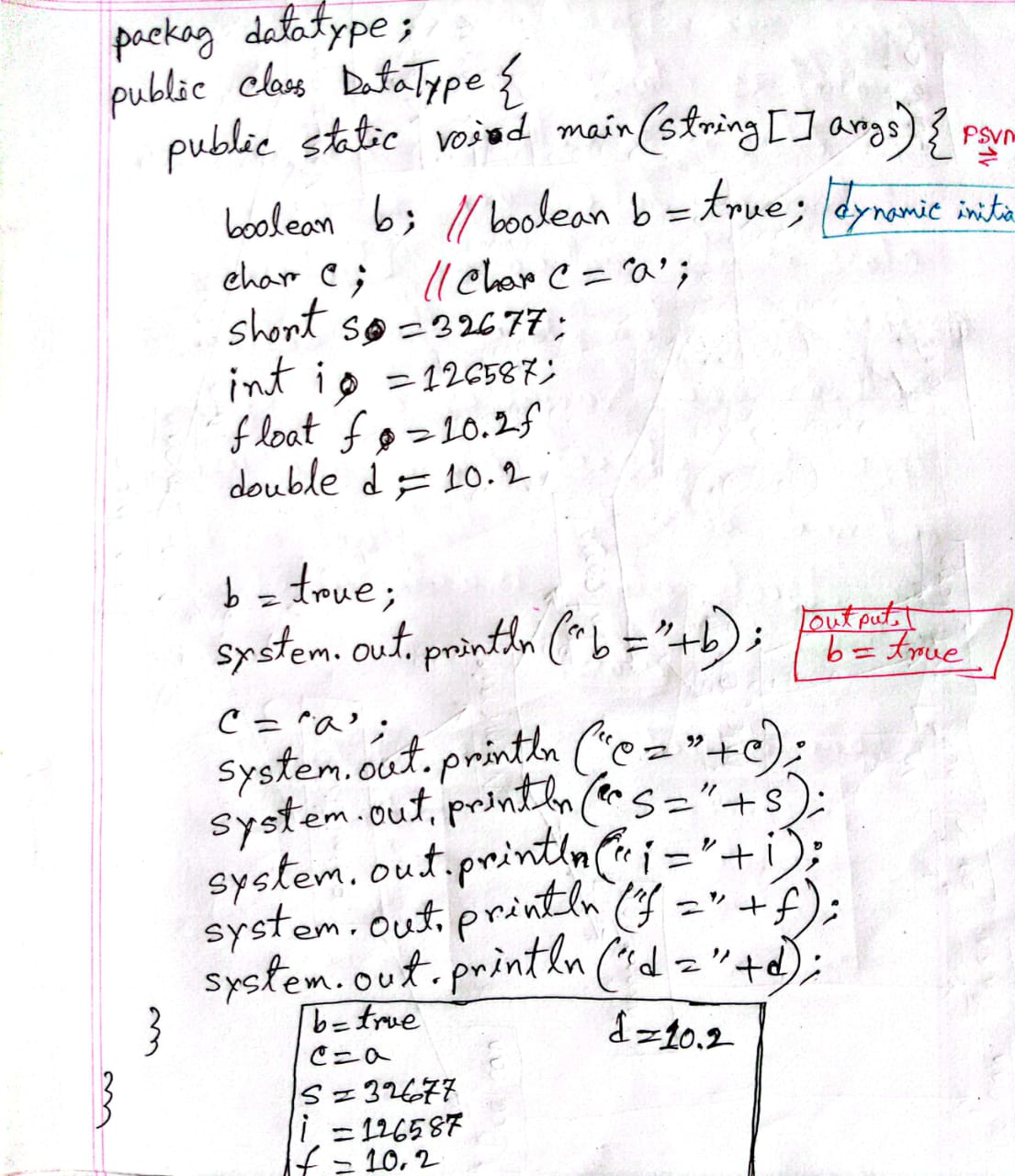
### Summary

- \*\*Instance blocks\*\* are useful for object-specific initialization, while \*\*static blocks\*\* are used for class-level initialization that should happen only once.

**Data Type**

****

****

****