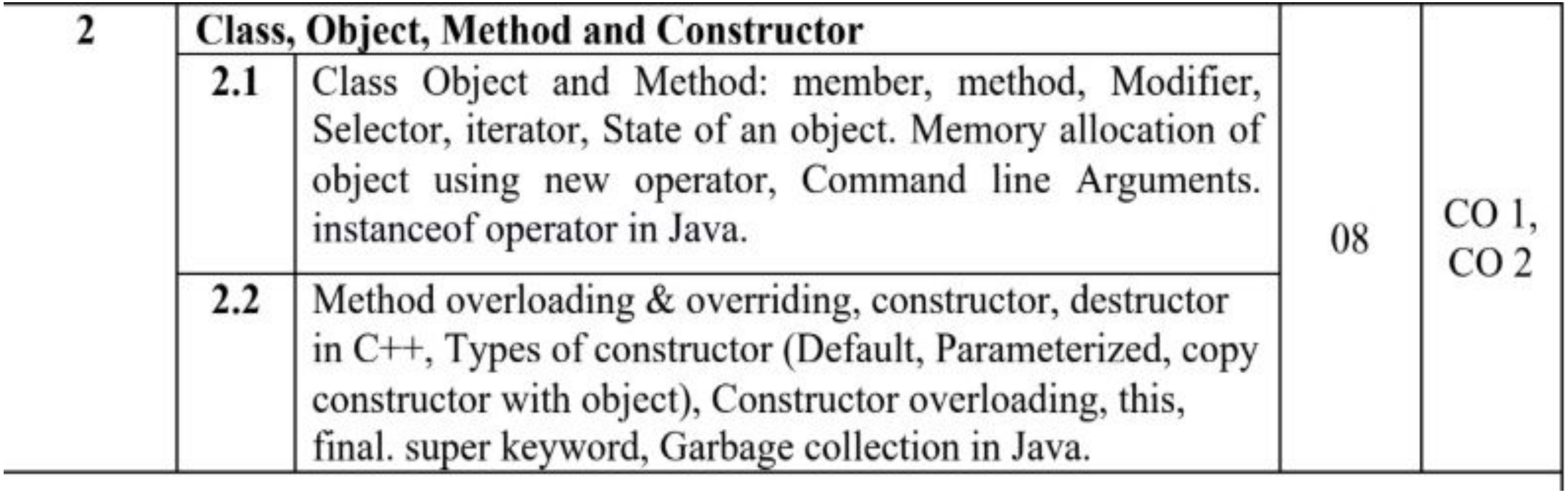
SOMAIYA
VIDYAVIHAR UNIVERSITY
K J Somaiya School of Engineering
(formerly K J Somaiya College of Engineering)

Somaiya
TRUST

Java Programming –Core Concepts

Dr. Shila Jawale
Asst.Professor, KJSSE



2	**Class, Object, Method and Constructor**		08	CO 1, CO 2
	2.1	Class Object and Method: member, method, Modifier, Selector, iterator, State of an object. Memory allocation of object using new operator, Command line Arguments. instanceof operator in Java.		
	2.2	Method overloading & overriding, constructor, destructor in C++, Types of constructor (Default, Parameterized, copy constructor with object), Constructor overloading, this, final. super keyword, Garbage collection in Java.		

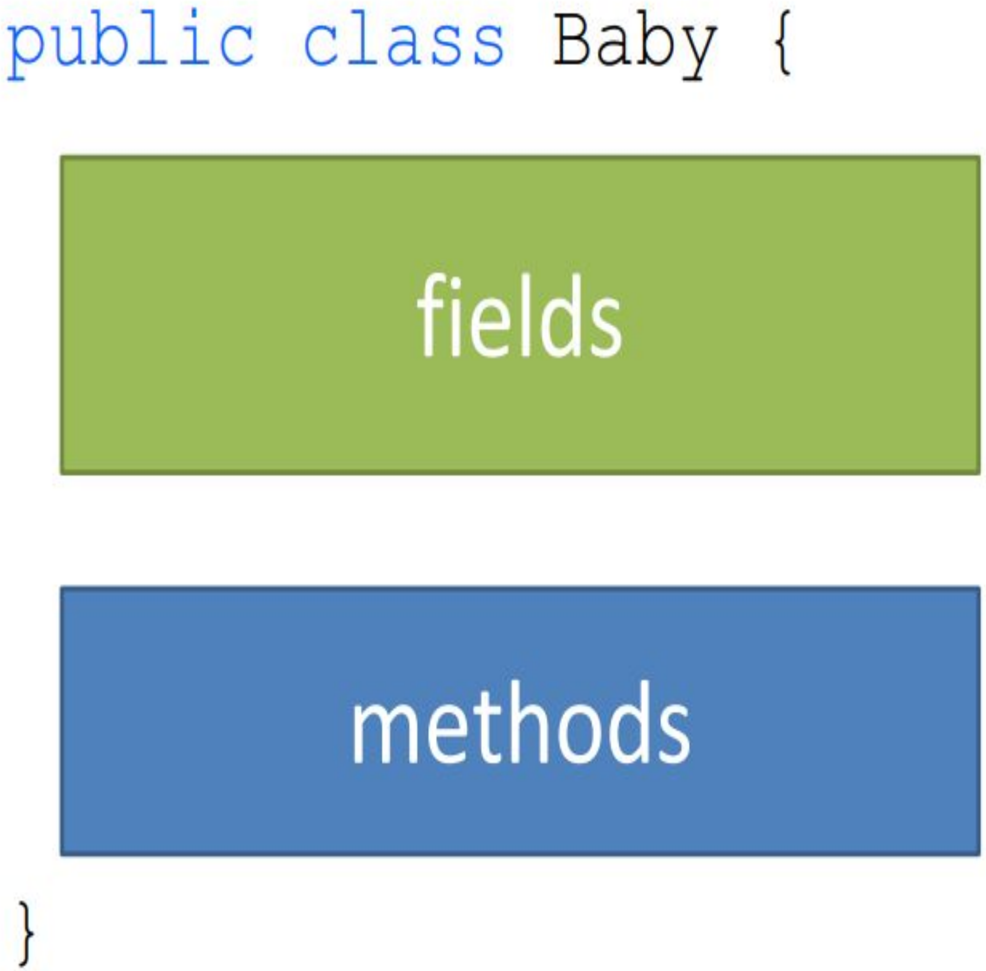
Introducing Classes

- Class Structure
- Class Methods
- Constructor
- Overloading Methods and constructor
- The this keyword
- Object as parameters and Returning objects
- Call by Values and Call by Reference
- Recursion
- Static field and static Methods
- Access specifier
- Nested classes
- Java is garbage collector

Java Essentials – Class & Object Basics

Topics:

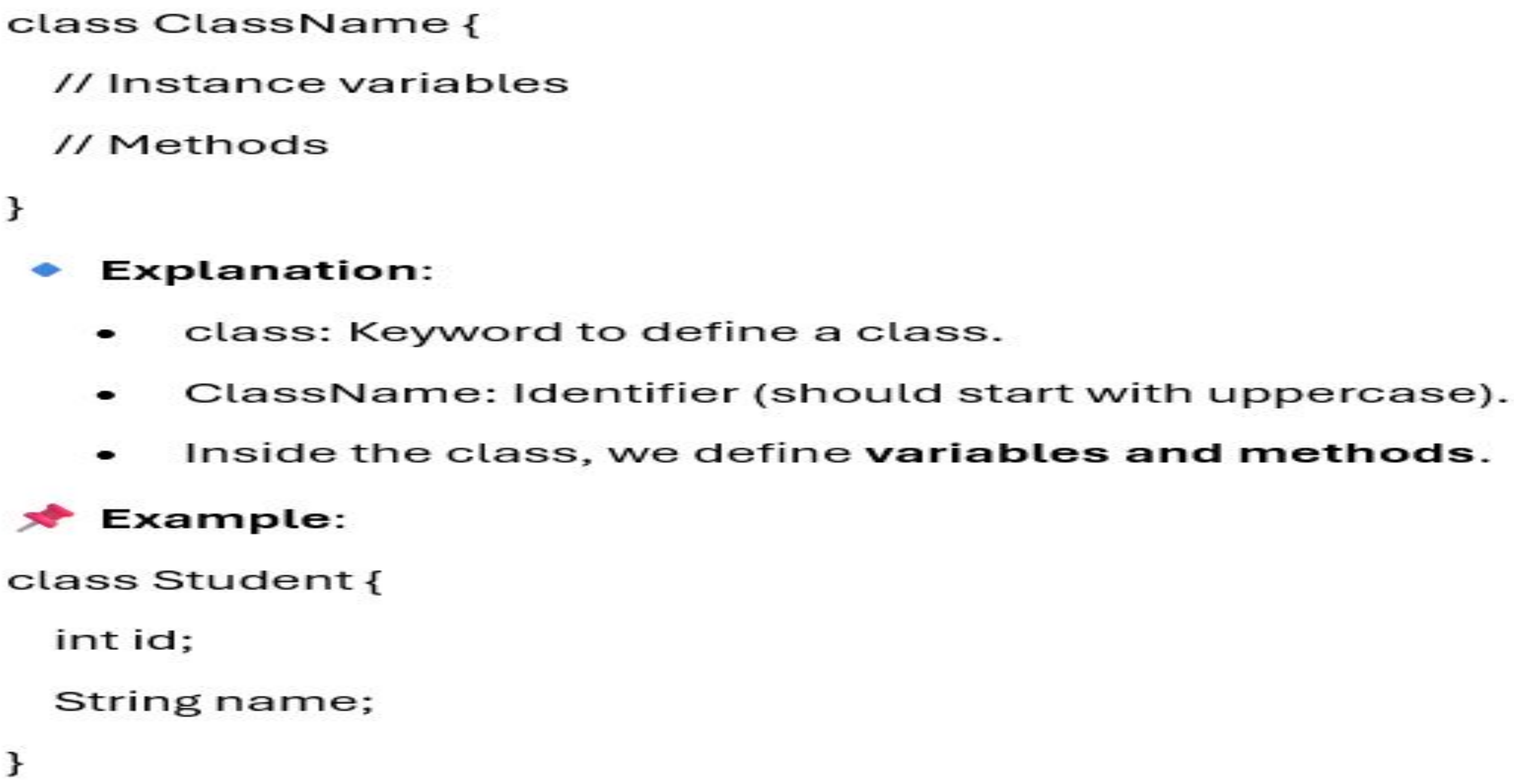
- General form of a class
- Declaration of instance variable
- Declaring an object
- Accessing instance variables
- Assigning object reference variables



```
public class Baby {

    fields

    methods

}
```

Note

- Class names are Capitalized
- 1 Class = 1 file
- Having a `main` method means the class can be run



```
class ClassName {
   // Instance variables
   // Methods
}
```

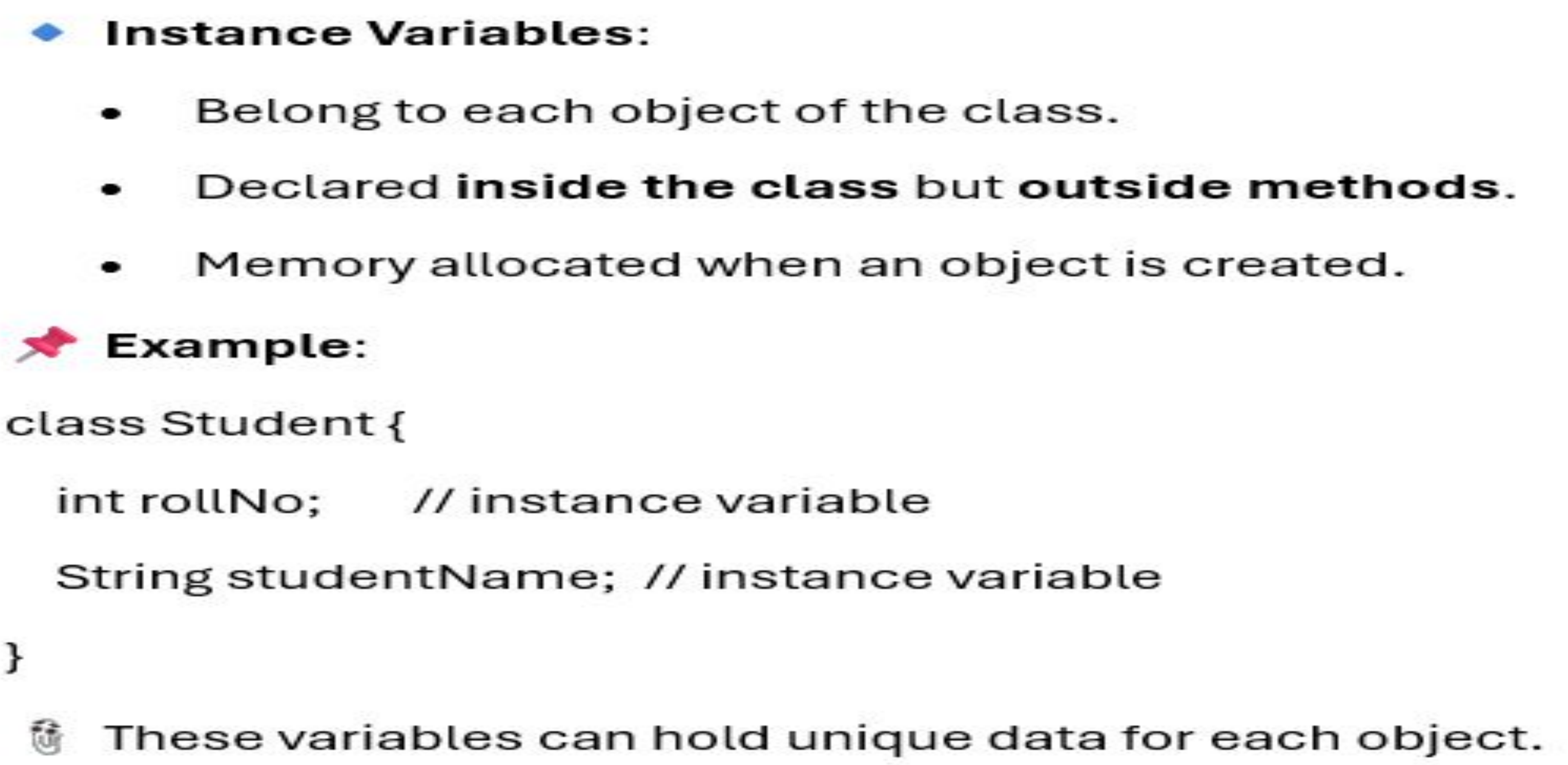
🔹 **Explanation:**

- class: Keyword to define a class.
- ClassName: Identifier (should start with uppercase).
- Inside the class, we define **variables and methods**.

📌 **Example:**

```
class Student {
   int id;
   String name;
}
```

Declaration of Instance Variables



🔹 **Instance Variables:**

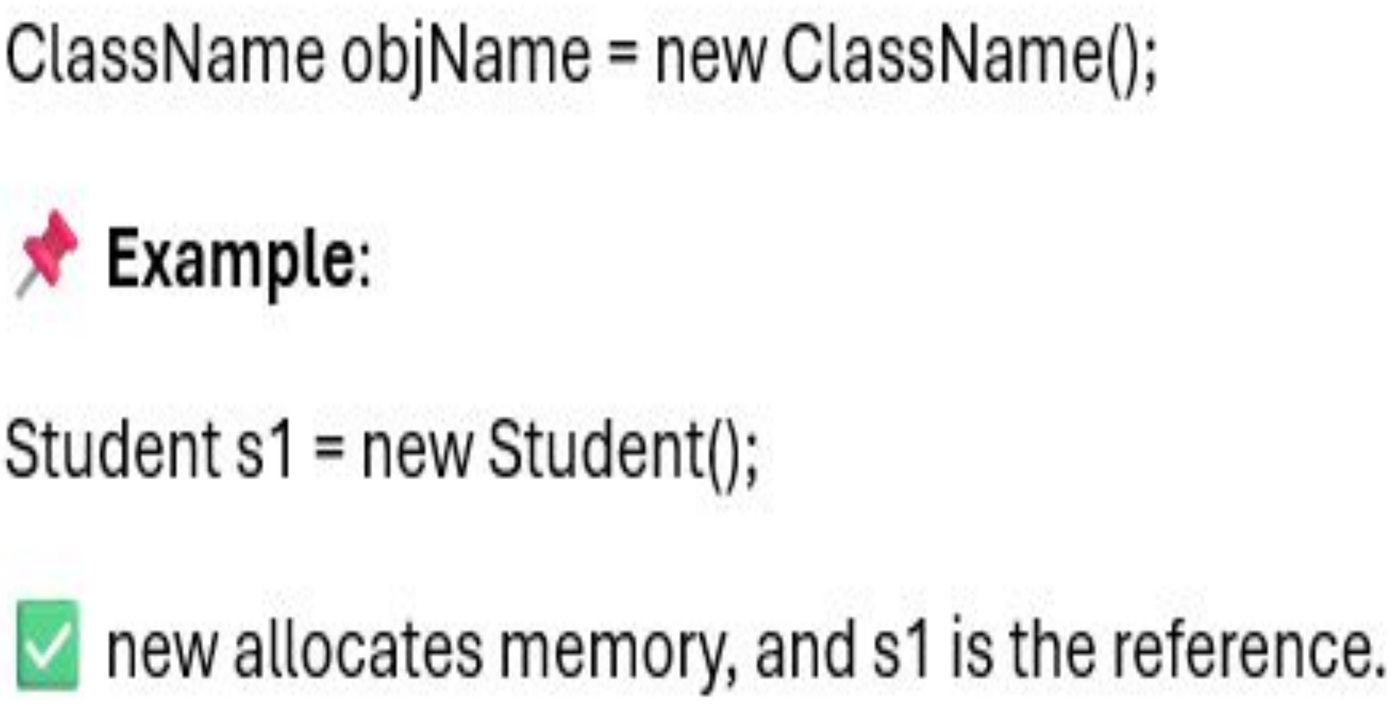
- Belong to each object of the class.
- Declared **inside the class** but **outside methods**.
- Memory allocated when an object is created.

📌 **Example:**

```
class Student {
    int rollNo;        // instance variable
    String studentName;  // instance variable
}
```

☺ These variables can hold unique data for each object.

Declaring an Object



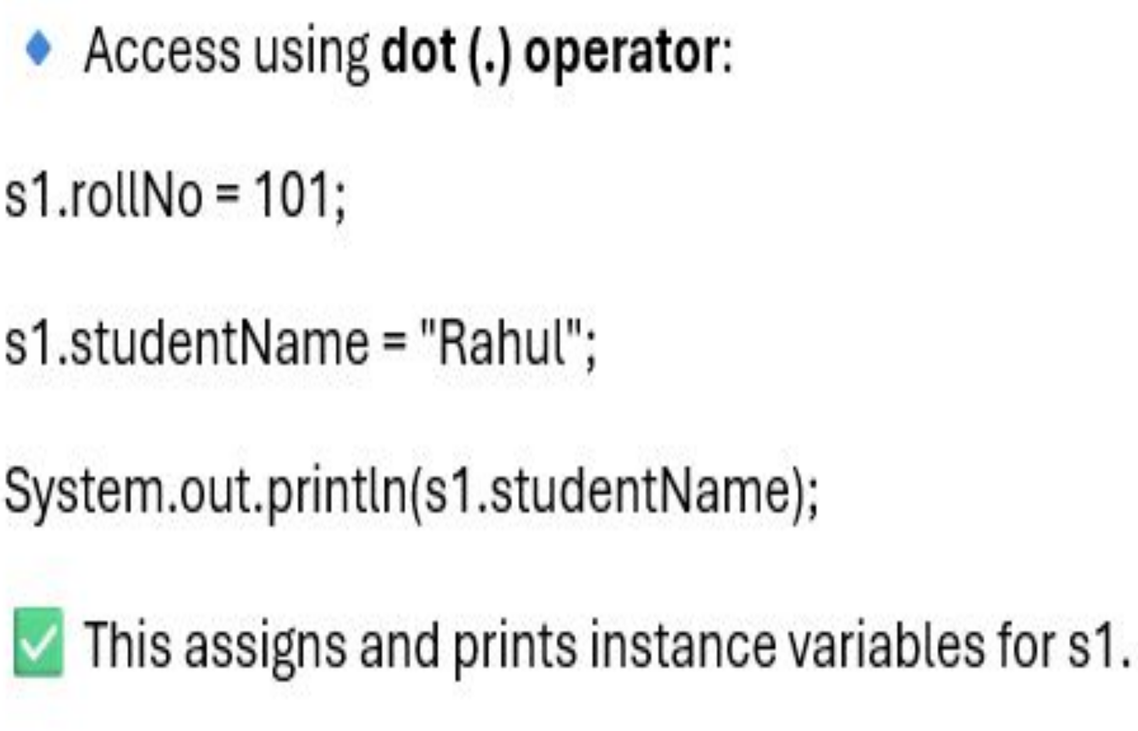
```
ClassName objName = new ClassName();
```

📌 **Example:**

```
Student s1 = new Student();
```

✅ new allocates memory, and s1 is the reference.

Accessing Instance Variables

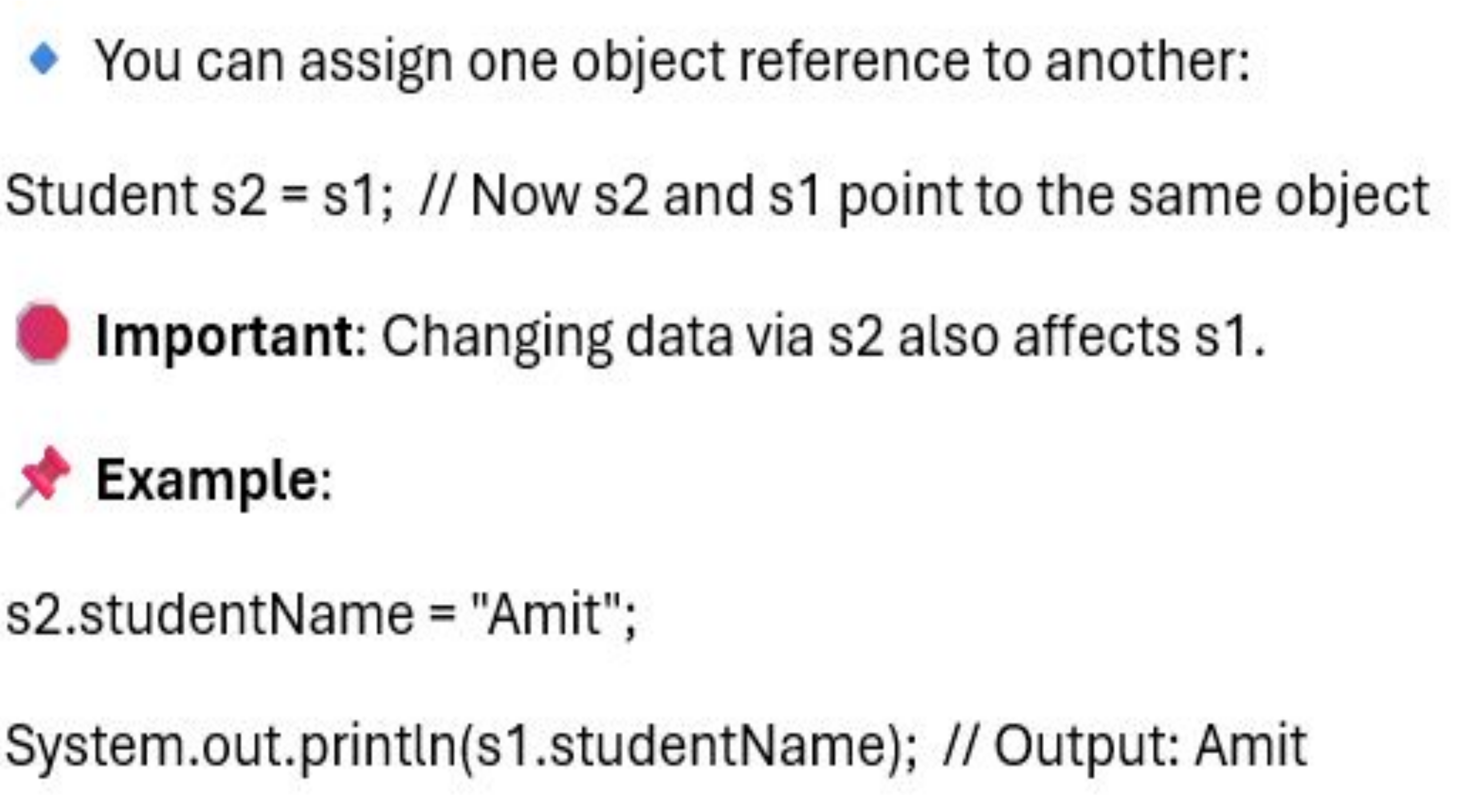


◆ Access using **dot (.) operator**:

```
s1.rollNo = 101;

s1.studentName = "Rahul";

System.out.println(s1.studentName);
```

✅ This assigns and prints instance variables for s1.

Assigning Object Reference Variables



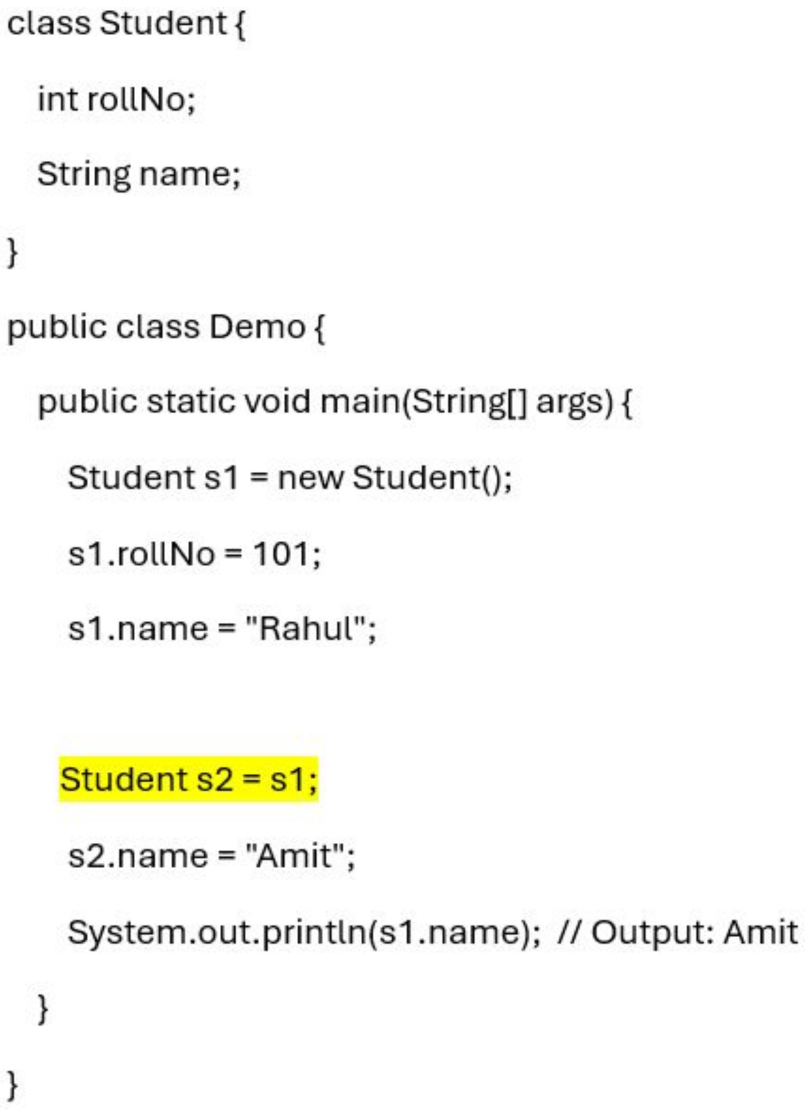
🔹 You can assign one object reference to another:

```
Student s2 = s1;  // Now s2 and s1 point to the same object
```

🔴 **Important**: Changing data via s2 also affects s1.

📌 **Example**:

```
s2.studentName = "Amit";

System.out.println(s1.studentName);  // Output: Amit
```



```
class Student {
   int rollNo;
   String name;
}
public class Demo {
   public static void main(String[] args) {
      Student s1 = new Student();
      s1.rollNo = 101;
      s1.name = "Rahul";

      Student s2 = s1;
      s2.name = "Amit";
      System.out.println(s1.name);  // Output: Amit
   }
}
```

State of an Object

What is Object State?

The **state** of an object refers to the values stored in its instance variables (fields) at a specific point in time.

How is State Represented?

- Through **instance variables** (non-static fields)
- Each object has its own copy of instance variables
- State can change throughout the object's lifecycle

```
public class BankAccount {
  // Instance variables represent state
  private String accountNumber;
  private String accountHolder;
  private double balance;

  // Constructor initializes state
  public BankAccount(String number, String holder) {
    this.accountNumber = number;
    this.accountHolder = holder;
    this.balance = 0.0;
  }
}
```

State Management

Initializing State:

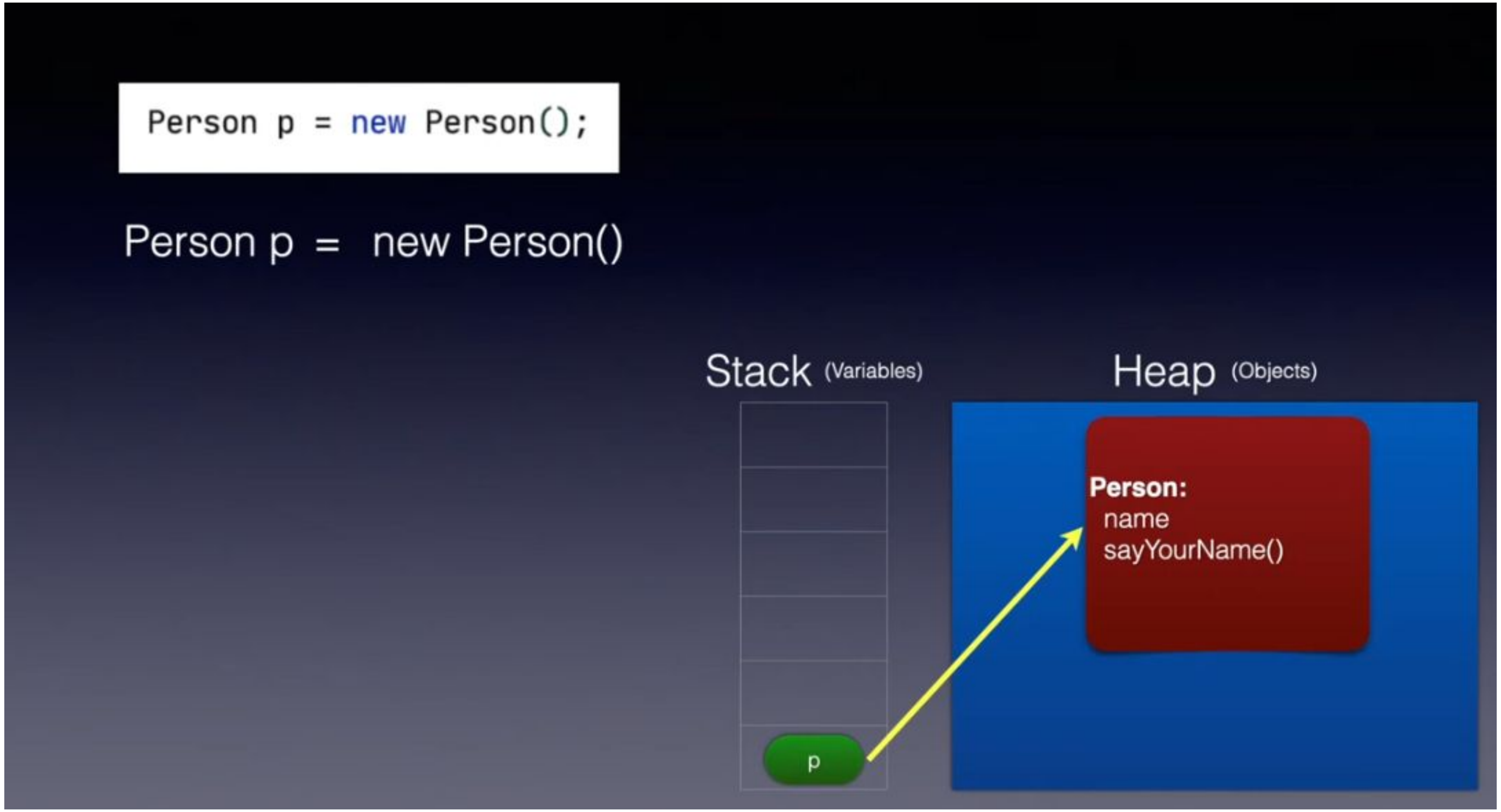
- Through **constructors**
- Through default values
- Through initialization blocks

Modifying State:

- Through **methods** (behavior)
- Through setters (encapsulation)

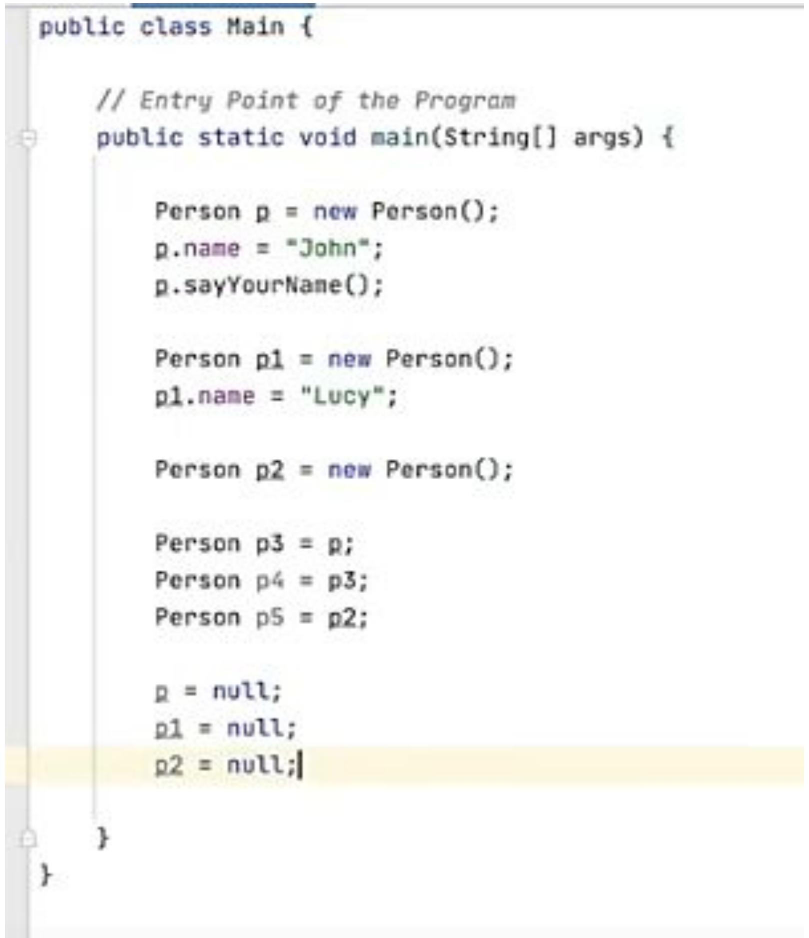
```
// Modifying state through methods
public void deposit(double amount) {
  if (amount > 0) {
    this.balance += amount;
  }
}

// Accessing state
public double getBalance() {
  return this.balance;
}
```

Memory allocation

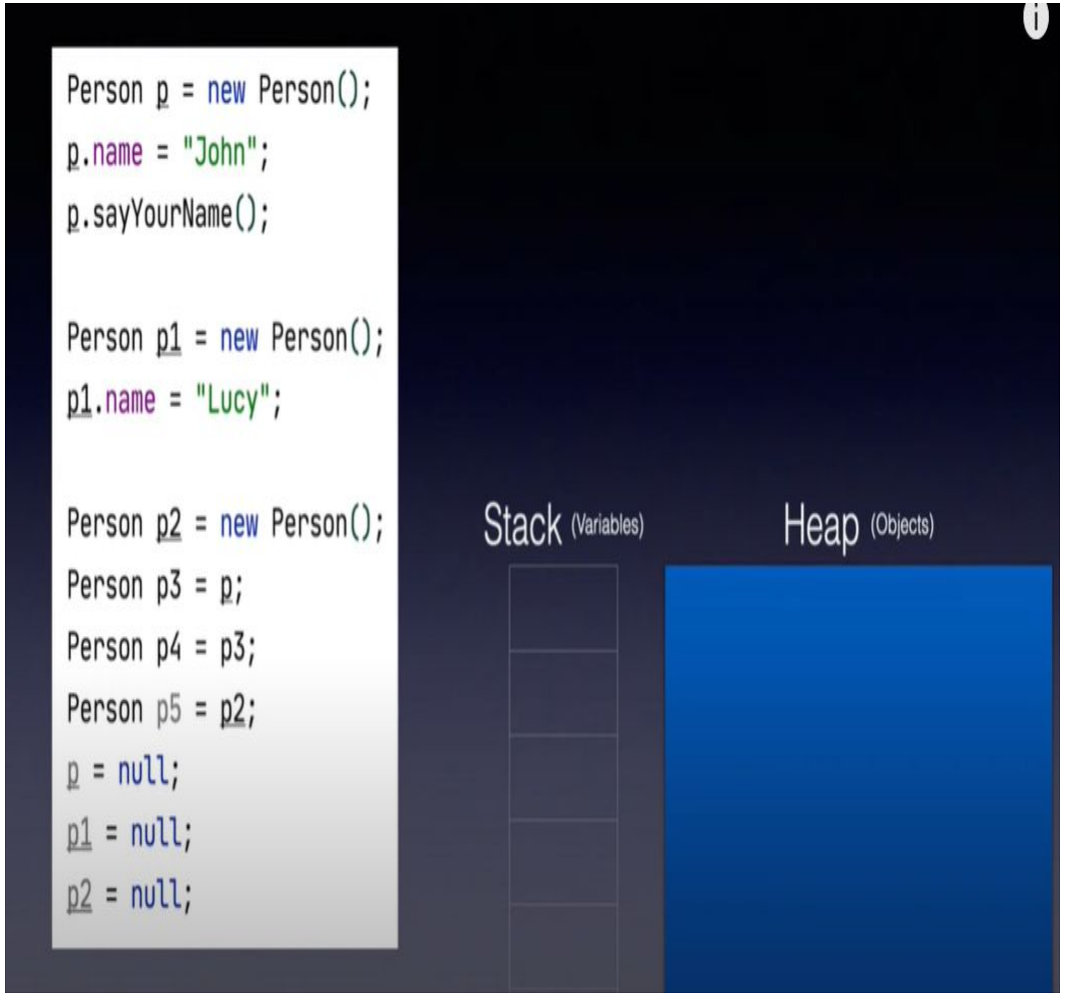


```
Person p = new Person();
```

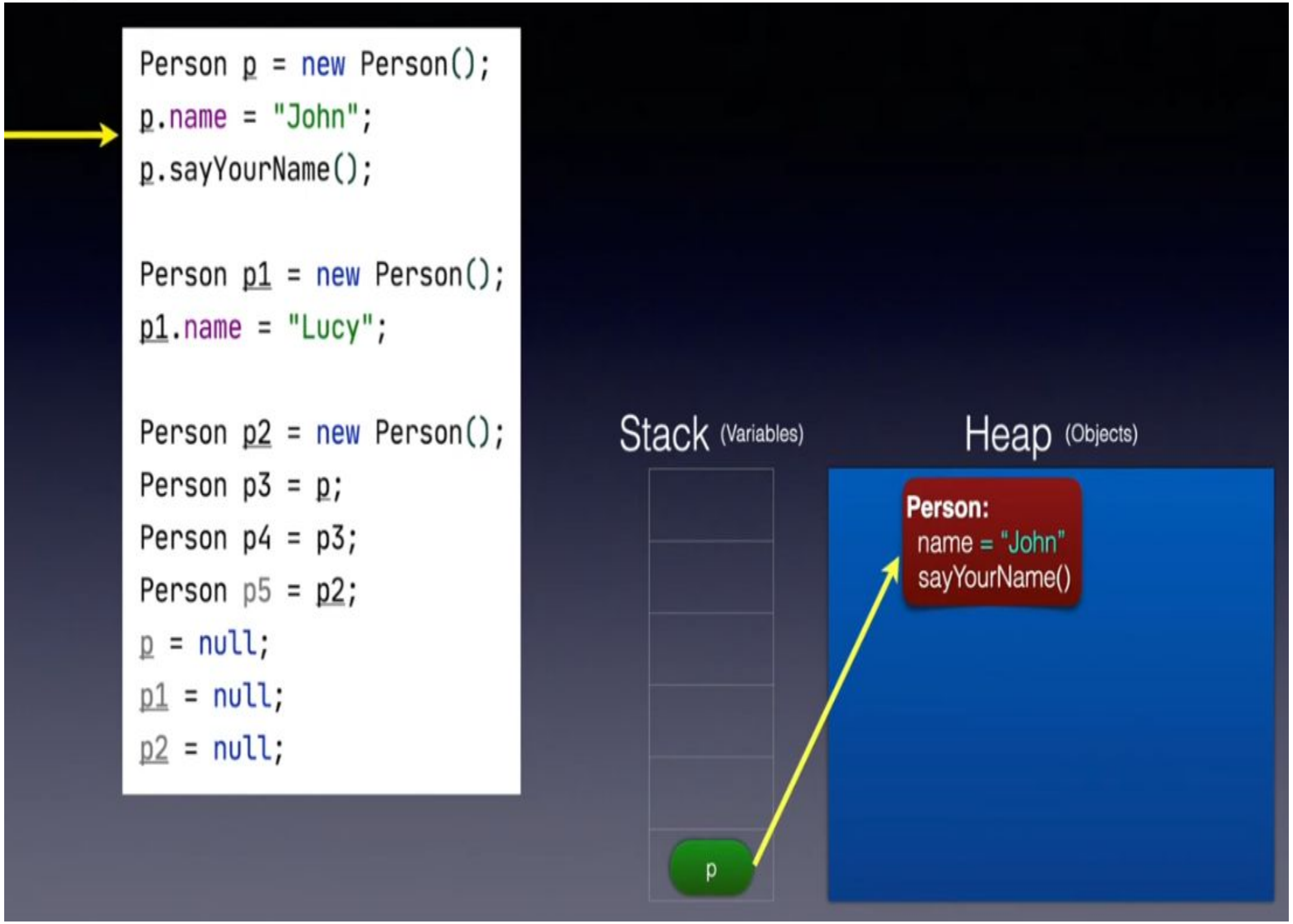
Person p = new Person()



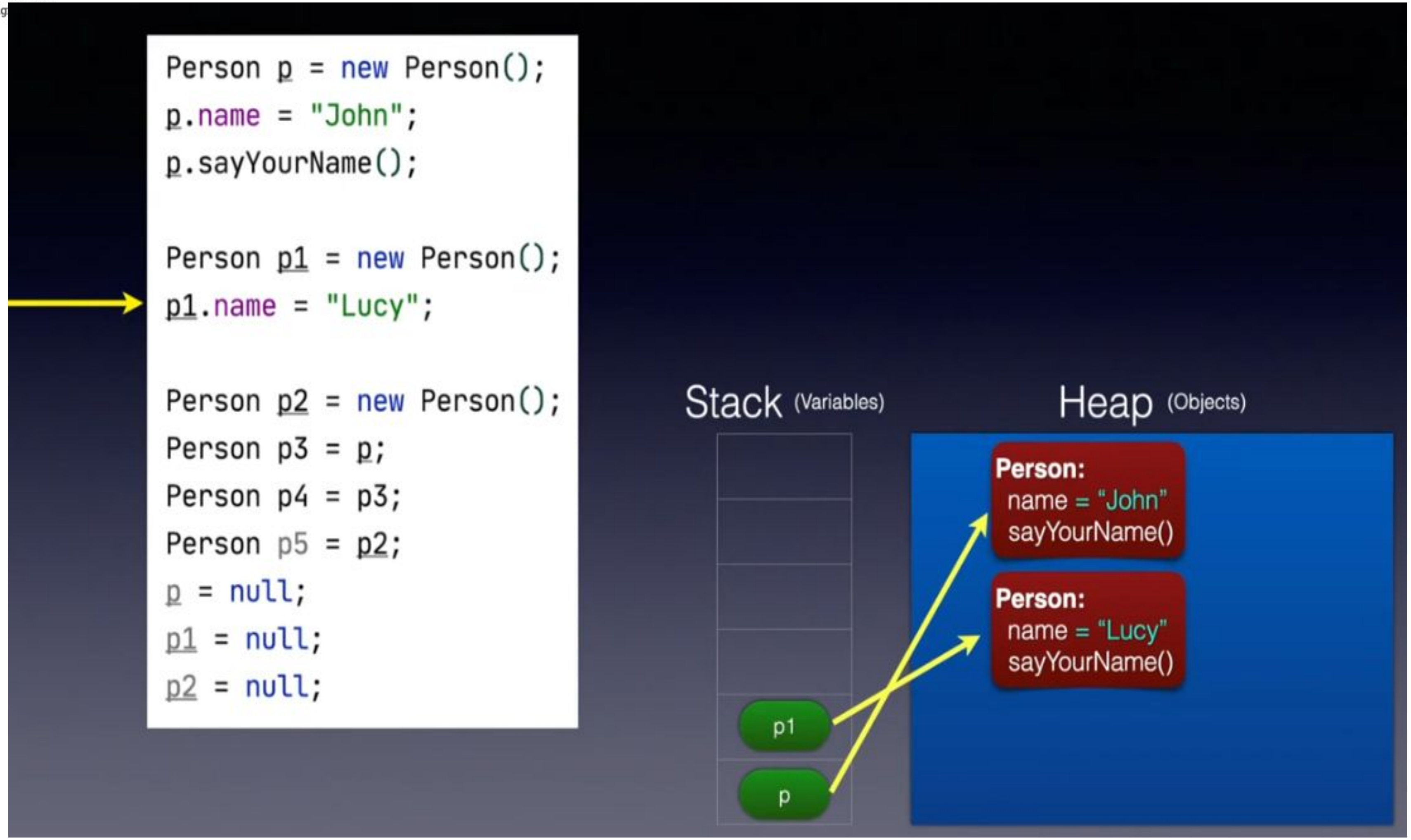
```java
public class Main {

    // Entry Point of the Program
    public static void main(String[] args) {

        Person p = new Person();
        p.name = "John";
        p.sayYourName();

        Person p1 = new Person();
        p1.name = "Lucy";

        Person p2 = new Person();

        Person p3 = p;
        Person p4 = p3;
        Person p5 = p2;

        p = null;
        p1 = null;
        p2 = null;

    }
}
```



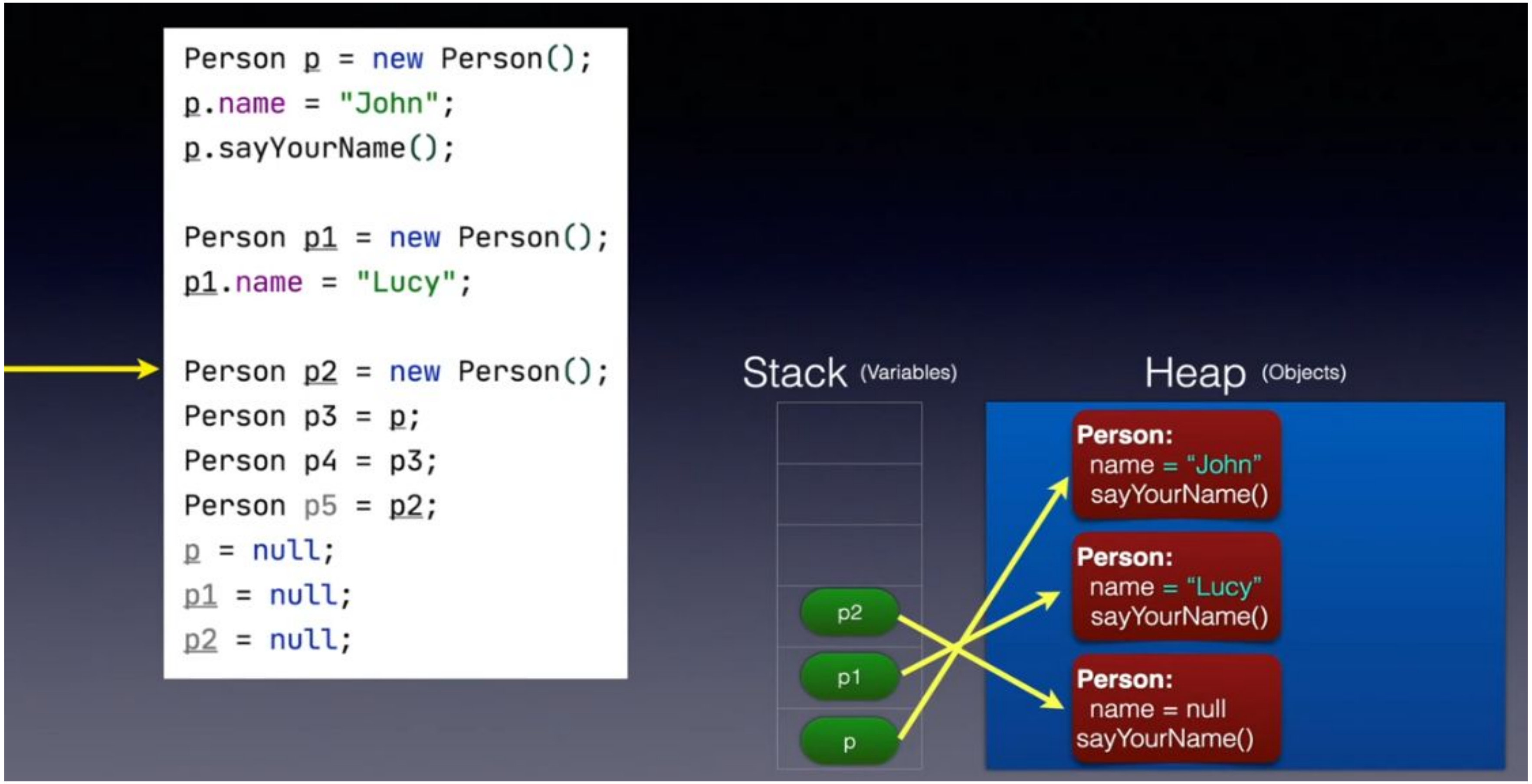
```java
Person p = new Person();
p.name = "John";
p.sayYourName();

Person p1 = new Person();
p1.name = "Lucy";

Person p2 = new Person();
Person p3 = p;
Person p4 = p3;
Person p5 = p2;
p = null;
p1 = null;
p2 = null;
```



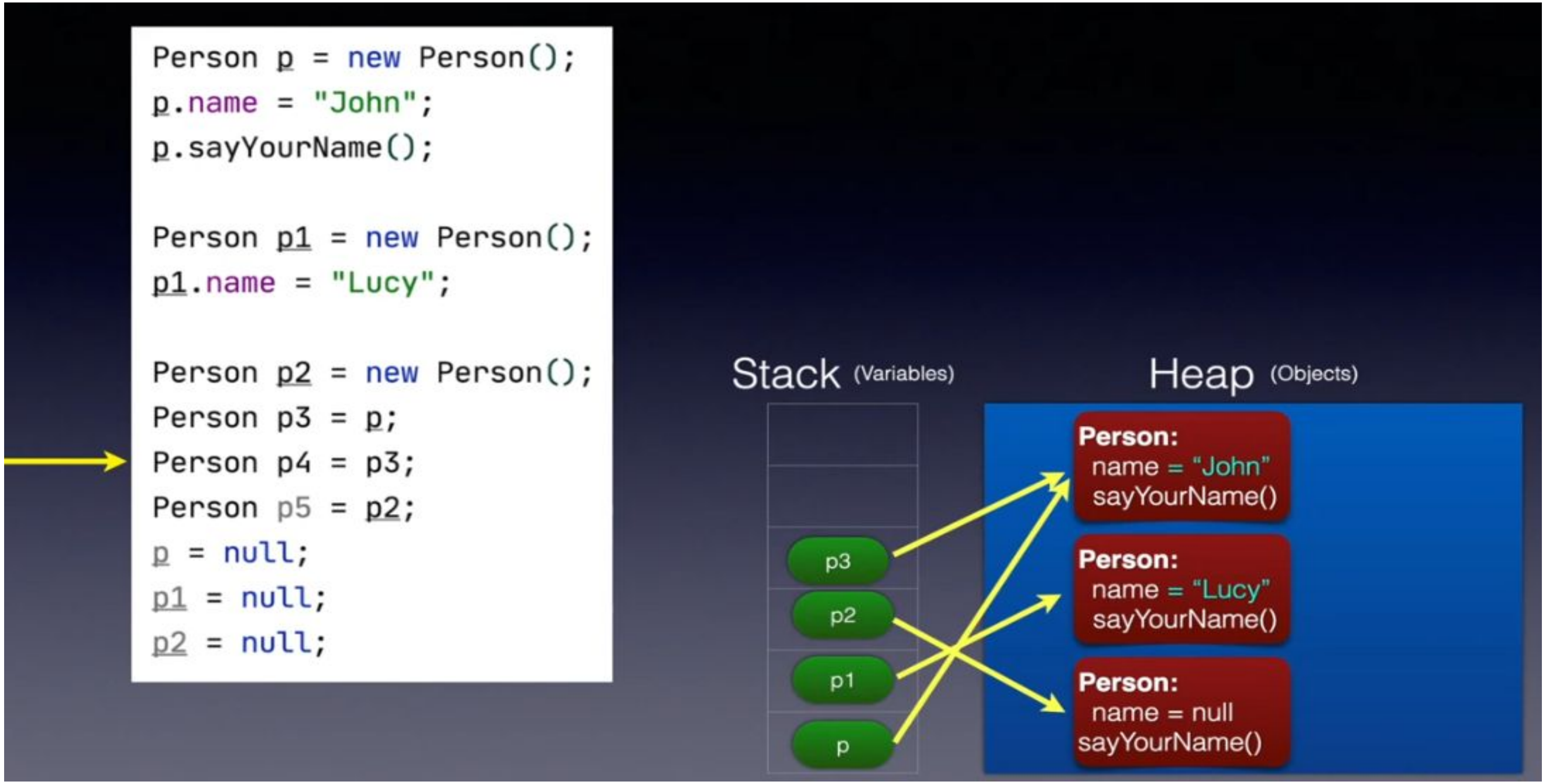
```
Person p = new Person();
p.name = "John";
p.sayYourName();

Person p1 = new Person();
p1.name = "Lucy";

Person p2 = new Person();
Person p3 = p;
Person p4 = p3;
Person p5 = p2;
p = null;
p1 = null;
p2 = null;
```



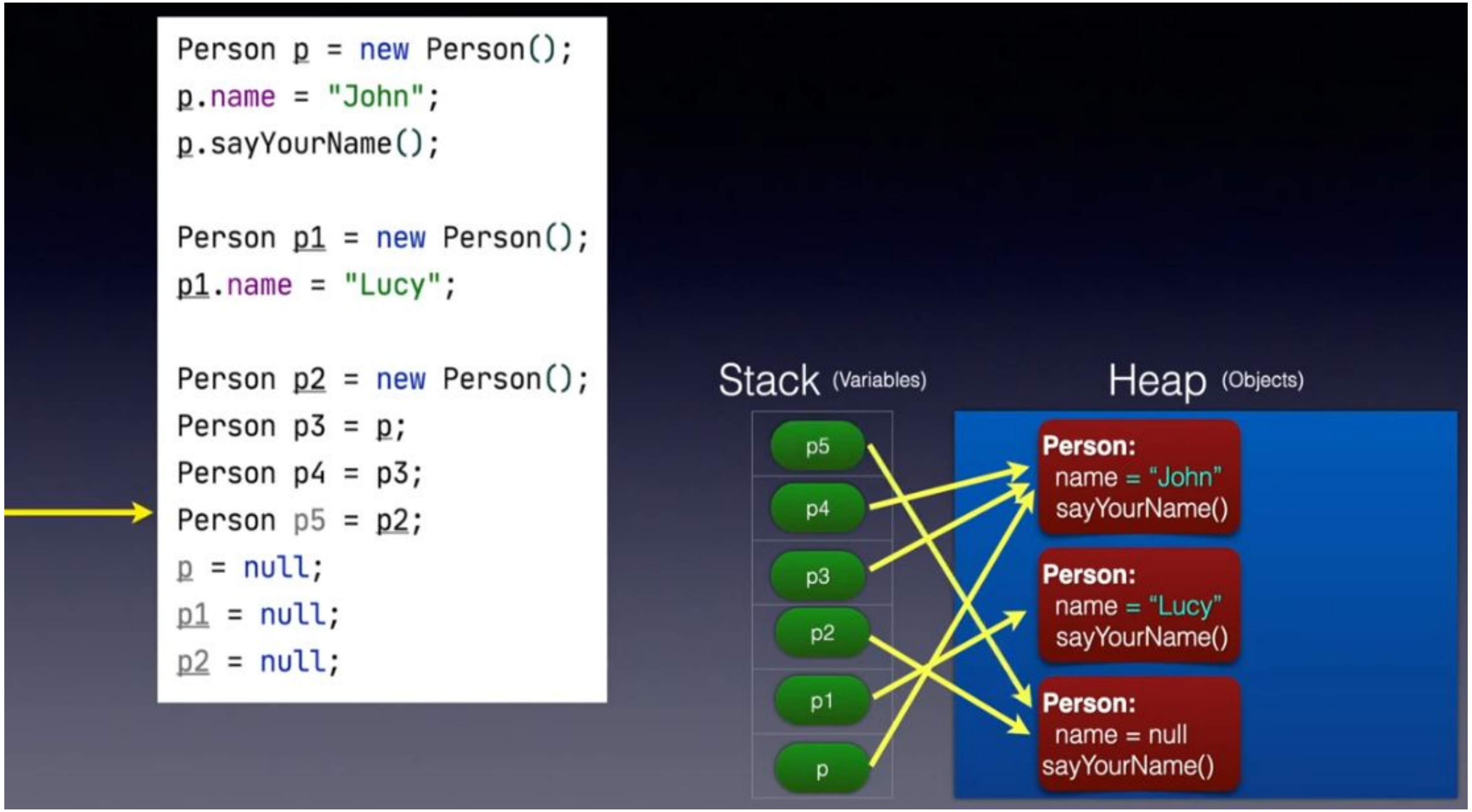
```
Person p = new Person();
p.name = "John";
p.sayYourName();

Person p1 = new Person();
p1.name = "Lucy";

Person p2 = new Person();
Person p3 = p;
Person p4 = p3;
Person p5 = p2;
p = null;
p1 = null;
p2 = null;
```



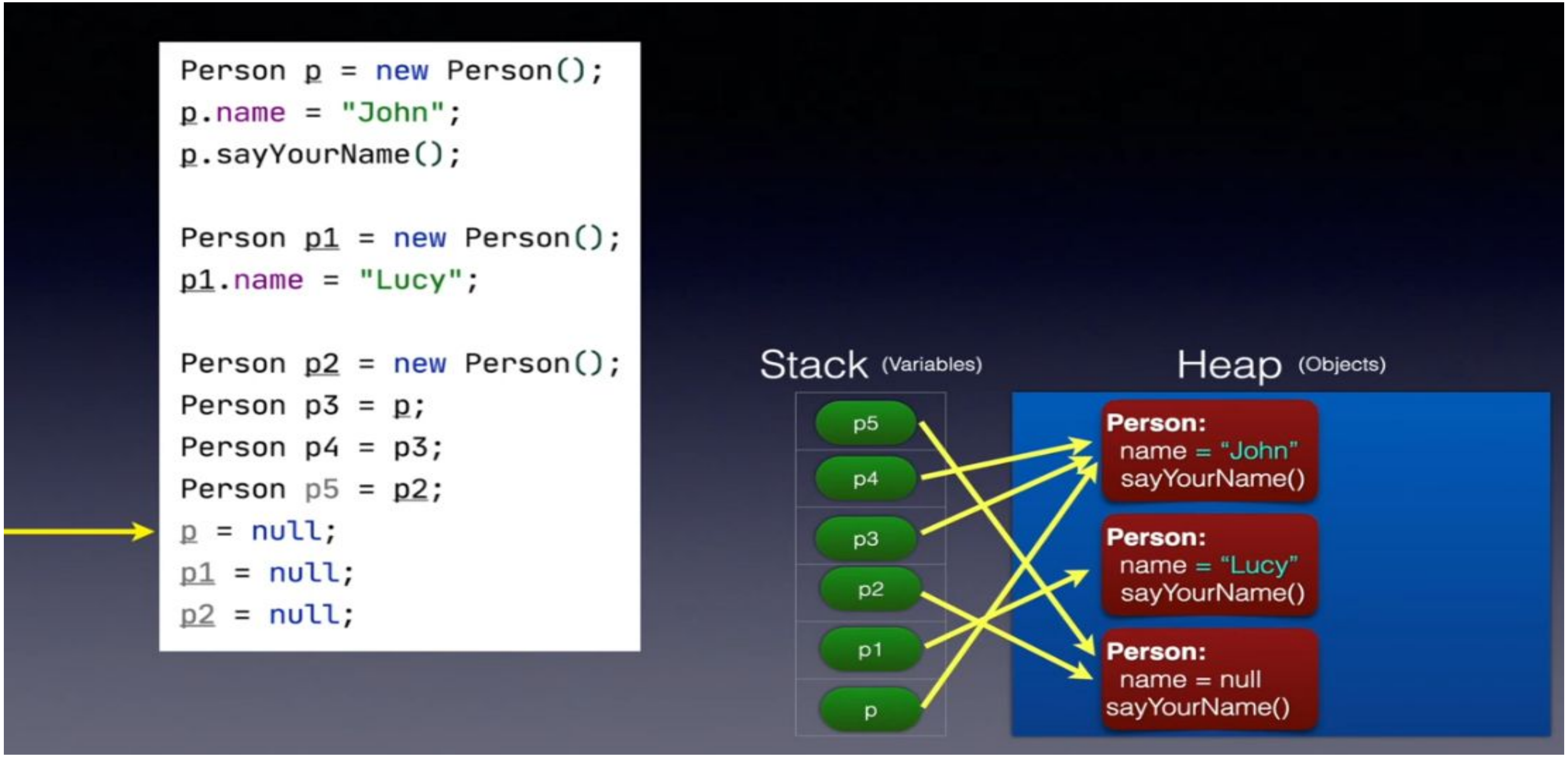
```
Person p = new Person();
p.name = "John";
p.sayYourName();

Person p1 = new Person();
p1.name = "Lucy";

Person p2 = new Person();
Person p3 = p;
Person p4 = p3;
Person p5 = p2;
p = null;
p1 = null;
p2 = null;
```



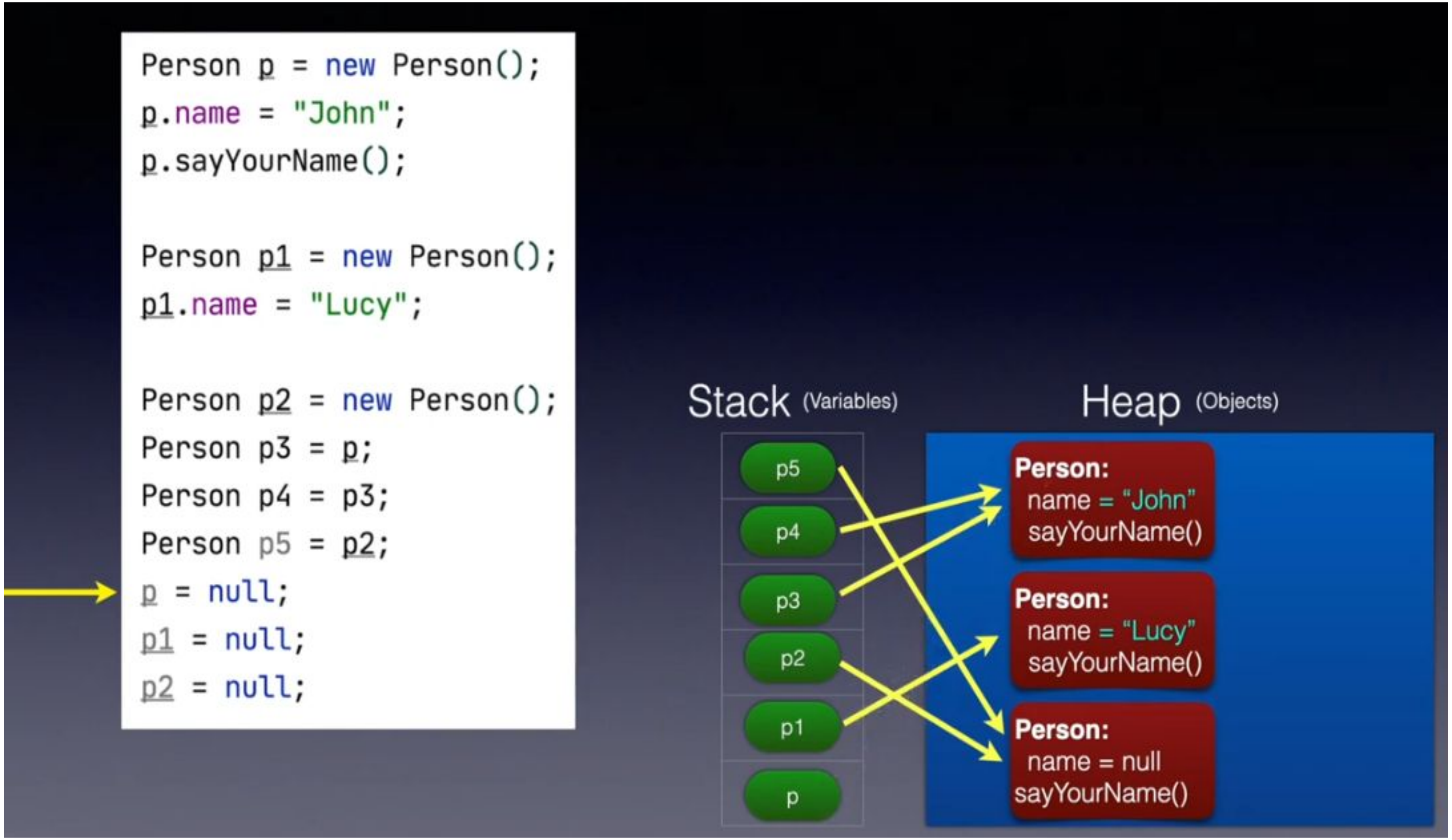
```
Person p = new Person();
p.name = "John";
p.sayYourName();

Person p1 = new Person();
p1.name = "Lucy";

Person p2 = new Person();
Person p3 = p;
Person p4 = p3;
Person p5 = p2;
p = null;
p1 = null;
p2 = null;
```



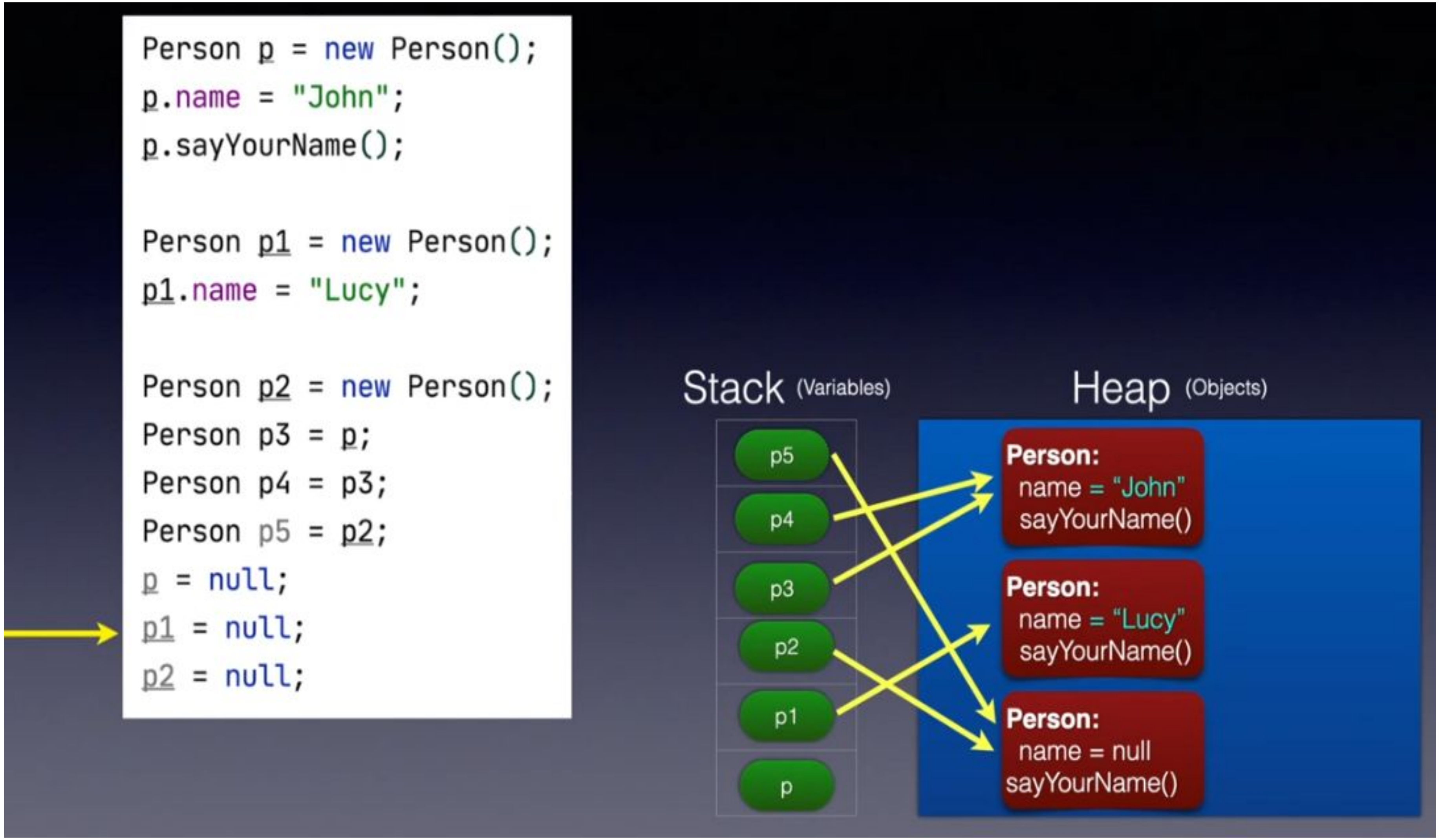
```
Person p = new Person();
p.name = "John";
p.sayYourName();

Person p1 = new Person();
p1.name = "Lucy";

Person p2 = new Person();
Person p3 = p;
Person p4 = p3;
Person p5 = p2;
p = null;
p1 = null;
p2 = null;
```



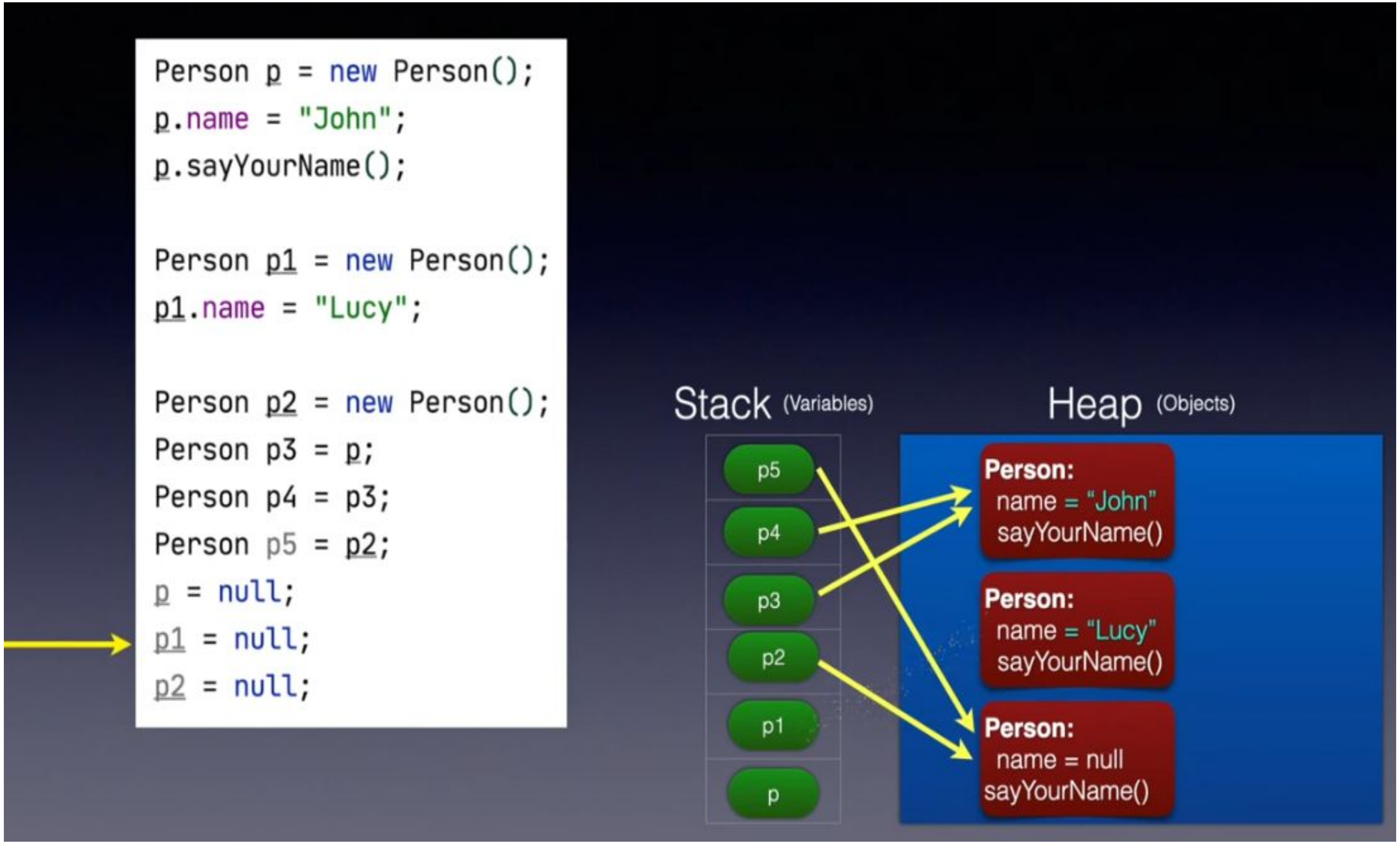
```
Person p = new Person();
p.name = "John";
p.sayYourName();

Person p1 = new Person();
p1.name = "Lucy";

Person p2 = new Person();
Person p3 = p;
Person p4 = p3;
Person p5 = p2;
p = null;
p1 = null;
p2 = null;
```



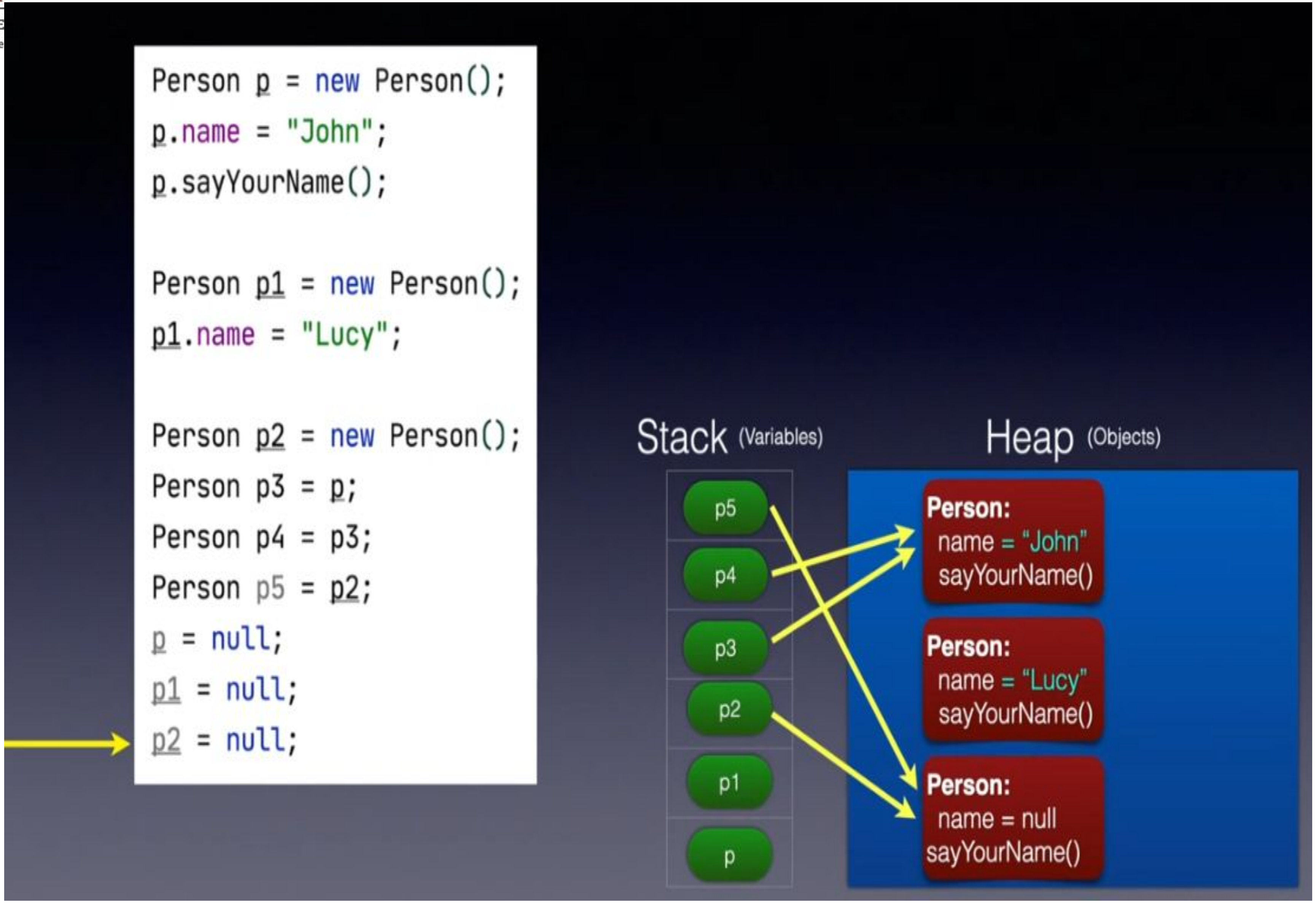
```
Person p = new Person();
p.name = "John";
p.sayYourName();

Person p1 = new Person();
p1.name = "Lucy";

Person p2 = new Person();
Person p3 = p;
Person p4 = p3;
Person p5 = p2;
p = null;
p1 = null;
p2 = null;
```



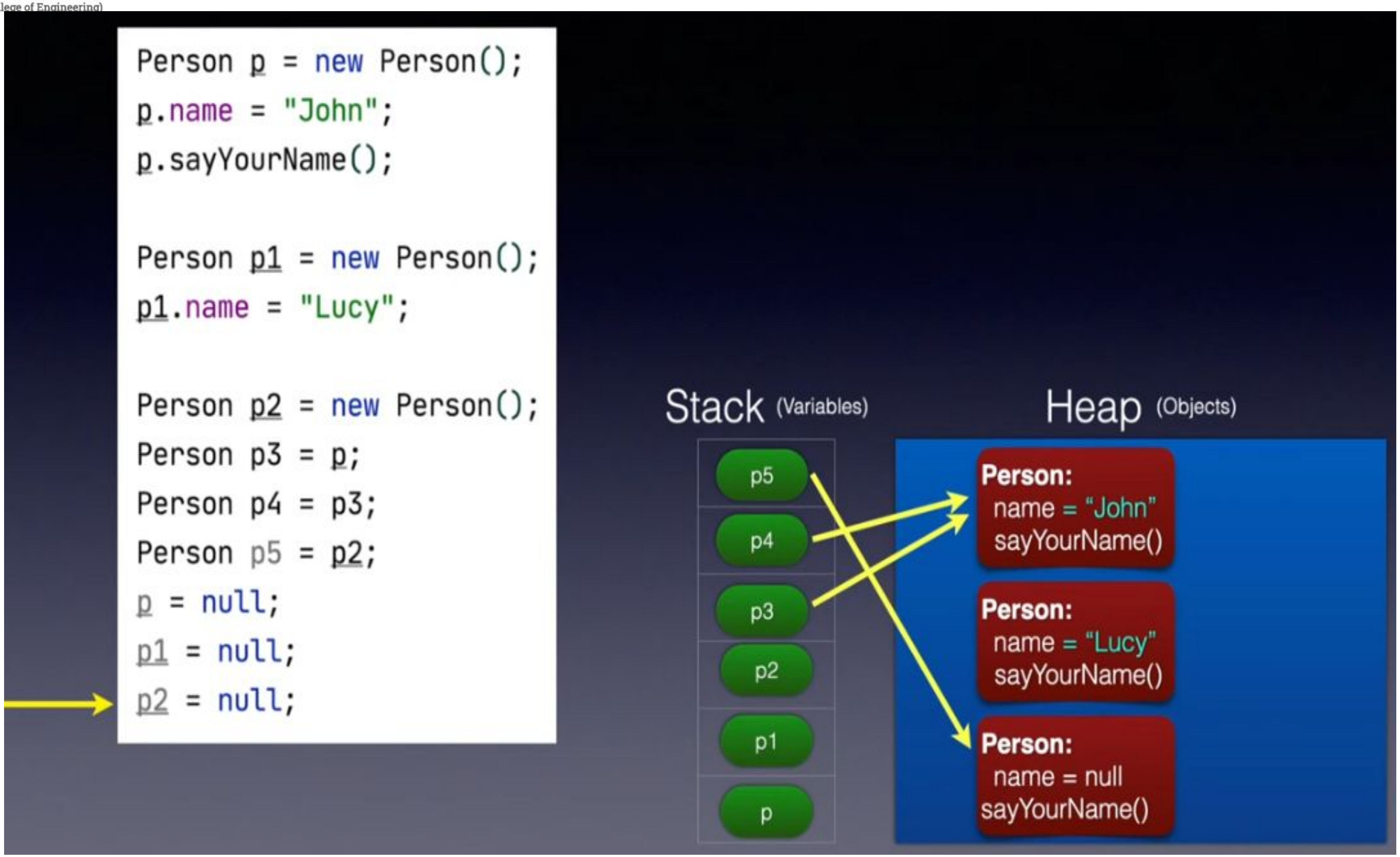
```
Person p = new Person();
p.name = "John";
p.sayYourName();

Person p1 = new Person();
p1.name = "Lucy";

Person p2 = new Person();
Person p3 = p;
Person p4 = p3;
Person p5 = p2;
p = null;
p1 = null;
p2 = null;
```



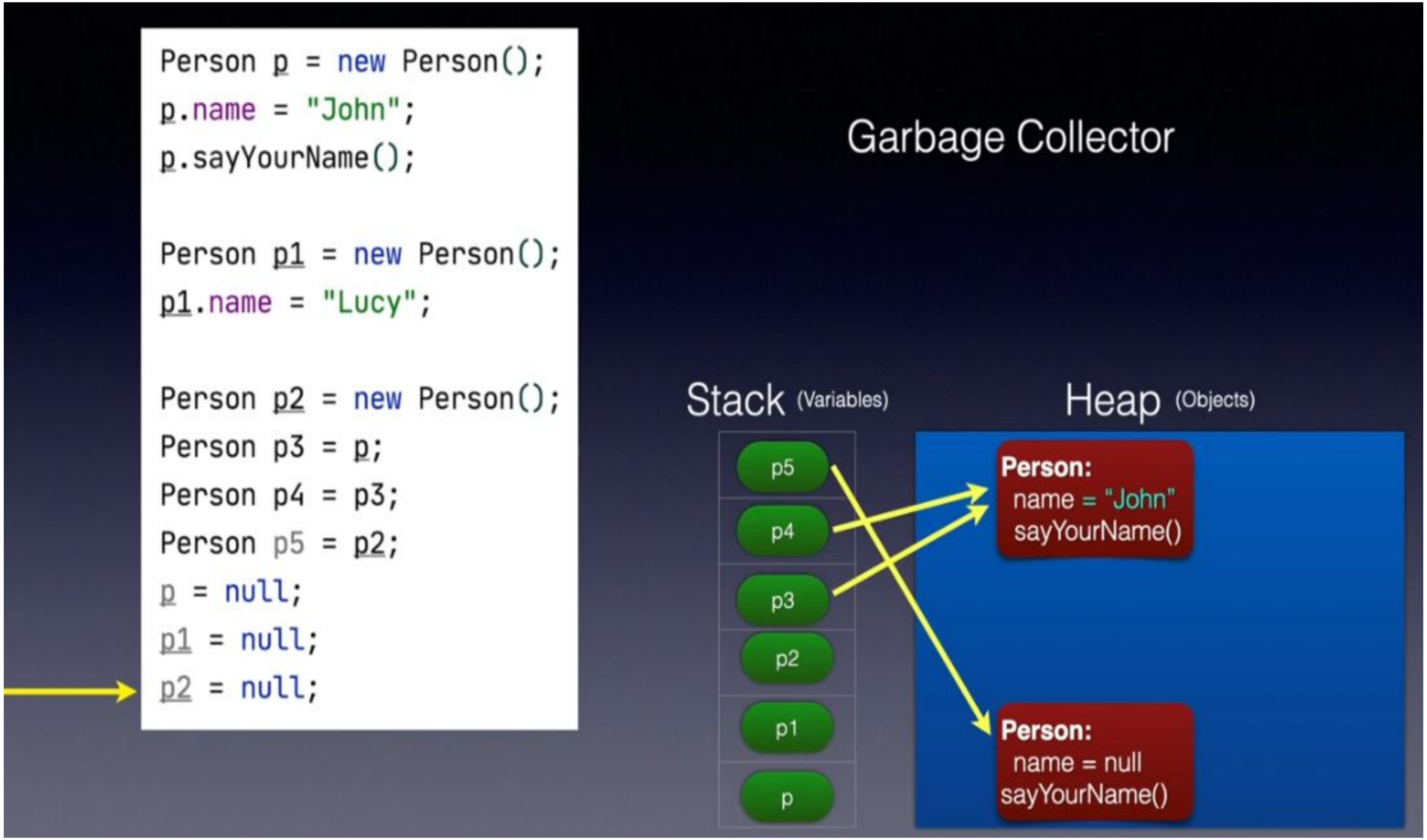
```
Person p = new Person();
p.name = "John";
p.sayYourName();

Person p1 = new Person();
p1.name = "Lucy";

Person p2 = new Person();
Person p3 = p;
Person p4 = p3;
Person p5 = p2;
p = null;
p1 = null;
p2 = null;
```



```
Person p = new Person();
p.name = "John";
p.sayYourName();

Person p1 = new Person();
p1.name = "Lucy";

Person p2 = new Person();
Person p3 = p;
Person p4 = p3;
Person p5 = p2;
p = null;
p1 = null;
p2 = null;
```



```
Person p = new Person();
p.name = "John";
p.sayYourName();

Person p1 = new Person();
p1.name = "Lucy";

Person p2 = new Person();
Person p3 = p;
Person p4 = p3;
Person p5 = p2;
p = null;
p1 = null;
p2 = null;
```



Garbage Collector

```
Person p = new Person();
p.name = "John";
p.sayYourName();

Person p1 = new Person();
p1.name = "Lucy";

Person p2 = new Person();
Person p3 = p;
Person p4 = p3;
Person p5 = p2;
p = null;
p1 = null;
p2 = null;
```

Command Line Arguments

What are Command Line Arguments?

Command line arguments are parameters passed to a Java program when it is executed from the command line.

Key Points:

- Arguments are passed to the **main()** method as a String array
- The array is named **args** by convention
- Arguments are separated by spaces when entered
- Arguments are accessed by index (zero-based)
- Arguments are always passed as strings

```
public class CommandLineExample {
    public static void main(String[] args) {
        // Check if arguments were provided
        if (args.length > 0) {
            System.out.println("Arguments provided: " + args.len

            // Print all arguments
            for (int i = 0; i < args.length; i++) {
                System.out.println("Argument " + i + ": " + args[i]);
            }
        } else {
            System.out.println("No arguments provided");
        }
    }
}
```

Using Command Line Arguments

Common Use Cases:

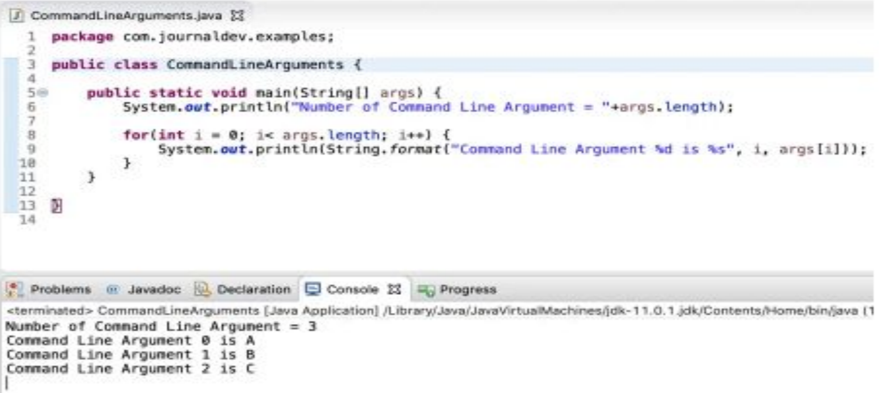
- Configuration options
- Input file paths
- Program modes or flags
- User credentials

Running with Arguments:

```
java CommandLineExample arg1 arg2 arg3
```

Output:

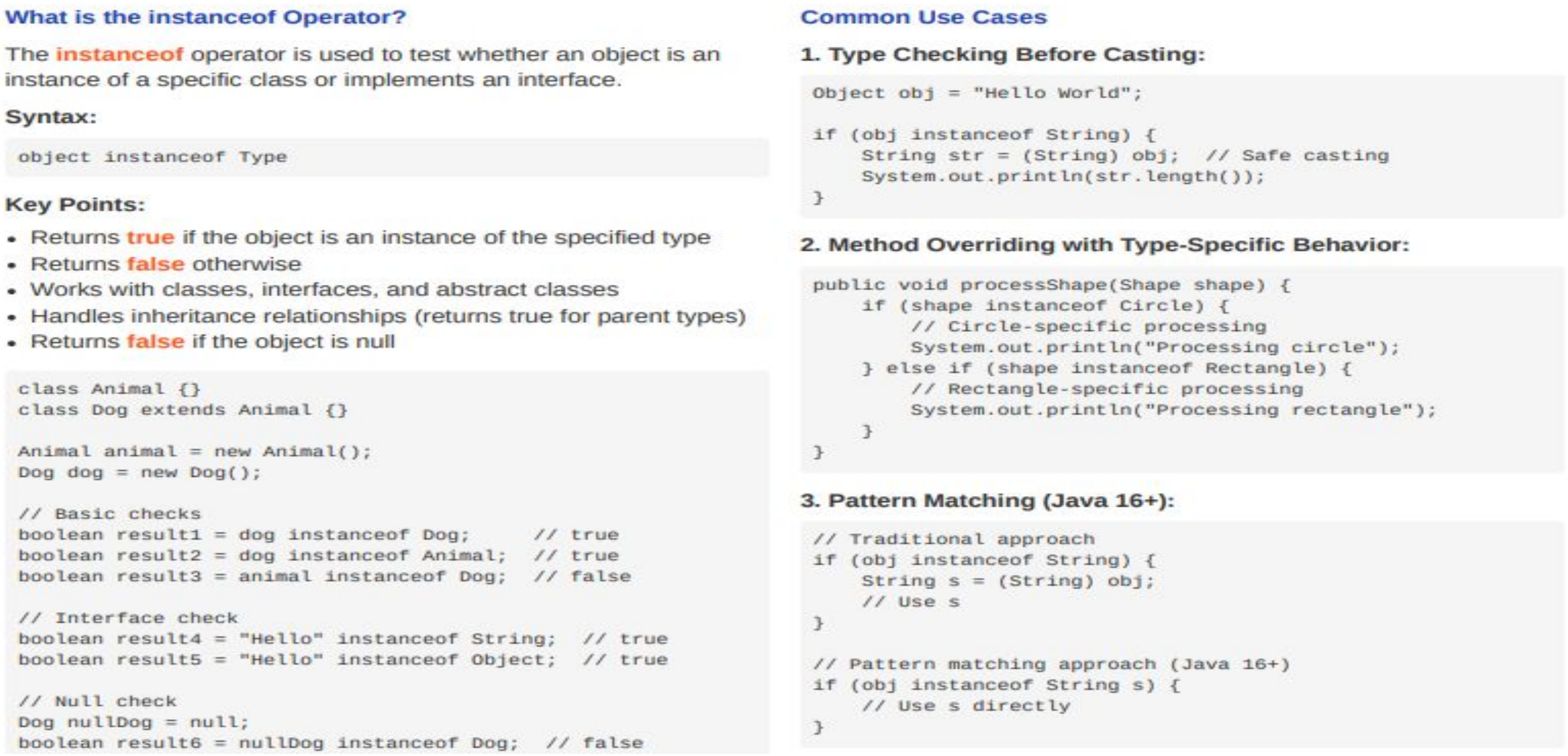
```
Arguments provided: 3
Argument 0: arg1
Argument 1: arg2
Argument 2: arg3
```



```
package com.journaldev.examples;

public class CommandLineArguments {

    public static void main(String[] args) {
        System.out.println("Number of Command Line Argument = "+args.length);

        for(int i = 0; i< args.length; i++) {
            System.out.println(String.format("Command Line Argument %d is %s", i, args[i]));
        }
    }

}
```

```
<terminated> CommandLineArguments [Java Application] /Library/Java/JavaVirtualMachines/jdk-11.0.1.jdk/Contents/Home/bin/java (1
Number of Command Line Argument = 3
Command Line Argument 0 is A
Command Line Argument 1 is B
Command Line Argument 2 is C
```

instanceof Operator



What is the instanceof Operator?

The **instanceof** operator is used to test whether an object is an instance of a specific class or implements an interface.

Syntax:

```
object instanceof Type
```

Key Points:

- Returns **true** if the object is an instance of the specified type
- Returns **false** otherwise
- Works with classes, interfaces, and abstract classes
- Handles inheritance relationships (returns true for parent types)
- Returns **false** if the object is null

```
class Animal {}
class Dog extends Animal {}

Animal animal = new Animal();
Dog dog = new Dog();

// Basic checks
boolean result1 = dog instanceof Dog;      // true
boolean result2 = dog instanceof Animal;   // true
boolean result3 = animal instanceof Dog;   // false

// Interface check
boolean result4 = "Hello" instanceof String;  // true
boolean result5 = "Hello" instanceof Object;  // true

// Null check
Dog nullDog = null;
boolean result6 = nullDog instanceof Dog;  // false
```

Common Use Cases

1. Type Checking Before Casting:

```
Object obj = "Hello World";

if (obj instanceof String) {
    String str = (String) obj;  // Safe casting
    System.out.println(str.length());
}
```

2. Method Overriding with Type-Specific Behavior:

```
public void processShape(Shape shape) {
    if (shape instanceof Circle) {
        // Circle-specific processing
        System.out.println("Processing circle");
    } else if (shape instanceof Rectangle) {
        // Rectangle-specific processing
        System.out.println("Processing rectangle");
    }
}
```

3. Pattern Matching (Java 16+):

```
// Traditional approach
if (obj instanceof String) {
    String s = (String) obj;
    // Use s
}

// Pattern matching approach (Java 16+)
if (obj instanceof String s) {
    // Use s directly
}
```

Problem:

Create a class Book with instance variables:

- title, author, price

Write a main() method that:

- Creates 2 book objects
- Assigns values
- Displays details using : System.out.println()

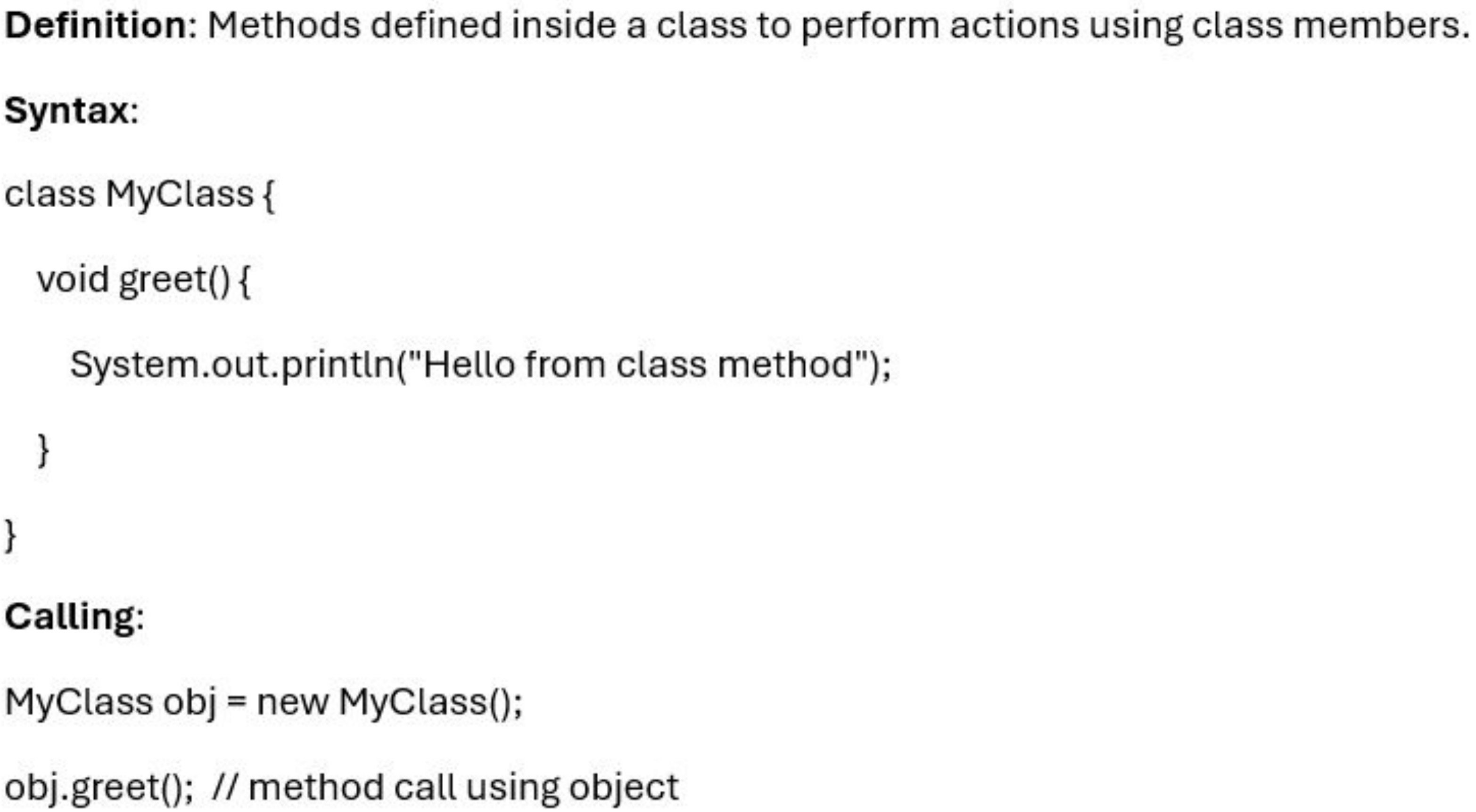
Summary

- Java class = blueprint for objects
- Instance variables = per-object data
- Objects created using new
- Use dot operator to access members
- Object references can point to the same object

Class methods

- Return values
- Method that takes parameters
- Member vs local variables
- Constructors (default and parameterized)
- Method and constructor overloading
- this keyword

Class Methods



Definition: Methods defined inside a class to perform actions using class members.

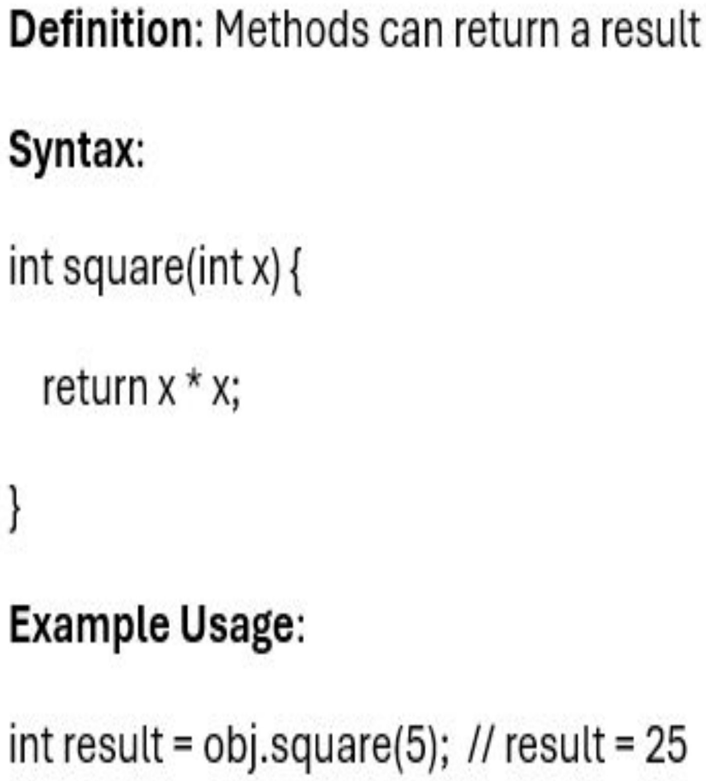
Syntax:

```
class MyClass {
   void greet() {
      System.out.println("Hello from class method");
   }
}
```

Calling:

```
MyClass obj = new MyClass();
obj.greet();  // method call using object
```

Returning a Value



Definition: Methods can return a result using the return keyword.

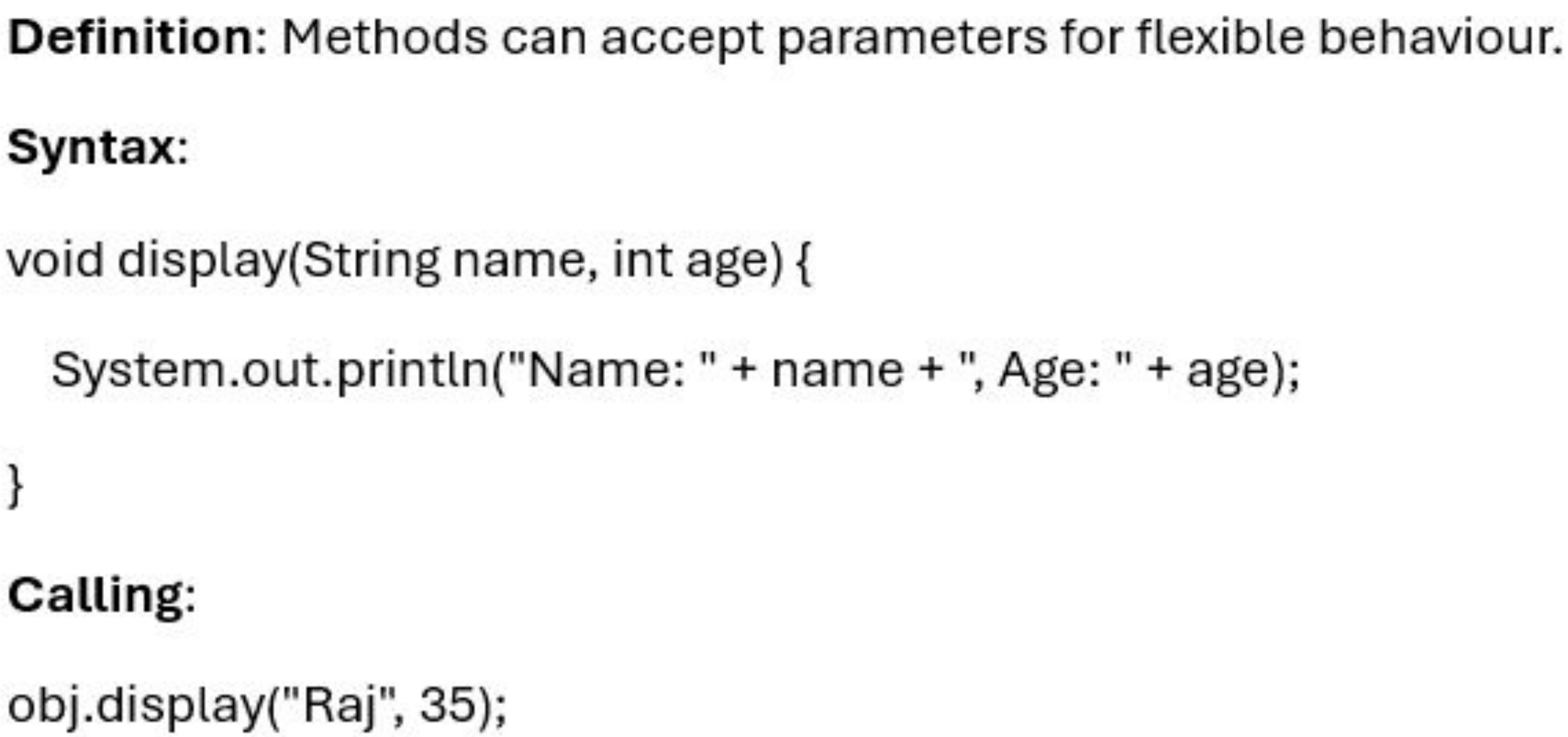
Syntax:

```
int square(int x) {
   return x * x;
}
```

Example Usage:

```
int result = obj.square(5);  // result = 25
```

Method That Takes Parameters



Definition: Methods can accept parameters for flexible behaviour.

Syntax:

```
void display(String name, int age) {
   System.out.println("Name: " + name + ", Age: " + age);
}
```

Calling:

```
obj.display("Raj", 35);
```

```
class Printer {
    void printMessage(String message, int times) {
        for (int i = 0; i < times; i++) {
            System.out.println(message);
        }
    }
}

// Usage
Printer printer = new Printer();
printer.printMessage("Hello", 3);

// Output:
// Hello
// Hello
// Hello
```

Member Variables and Local Variables

- Member variables (fields) are declared inside the class, but outside all methods. They belong to the object (or class if static).

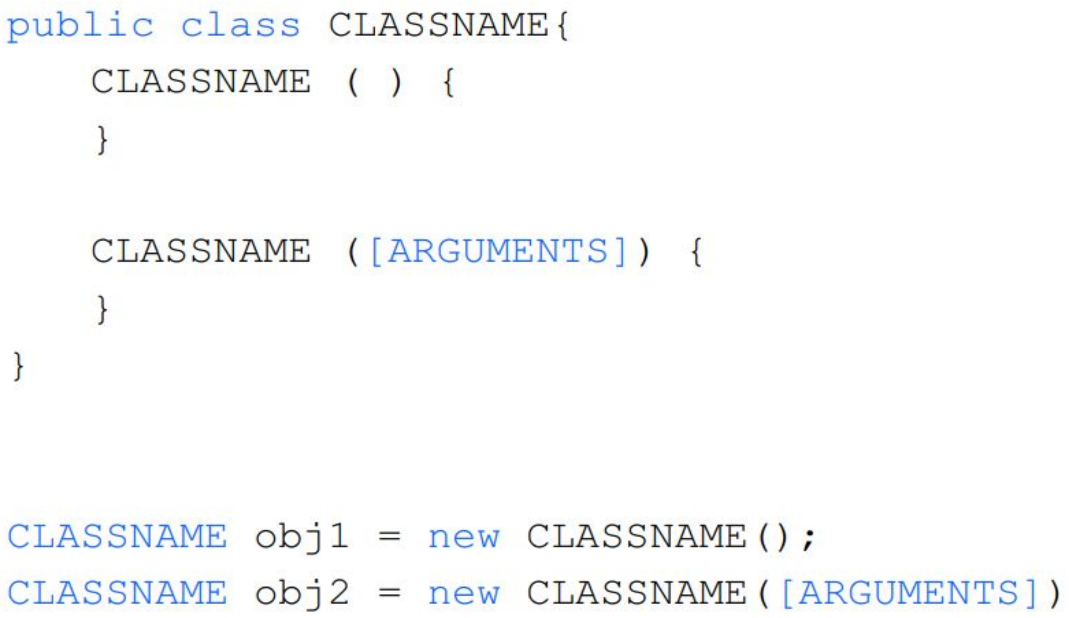
- Local variables are declared inside methods and only accessible within those methods

```
class Person {
    // Member variable
    String name;

  void setName(String newName)
      {

        // Local variable
         String prefix = "Mr./Ms. ";
         name = prefix + newName;
         }
   }
```

Feature	**Member Variable**	**Local Variable**
Scope	Whole class	Inside method or block only
Default Initialization	Yes (e.g., 0 for int, null for objects)	No (must be explicitly initialized)
Storage Location	Heap (as part of object)	Stack
Example	int id;	int temp = 10; inside a method

- Constructor,
- parameterised constructor,
- method overloading,
- constructor overloading,
- this keyword,
- Call by Value and call by reference, recursion,
- static fields, members,
- access protection,
- nested classes,
- Java a garbage collected,
- finalization

Constructors



```
public class CLASSNAME{
    CLASSNAME ( ) {
    }

    CLASSNAME ([ARGUMENTS]) {
    }
}


CLASSNAME obj1 = new CLASSNAME();
CLASSNAME obj2 = new CLASSNAME([ARGUMENTS])
```

A constructor is a special method used to initialize new objects. Its name matches the class name and it has no return type. It's called automatically when an object is created.

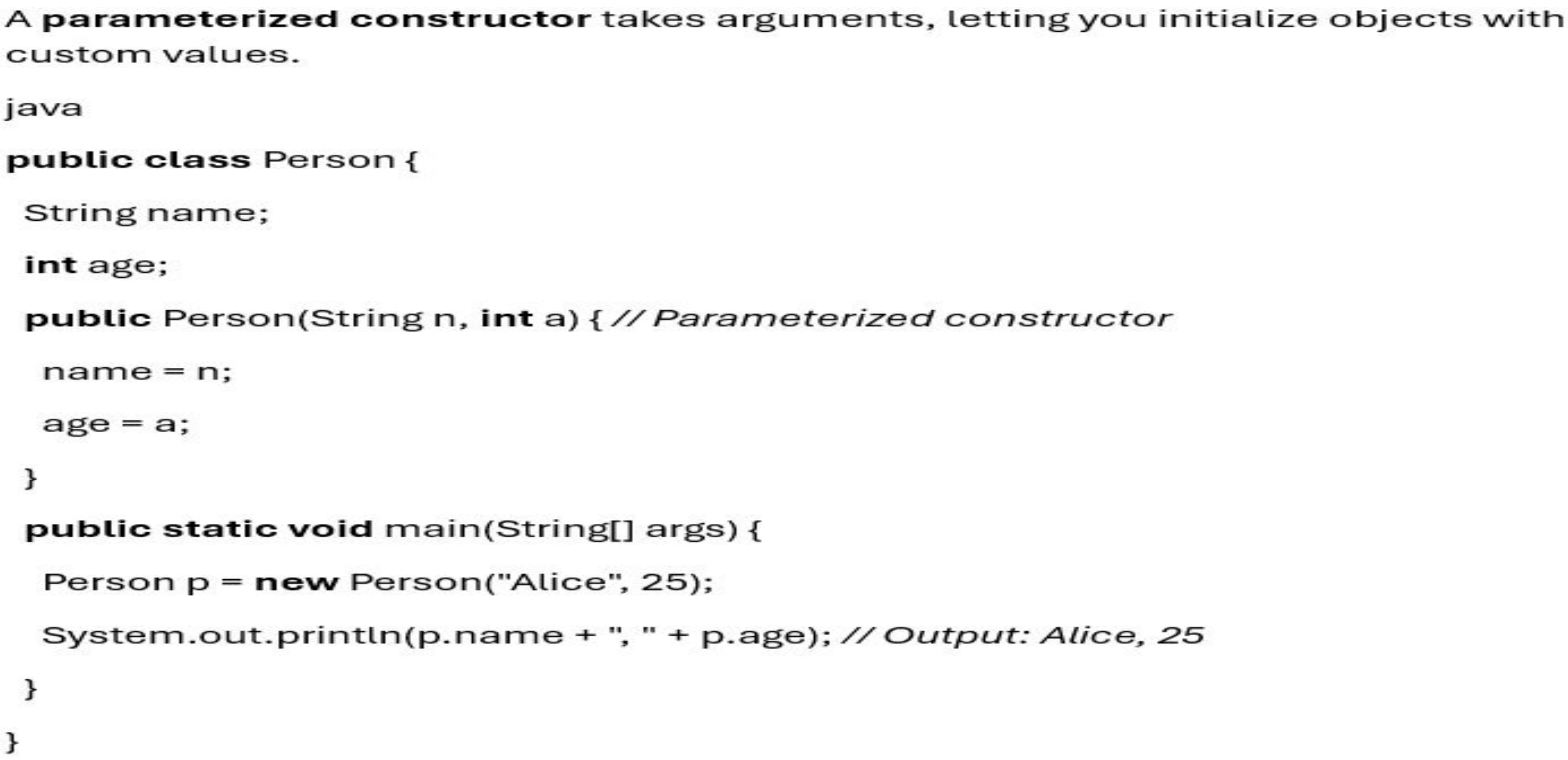
Constructors

- Constructor name == the class name
- No return type – never returns anything
- Usually initialize fields
- All classes need at least one constructor
  - If you don't write one, defaults to

```
CLASSNAME () {
}
```

```
public class Main {
  int x;
  public Main() { // default Constructor
    x = 5;
  }
  public static void main (String[] args) {
    Main myObj = new Main();
    System.out.println(myObj.x); // Output: 5
  }
}
```

Parameterized Constructor

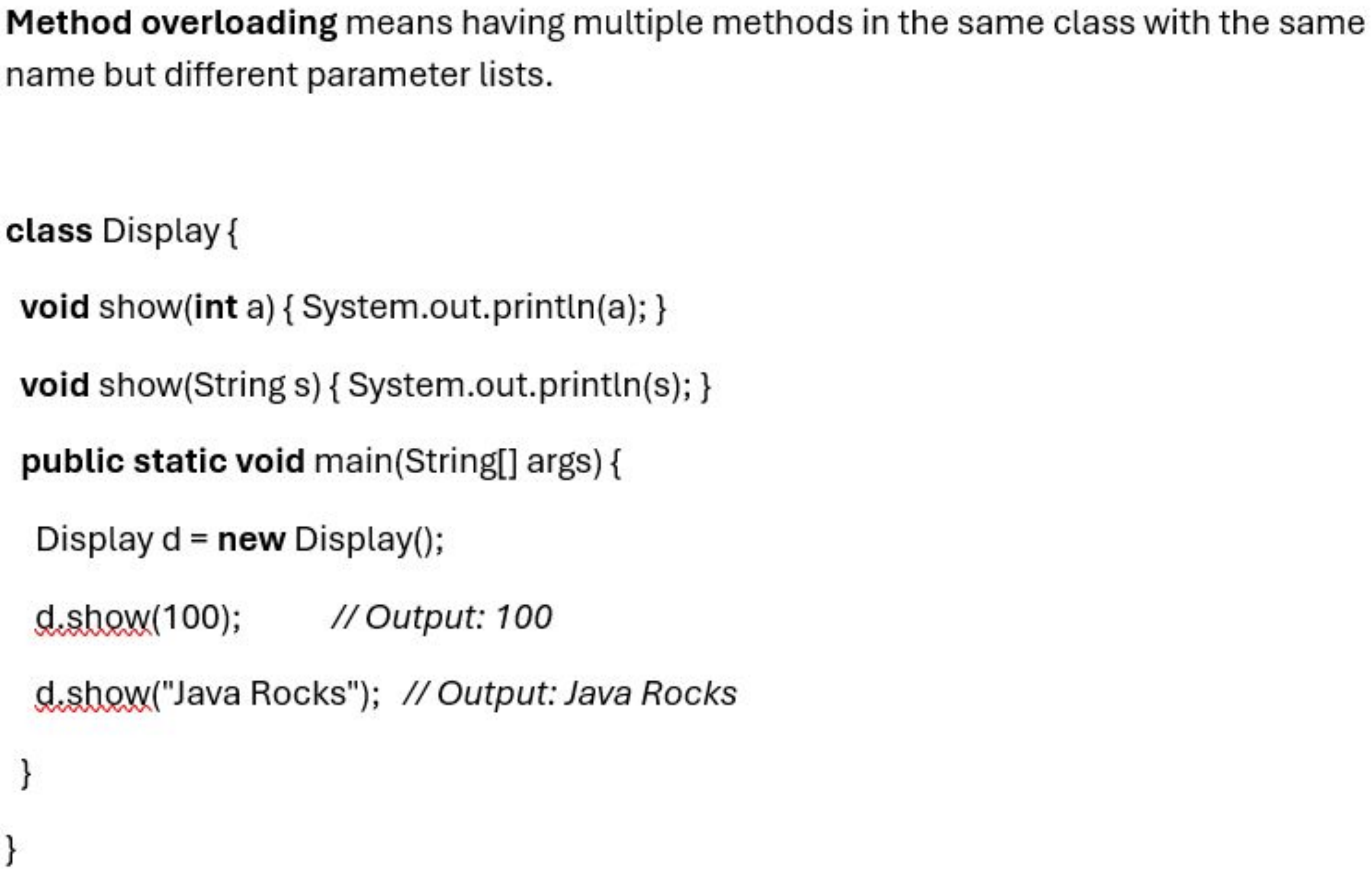


A **parameterized constructor** takes arguments, letting you initialize objects with custom values.

java

```
public class Person {
  String name;
  int age;
  public Person(String n, int a) { // Parameterized constructor
    name = n;
    age = a;
  }
  public static void main(String[] args) {
    Person p = new Person("Alice", 25);
    System.out.println(p.name + ", " + p.age); // Output: Alice, 25
  }
}
```

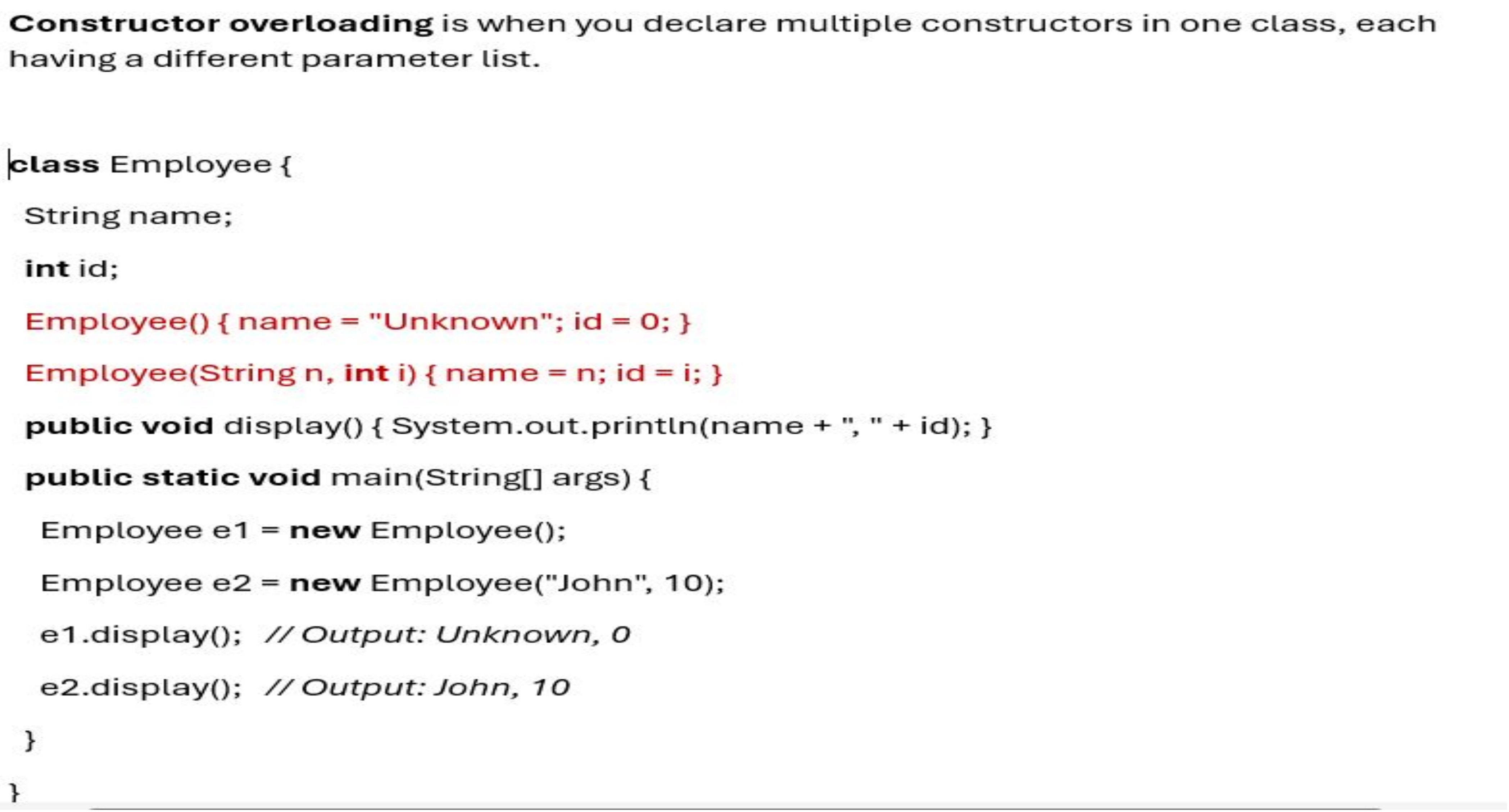
Method Overloading



Method overloading means having multiple methods in the same class with the same name but different parameter lists.

```
class Display {
 void show(int a) { System.out.println(a); }
 void show(String s) { System.out.println(s); }
 public static void main(String[] args) {
  Display d = new Display();
  d.show(100);         // Output: 100
  d.show("Java Rocks");  // Output: Java Rocks
 }
}
```

Constructor Overloading

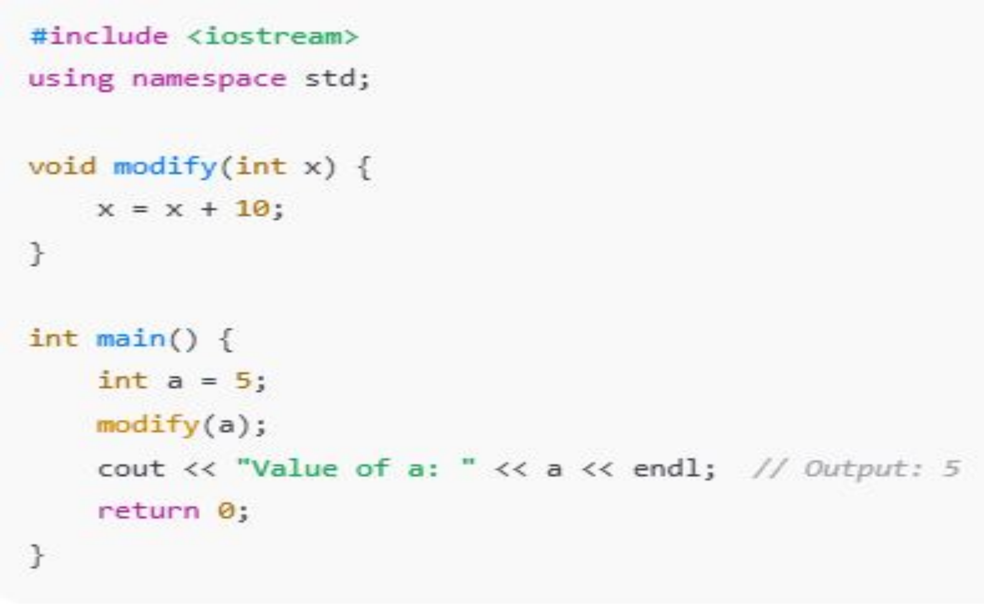


Constructor overloading is when you declare multiple constructors in one class, each having a different parameter list.

```
class Employee {
 String name;
 int id;
 Employee() { name = "Unknown"; id = 0; }
 Employee(String n, int i) { name = n; id = i; }
 public void display() { System.out.println(name + ", " + id); }
 public static void main(String[] args) {
  Employee e1 = new Employee();
  Employee e2 = new Employee("John", 10);
  e1.display();  // Output: Unknown, 0
  e2.display();  // Output: John, 10
 }
}
```

Call by Value and Call by Reference C++

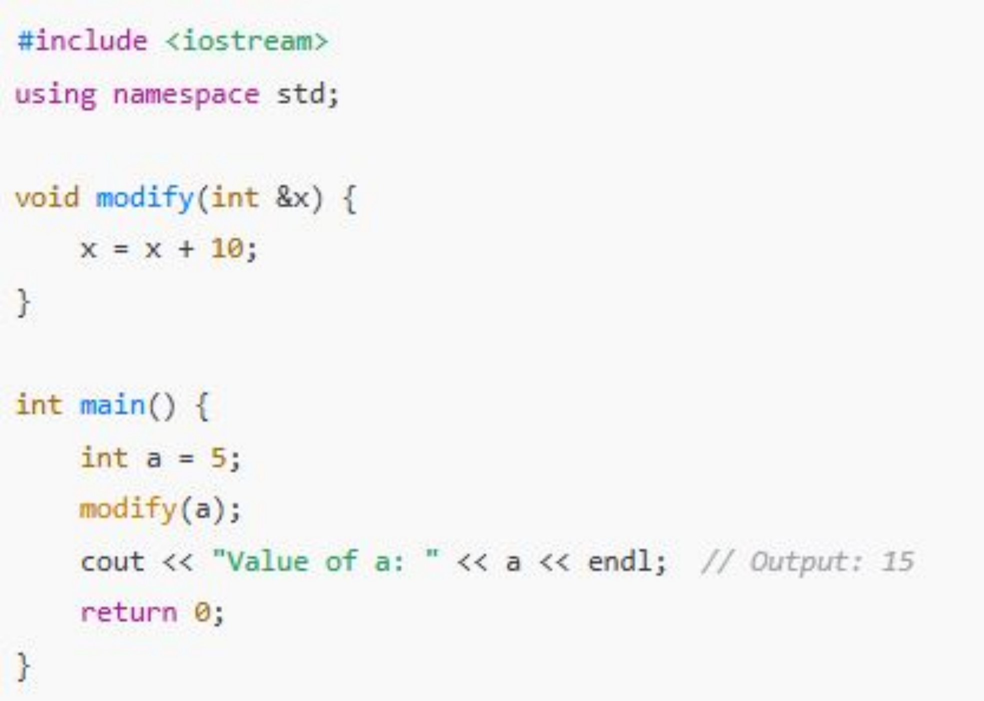
- A **copy** of the actual variable is passed.
- Changes do **not** reflect in the original variable.



```cpp
#include <iostream>
using namespace std;

void modify(int x) {
    x = x + 10;
}

int main() {
    int a = 5;
    modify(a);
    cout << "Value of a: " << a << endl;  // Output: 5
    return 0;
}
```

Reference (alias) to original variable is passed.

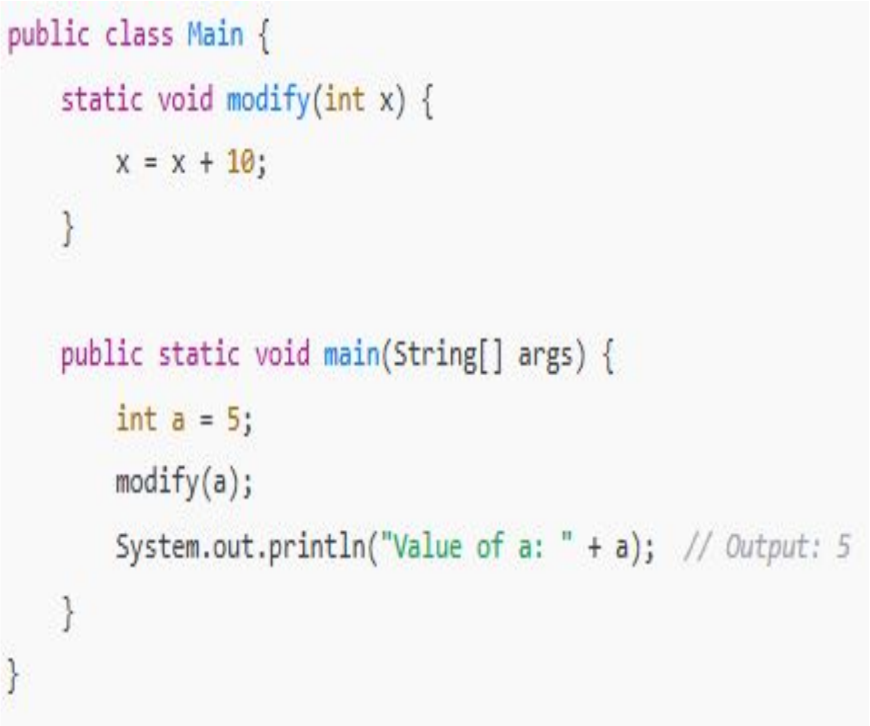
Changes **reflect** in the original variable.



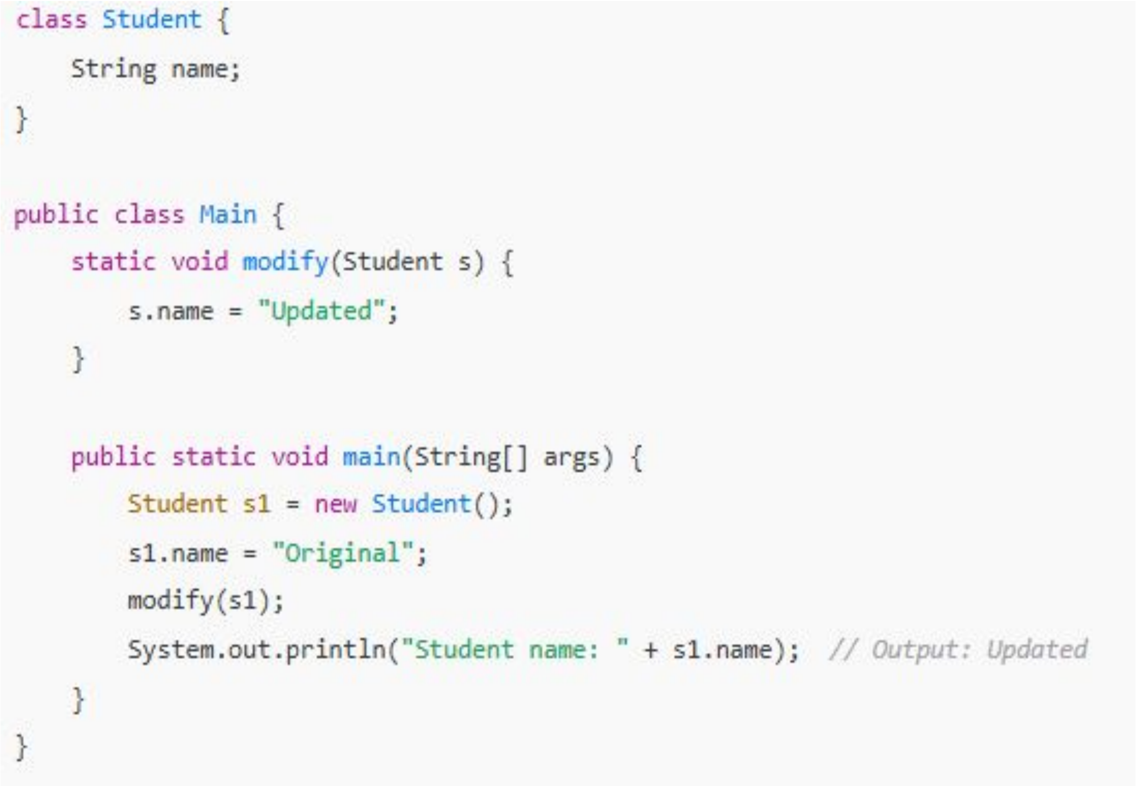
```cpp
#include <iostream>
using namespace std;

void modify(int &x) {
    x = x + 10;
}

int main() {
    int a = 5;
    modify(a);
    cout << "Value of a: " << a << endl;  // Output: 15
    return 0;
}
```

Java: Only Call by Value (but... objects behave like reference)

- Java is **strictly call by value**.
- For **primitive types**, the value is copied → no effect on original.
- For **objects**, the **reference (memory address) is copied**, so original object can be modified.



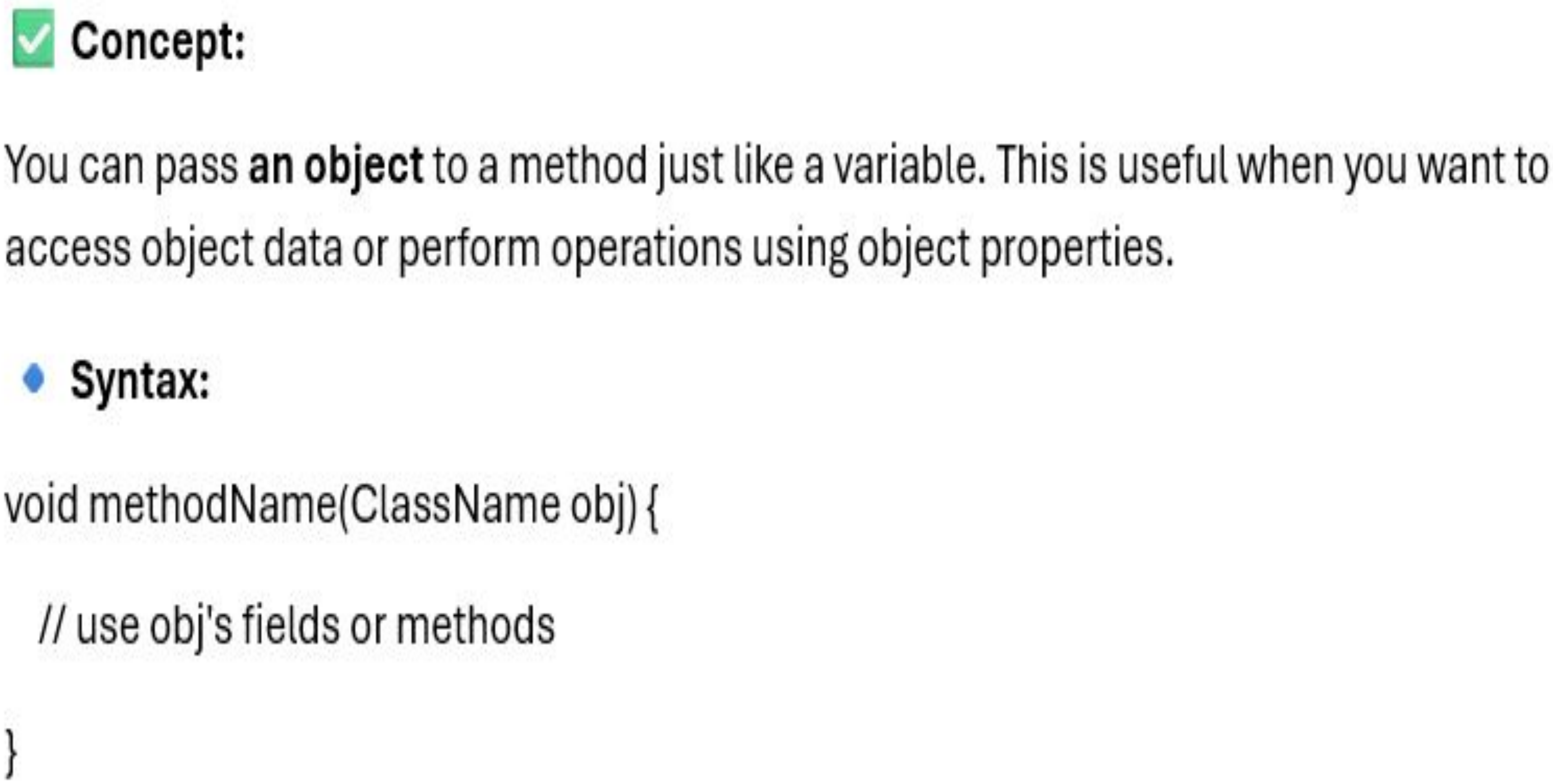
```
public class Main {
    static void modify(int x) {
        x = x + 10;
    }

    public static void main(String[] args) {
        int a = 5;
        modify(a);
        System.out.println("Value of a: " + a);  // Output: 5
    }
}
```



```
class Student {
    String name;
}

public class Main {
    static void modify(Student s) {
        s.name = "Updated";
    }

    public static void main(String[] args) {
        Student s1 = new Student();
        s1.name = "Original";
        modify(s1);
        System.out.println("Student name: " + s1.name);  // Output: Updated
    }
}
```

Objects as parameters
and
Returning objects

objects as parameters



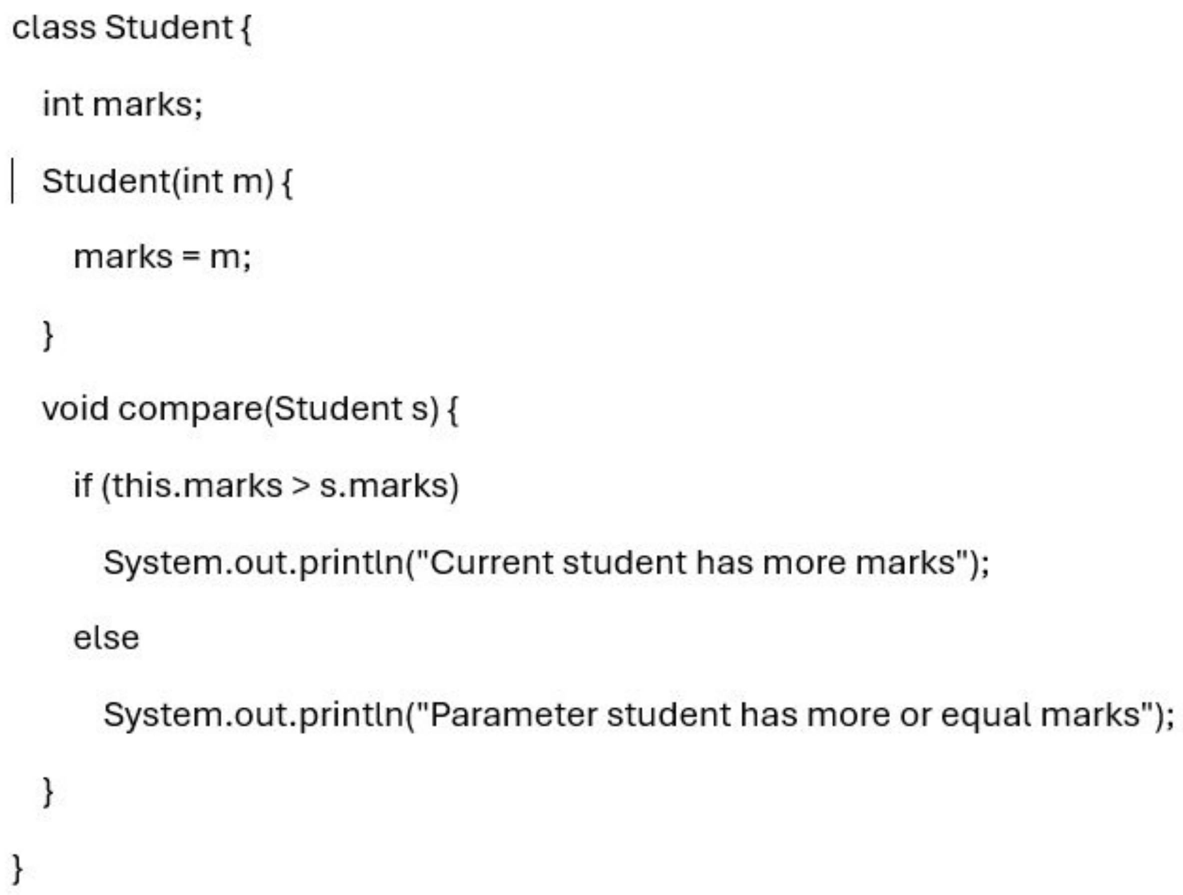
✅ Concept:

You can pass **an object** to a method just like a variable. This is useful when you want to access object data or perform operations using object properties.

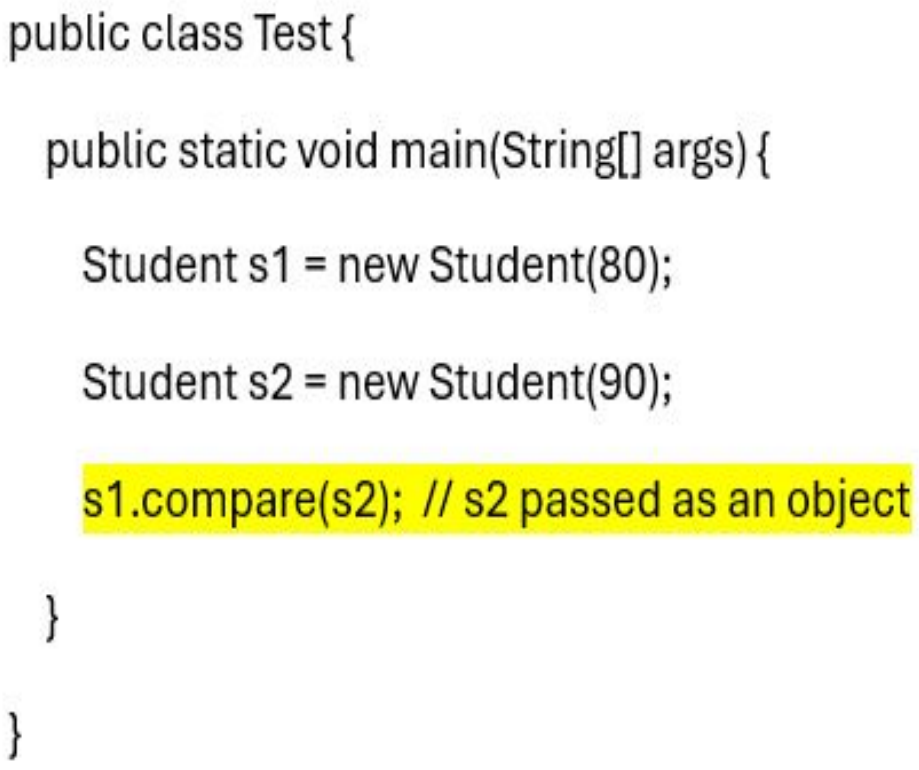
🔹 Syntax:

```
void methodName(ClassName obj) {
   // use obj's fields or methods
}
```

Example : Using Object as a Parameter



```
class Student {
  int marks;
  Student(int m) {
    marks = m;
  }
  void compare(Student s) {
    if (this.marks > s.marks)
      System.out.println("Current student has more marks");
    else
      System.out.println("Parameter student has more or equal marks");
  }
}
```



```
public class Test {
  public static void main(String[] args) {
    Student s1 = new Student(80);
    Student s2 = new Student(90);
    s1.compare(s2);  // s2 passed as an object
  }
}
```

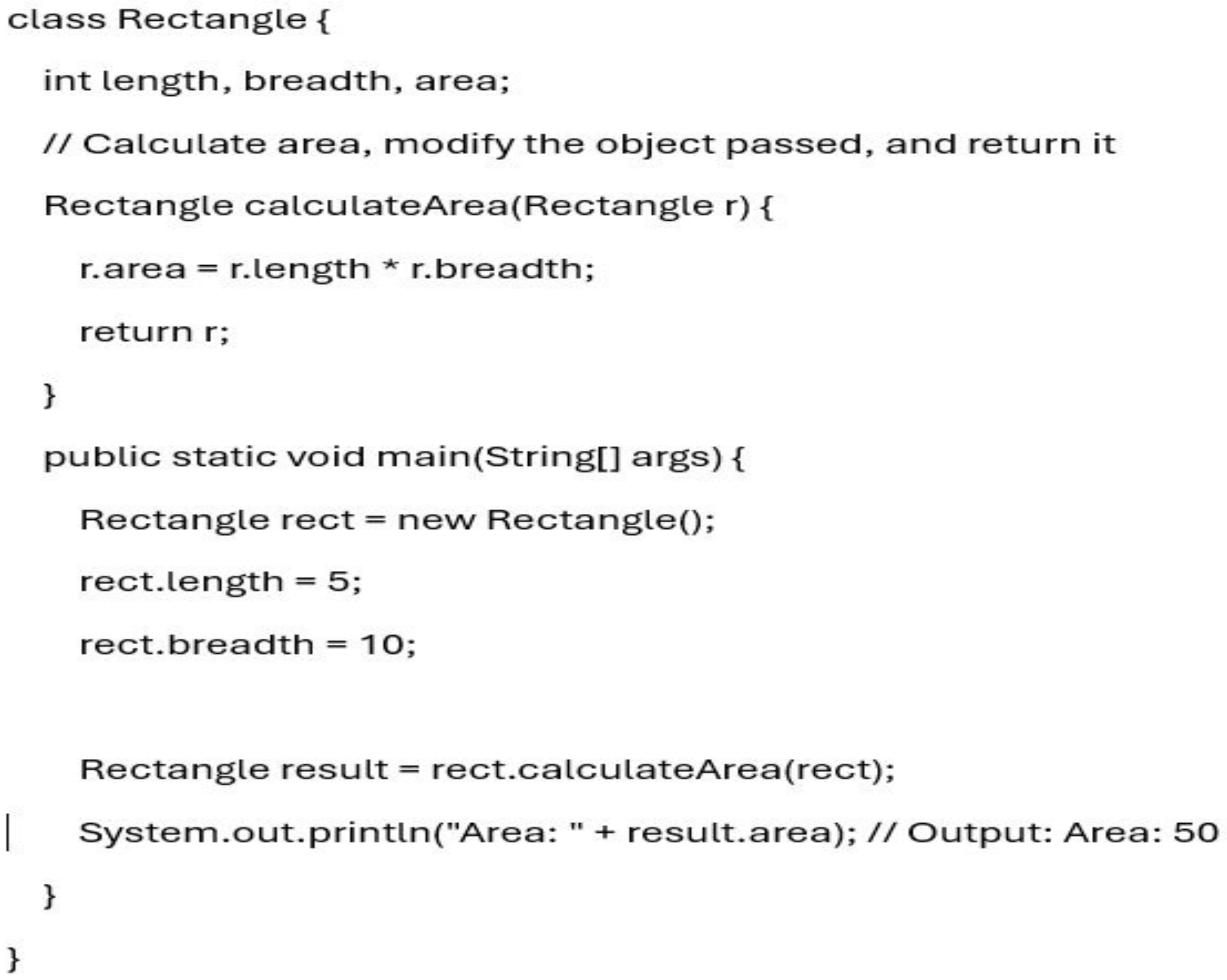
Returning Objects from a Method

✅ Concept:

A method can return an object, allowing dynamic object creation and returning results as object types.

🔹 Syntax:

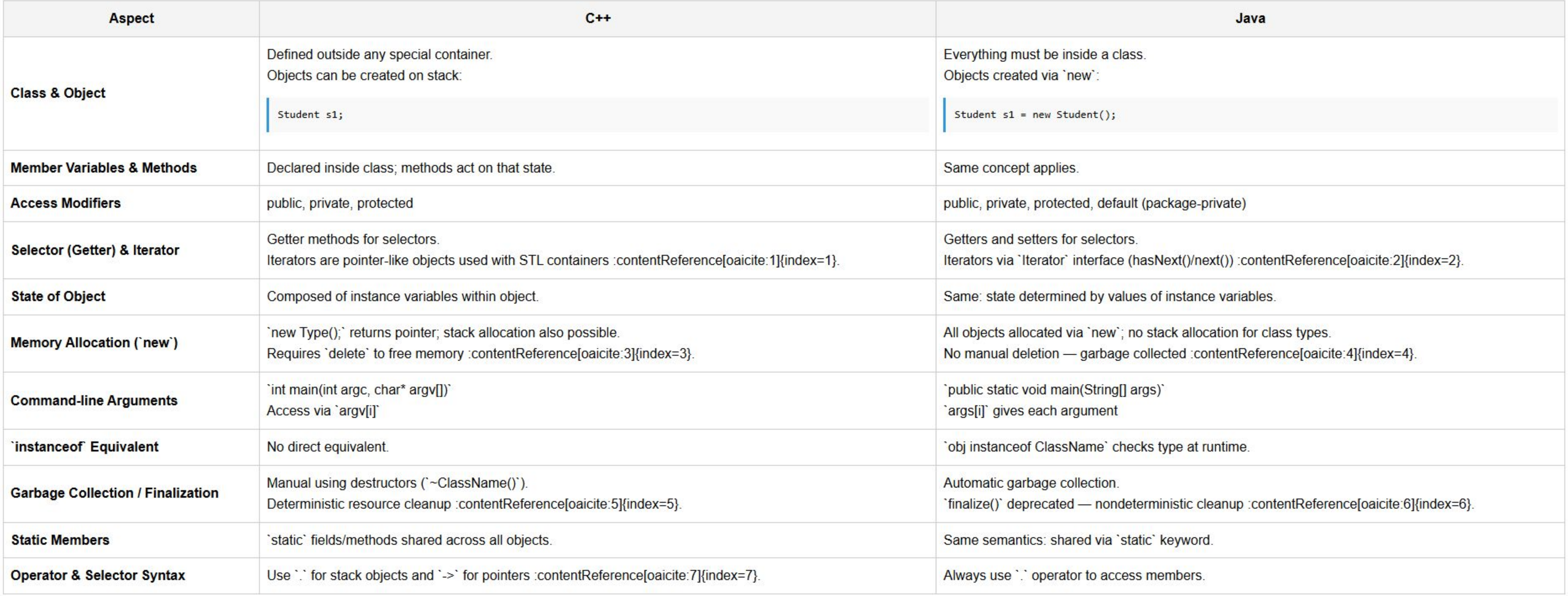
```
ClassName methodName() { return new ClassName(); // or return existing object }
```



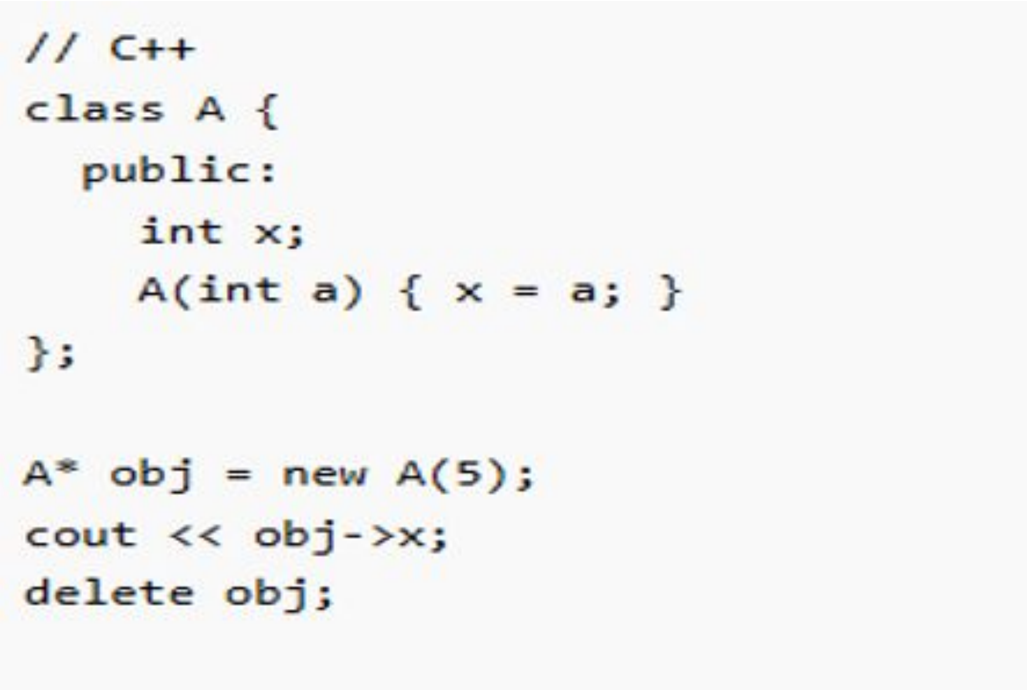
```
class Rectangle {
    int length, breadth, area;
    // Calculate area, modify the object passed, and return it
    Rectangle calculateArea(Rectangle r) {
        r.area = r.length * r.breadth;
        return r;
    }
    public static void main(String[] args) {
        Rectangle rect = new Rectangle();
        rect.length = 5;
        rect.breadth = 10;

        Rectangle result = rect.calculateArea(rect);
        System.out.println("Area: " + result.area); // Output: Area: 50
    }
}
```

https://www.iitk.ac.in/esc101/08Jul/notes.html

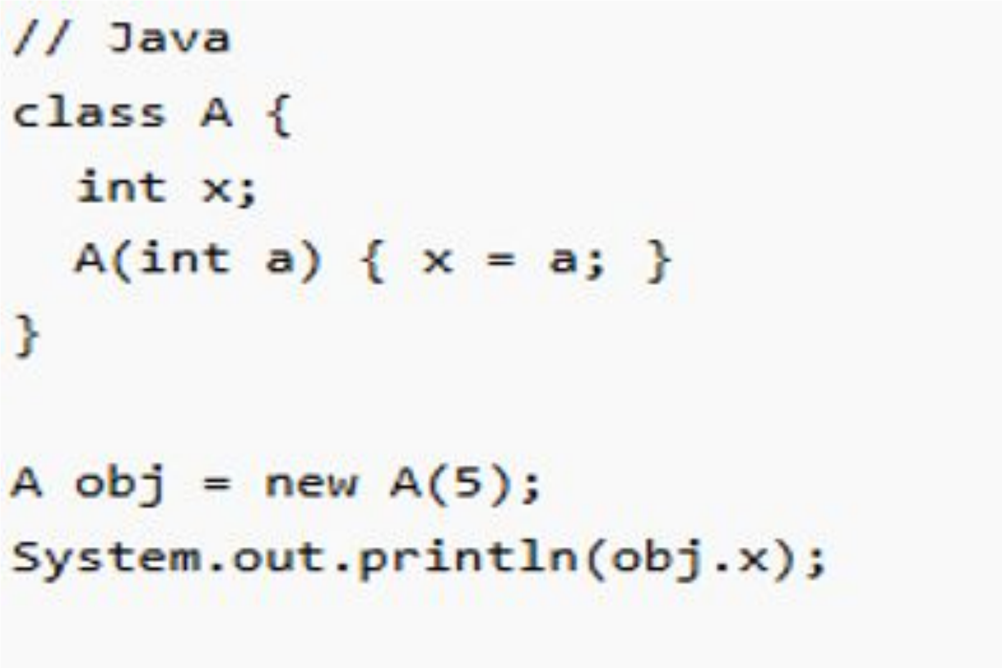
Class, Object & Method: C++ vs Java



Aspect	C++	Java
Class & Object	Defined outside any special container. Objects can be created on stack: `Student s1;`	Everything must be inside a class. Objects created via `new`: `Student s1 = new Student();`
Member Variables & Methods	Declared inside class; methods act on that state.	Same concept applies.
Access Modifiers	public, private, protected	public, private, protected, default (package-private)
Selector (Getter) & Iterator	Getter methods for selectors. Iterators are pointer-like objects used with STL containers :contentReference[oaicite:1]{index=1}.	Getters and setters for selectors. Iterators via `Iterator` interface (hasNext()/next()) :contentReference[oaicite:2]{index=2}.
State of Object	Composed of instance variables within object.	Same: state determined by values of instance variables.
Memory Allocation (`new`)	`new Type();` returns pointer; stack allocation also possible. Requires `delete` to free memory :contentReference[oaicite:3]{index=3}.	All objects allocated via `new`; no stack allocation for class types. No manual deletion — garbage collected :contentReference[oaicite:4]{index=4}.
Command-line Arguments	`int main(int argc, char* argv[])` Access via `argv[i]`	`public static void main(String[] args)` `args[i]` gives each argument
`instanceof` Equivalent	No direct equivalent.	`obj instanceof ClassName` checks type at runtime.
Garbage Collection / Finalization	Manual using destructors (`~ClassName()`). Deterministic resource cleanup :contentReference[oaicite:5]{index=5}.	Automatic garbage collection. `finalize()` deprecated — nondeterministic cleanup :contentReference[oaicite:6]{index=6}.
Static Members	`static` fields/methods shared across all objects.	Same semantics: shared via `static` keyword.
Operator & Selector Syntax	Use `.` for stack objects and `->` for pointers :contentReference[oaicite:7]{index=7}.	Always use `.` operator to access members.



```
// C++
class A {
  public:
    int x;
    A(int a) { x = a; }
};

A* obj = new A(5);
cout << obj->x;
delete obj;
```



```
// Java
class A {
  int x;
  A(int a) { x = a; }
}

A obj = new A(5);
System.out.println(obj.x);
```