Chapter 4.1: Objects and Classes

Object Oriented Programming with Java

Classes and objects | Constructors | Kinds of fields and methods | static members | Access and data hiding | References and this | == and equals | Wrapper classes | record and enum | Inner classes | Kinds of classes

Sercan KÜLCÜ · Giresun University · Computer Engineering

© Sercan Külcü. All rights reserved. · Interactive HTML version of these slides: https://sercankulcu.github.io/files/java/slides/Chapter_04_01_Objects_and_Classes.html

INTRODUCTION

Learning Outcomes

Classes and objects

Explain a class as a blueprint and an object as a concrete instance built from it; write a class with fields and methods.

Creating objects

Use the `new` operator and constructors; know default values and the default constructor.

Instance and class members

Tell instance members from `static` fields and methods; trace the order in which initializer blocks run.

References

Show with a memory diagram that a variable holds a **reference**, not the object; tell apart `this`, `==` and `equals`.

Constructor chaining and access

Chain constructors with `this(...)`; explain the purpose of access modifiers and data hiding.

Special kinds of classes

Recognize wrapper classes, `record`, `enum` and inner classes; design a class from scratch with the `Rational` example.

Object-Oriented Programming

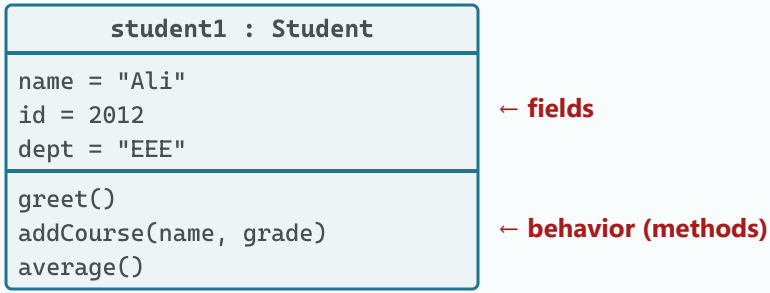
- **Object-oriented programming** (OOP) builds a program out of **interacting objects**.
- **An object** represents a clearly identifiable **entity** in the real world: a student, a table, a circle, a button on screen, even a loan.
- Every object has **state** (its data) and **behavior** (what it can do).

Analogy: a school works through student, teacher and course objects **sending messages** to one another (calling methods).

Object	Fields (state)	Behavior
Student	name, id, dept	enroll(), viewGrades()
Table	width, depth, material	move()
Circle	center, radius	area(), perimeter()
Button	label, color, enabled	click()
Loan	amount, rate, term	installment()

Objects: Fields and Behavior

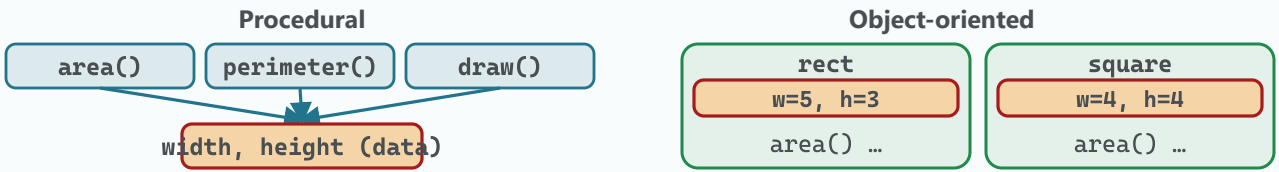
- A class has **members**: **fields**, **methods** and **constructors**.
- Objects **store data**: these pieces of data are called **fields**.
- Objects **perform operations**: these operations are called **behavior** (methods).
- Bundling data with the code that works on it makes a program more **organized**, **reusable** and **intuitive**.



Procedural vs Object-Oriented

	Procedural approach	Object-oriented approach
Basic unit	Function (method)	Object (data + methods)
Data	In separate variables; any function can access it	Inside the object; exposed in a controlled way
Example	`area(width, height)`	`rect.area()`
Large programs	A change affects many functions	A change usually stays inside one class
Reuse	By copying functions	Through classes and inheritance

The `static`-method programs we have written so far were procedural. In Java all code lives in a class; in OOP, classes are used **to create objects**.



Three Core Properties

Encapsulation

An object can **hide** its internal details from other objects. It acts like a closed box, used only through permitted methods.

Inheritance

A class can **inherit** the fields and methods of another class. This enables code reuse and class hierarchies.

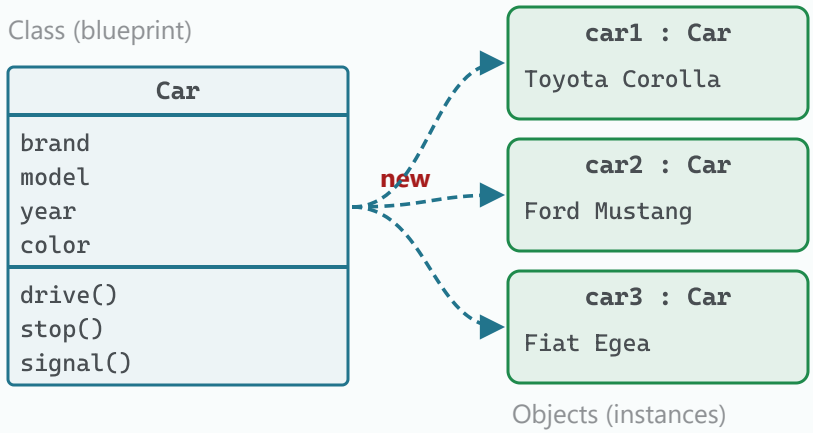
Polymorphism

An object **can take many forms** (types); the same call behaves differently on different objects, giving flexibility and dynamic behavior.

A fourth is often added: **abstraction**, hiding needless detail and showing only the essential features.

Class: The Blueprint of an Object

- **A class** is the **blueprint** of an object; one class can produce many objects carrying different data.
- A class specifies the **data** (fields) an object holds and the **operations** (methods) it can perform.
- An object is an **instance** of a class: its concrete form in memory.
- Classes let us **model** real-world entities: a `Car` class can represent any automobile; each make and model is an object of that class.



Analogy: the class is a cookie cutter; objects are the cookies cut with it.

Example: The Car Class

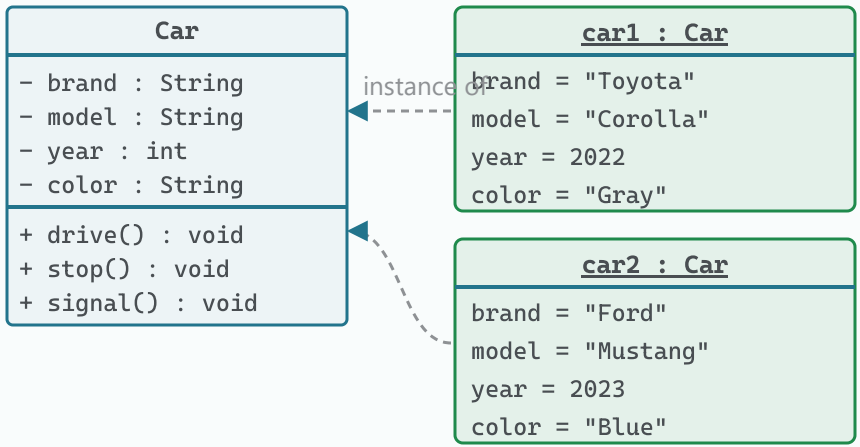
Fields (data)

Brand, Model, Year, Color

Methods (operations)

Drive, Stop, Signal

Two objects of the same class have the same **field names** but different **values**.



UML class diagram: name / fields / methods; `-` private, `+` public. Object names are underlined.

Defining a Class

```java
public class Car {
  // Fields
  String brand;
  String model;
  int year;
  String color;

  // Methods
  void drive() {
    System.out.println(brand + " is moving.");
  }

  void stop() {
    System.out.println(brand + " stopped.");
  }

  void signal() {
    System.out.println(brand + " is signaling.");
  }
}
```

- The `class` keyword and the class name come first; the body goes between `{ }`.
- Class names start with a **capital letter** (`Car` , `BankAccount`); a `public` class must be in a **file of the same name** (`Car.java`).
- Fields are declared in the class body, **outside** the methods; every method can use them directly.
- The methods have **no** `static` : each belongs to a particular car (object).
- This class has no `main` ; it does not run on its own; another class uses it.

Using a Class

```
public class CarTest {
  public static void main(String[] args) {
    Car car1 = new Car();
    car1.brand = "Toyota";
    car1.model = "Corolla";
    car1.year = 2022;
    car1.color = "Gray";

    Car car2 = new Car();
    car2.brand = "Ford";
    car2.model = "Mustang";
    car2.year = 2023;
    car2.color = "Blue";

    car1.drive();
    car2.signal();
    car1.stop();
  }
}
```

OUTPUT

```
Toyota is moving.
Ford is signaling.
Toyota stopped.
```

- `new Car()` creates an object; `car1` refers to it.
- `car1.brand = "Toyota";` : the dot (`.`) accesses a field of the object.
- `car1.drive()` : the method works with **car1's** data; `car2.signal()` uses car2's.
- Changing fields directly from outside like this is a bad habit; the right way is to use **constructors** and **accessor methods**.

Members of a Class

Fields

Represent the **data** objects can hold. Example: a `Student` class with `name` and `surname`.

Methods

Define the **operations** objects can perform. Example: a `Dog` class with `bark()` and `run()`.

Access level

Each member can be `private`, `protected`, default (nothing written) or `public`: this decides who may use it.

Ownership

Each member can **belong to the object** (instance, the default) or, with `static`, **belong to the class**.

Constructors are also written inside the class, but they run only when an object is created and are technically a separate kind of member, not methods. A class can also contain other classes (inner classes).

Access Modifiers and Data Hiding

- **Access modifiers** (`public` , `protected` , `private` ; package access if none is written) specify who can reach a member from **outside** the class.
- Rule of thumb: fields `private` , methods offered to others `public` .
- A field's value is read by an **accessor** (getter, `get...`) and changed by a **mutator** (setter, `set...`); the setter rejects invalid values.
- This arrangement is **data hiding**: only the class's own methods access its fields directly. It is the basis of encapsulation.

```
public class BankAccount {
  private double balance;   // hidden field
  public double getBalance() {
    return balance;
  }
  public void setBalance(double b) {
    if (b >= 0) balance = b;  // reject invalid
  }
}
// in another class:
acct.balance = -500;
```

```
error: balance has private access in BankAccount
```

Constructors

- **A constructor** is a **special method**; `new` calls it **automatically** when creating an object.
- Its job is to give the fields their **initial values** and do any other setup.
- Its name **matches the class name** (including case).
- **It has no return type**, not even `void`; it cannot return a value.
- Usually declared `public`; it can also be `private` to restrict object creation (e.g. the singleton pattern).
- A class can have **more than one** constructor (with different parameters).

```
public class Student {
  private String name;
  private int id;

  // Constructor: class name, no return type
  public Student(String name, int id) {
    this.name = name;
    this.id = id;
  }
}

// Usage:
Student s = new Student("Ali", 2012);
```

How Is an Object Created?

```
ClassName objectName = new ClassName(arguments);

Car corolla = new Car("Toyota", "Corolla", 2022);
Scanner in = new Scanner(System.in);
Book book = new Book();
```

```
import java.util.Scanner;
public class CreateDemo {
  public static void main(String[] args) {
    Scanner in = new Scanner(System.in);
    StringBuilder text = new StringBuilder("Java");
    String title = "Objects";           // shortcut
    String author = new String("Sercan");  // long way
    text.append(" 21");
    System.out.println(text + " / " + title);
    System.out.println(author.length());
    System.out.println(in.nextInt() * 2);
  }
}
```

Shortcut: `String title = "Objects";` creates a String object, whereas `String title;` only declares a reference.

1. `new` reserves **space** on the heap for the object; fields get default values.
2. **A constructor** is called and sets initial values.
3. `new` evaluates to the object's **reference**, which is assigned to the variable.

OUTPUT

```
(input: 21)
Java 21 / Objects
6
42
```

Constructor Example: Car

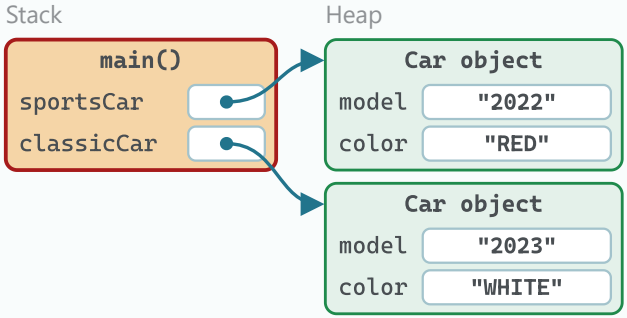
```java
public class Car {
  private String model;
  private String color;

  public Car(String model, String color) {
    this.model = model;
    this.color = color;
  }

  public void printInfo() {
    System.out.println(model + " - " + color);
  }

}
```

```java
public static void main(String[] args) {
  Car sportsCar = new Car("2022", "RED");
  Car classicCar = new Car("2023", "WHITE");
  sportsCar.printInfo();
  classicCar.printInfo();
}
```



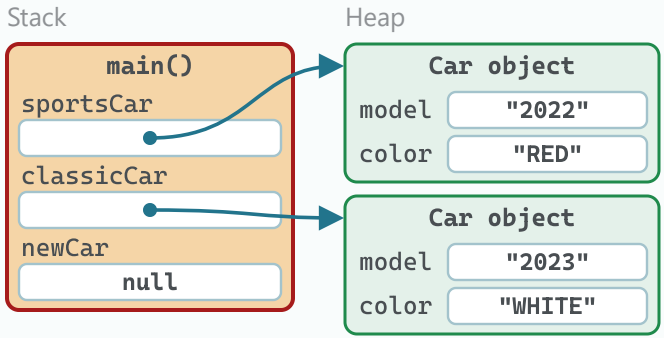
OUTPUT

```
2022 - RED
2023 - WHITE
```

References

- An object cannot be used **directly**; it is reached only through a **reference**. Variables store the object's reference, not the object itself.
- A reference can be thought of as an **address** telling where the object is in memory.
- `int`, `double`, `boolean` and other **primitive** variables hold the value itself; class-type variables (`Car`, `String`, `Scanner`) hold a **reference**.
- A reference that points to no object has the value `null`.

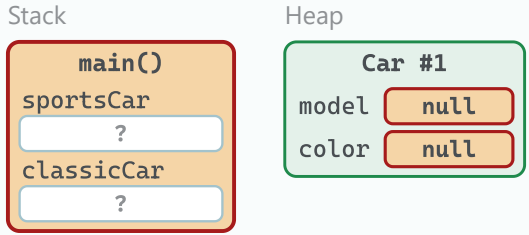
Analogy: the variable is a **TV remote**; the object is the TV. Copying the remote does not create a second TV.



Local variables live on the stack, objects on the heap; arrow = reference.

new Car(...): Step by Step

```
1  public class Car {
2    private String model;
3    private String color;
4
5    public Car(String model, String color) {
6      this.model = model;
7      this.color = color;
8    }
9
10   public void printInfo() {
11     System.out.println(model + " - " + color);
12   }
13
14   public static void main(String[] args) {
▶      Car sportsCar = new Car("2022", "RED");
16     Car classicCar = new Car("2023", "WHITE");
17     sportsCar.printInfo();
18     classicCar.printInfo();
19   }
20 }
```



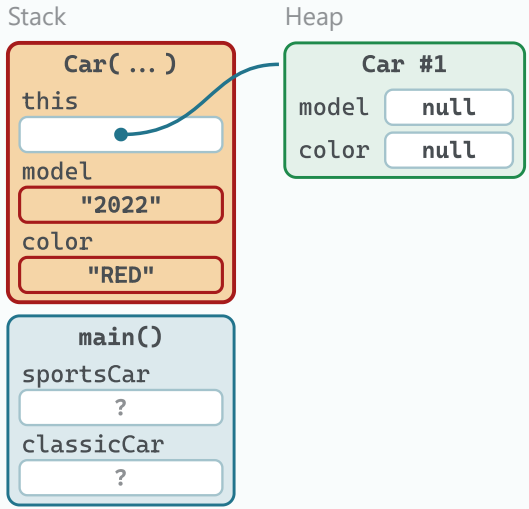
`new` allocates a new object on the heap; `String` fields first get the default value `null`.

Step 1 / 8

OUTPUT

new Car(...): Step by Step

```
1  public class Car {
2    private String model;
3    private String color;
4
5    public Car(String model, String color) {
6      this.model = model;
7      this.color = color;
8    }
9
10   public void printInfo() {
11     System.out.println(model + " - " + color);
12   }
13
14   public static void main(String[] args) {
15     Car sportsCar = new Car("2022", "RED");
16     Car classicCar = new Car("2023", "WHITE");
17     sportsCar.printInfo();
18     classicCar.printInfo();
19   }
20 }
```



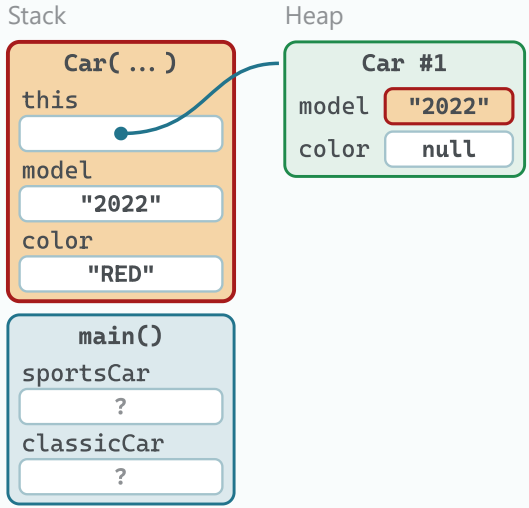
The constructor is called: parameters receive copies of the arguments; `this` refers to the new object.

Step 2 / 8

OUTPUT

new Car(...): Step by Step

```
1  public class Car {
2    private String model;
3    private String color;
4
5    public Car(String model, String color) {
▶      this.model = model;
7      this.color = color;
8    }
9
10   public void printInfo() {
11     System.out.println(model + " - " + color);
12   }
13
14   public static void main(String[] args) {
15     Car sportsCar = new Car("2022", "RED");
16     Car classicCar = new Car("2023", "WHITE");
17     sportsCar.printInfo();
18     classicCar.printInfo();
19   }
20 }
```



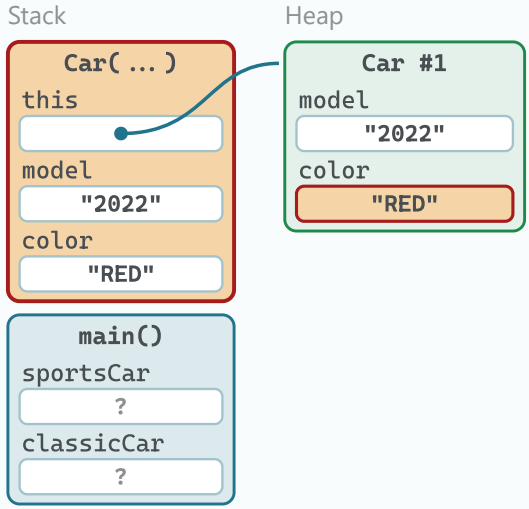
`this.model = model` : the left side is the object's field, the right side the parameter.

Step 3 / 8

OUTPUT

new Car(...): Step by Step

```
1  public class Car {
2    private String model;
3    private String color;
4
5    public Car(String model, String color) {
6      this.model = model;
▶      this.color = color;
8    }
9
10   public void printInfo() {
11     System.out.println(model + " - " + color);
12   }
13
14   public static void main(String[] args) {
15     Car sportsCar = new Car("2022", "RED");
16     Car classicCar = new Car("2023", "WHITE");
17     sportsCar.printInfo();
18     classicCar.printInfo();
19   }
20 }
```



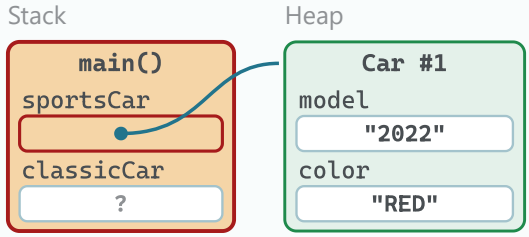
`this.color = color` : the object is now ready.

Step 4 / 8

OUTPUT

new Car(...): Step by Step

```
1  public class Car {
2    private String model;
3    private String color;
4
5    public Car(String model, String color) {
6      this.model = model;
7      this.color = color;
8    }
9
10   public void printInfo() {
11     System.out.println(model + " - " + color);
12   }
13
14   public static void main(String[] args) {
▶      Car sportsCar = new Car("2022", "RED");
16     Car classicCar = new Car("2023", "WHITE");
17     sportsCar.printInfo();
18     classicCar.printInfo();
19   }
20 }
```



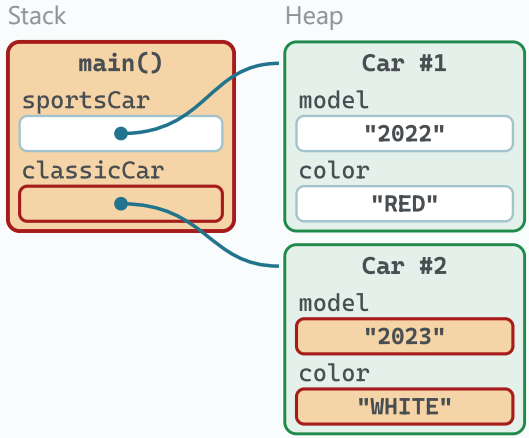
The constructor returns and its frame is removed. The `new` expression yields a **reference**, which is stored in `sportsCar`.

Step 5 / 8

OUTPUT

new Car(...): Step by Step

```
1  public class Car {
2    private String model;
3    private String color;
4
5    public Car(String model, String color) {
6      this.model = model;
7      this.color = color;
8    }
9
10   public void printInfo() {
11     System.out.println(model + " - " + color);
12   }
13
14   public static void main(String[] args) {
15     Car sportsCar = new Car("2022", "RED");
▶      Car classicCar = new Car("2023", "WHITE");
17     sportsCar.printInfo();
18     classicCar.printInfo();
19   }
20 }
```



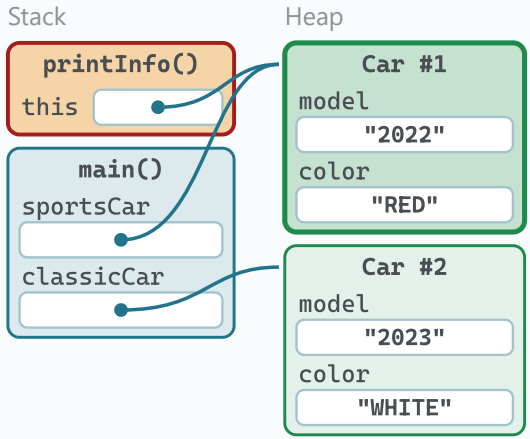
Same steps again: a **separate** object is created, with its own `model` and `color` copies.

Step 6 / 8

OUTPUT

new Car(...): Step by Step

```
1  public class Car {
2    private String model;
3    private String color;
4
5    public Car(String model, String color) {
6      this.model = model;
7      this.color = color;
8    }
9
10   public void printInfo() {
▶      System.out.println(model + " - " + color);
12   }
13
14   public static void main(String[] args) {
15     Car sportsCar = new Car("2022", "RED");
16     Car classicCar = new Car("2023", "WHITE");
▶      sportsCar.printInfo();
18     classicCar.printInfo();
19   }
20 }
```



`sportsCar.printInfo()` : inside the method, `this` = #1, so #1's fields are printed.

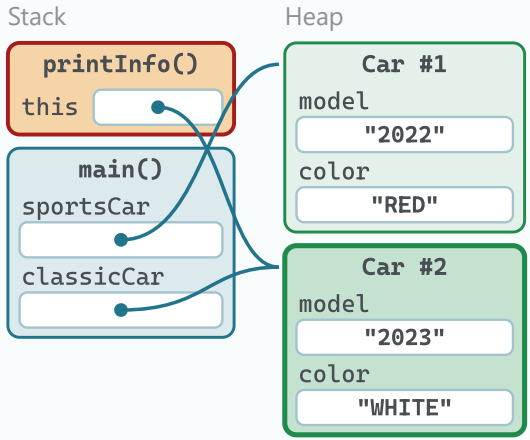
Step 7 / 8

OUTPUT

```
2022 - RED
```

new Car(...): Step by Step

```
1  public class Car {
2    private String model;
3    private String color;
4
5    public Car(String model, String color) {
6      this.model = model;
7      this.color = color;
8    }
9
10   public void printInfo() {
▶      System.out.println(model + " - " + color);
12   }
13
14   public static void main(String[] args) {
15     Car sportsCar = new Car("2022", "RED");
16     Car classicCar = new Car("2023", "WHITE");
17     sportsCar.printInfo();
▶      classicCar.printInfo();
19   }
20 }
```



Same method, but now `this` = #2. The code is shared; the data is separate per object.

Step 8 / 8

OUTPUT

```
2022 - RED
2023 - WHITE
```

Default Constructor

- If a class declares no constructor, the compiler adds the **default constructor**: no parameters, empty body. Creating an object always calls some constructor.
- Uninitialized **fields** get a default value (local variables do not):

Field type	Default value
`byte short int long`	`0`
`float double`	`0.0`
`boolean`	`false`
`char`	`'\u0000'` (code 0)
Reference (`String`, arrays, objects)	`null`

```
class Point3 {
  int x;
  double weight;
  boolean visible;
  char letter;
  String name;
}
public class Defaults {
  public static void main(String[] args) {
    Point3 p = new Point3();   // default constructor
    System.out.println(p.x + " " + p.weight);
    System.out.println(p.visible + " " + p.name);
    System.out.println((int) p.letter);
  }
}
```

OUTPUT

```
0 0.0
false null
0
```

No-Arg and Parameterized Constructors

```java
public class Vehicle {
  private String brand;
  private String model;
  private int year;
  public Vehicle() {                    // no-arg constructor
    brand = "Unknown";
    model = "Unknown";
    year = 0;
  }
  public Vehicle(String brand, String model, int year) {
    this.brand = brand;
    this.model = model;
    this.year = year;
  }
  public void printInfo() {
    System.out.println(brand + " " + model + " " + year);
  }
  public static void main(String[] args) {
    Vehicle v1 = new Vehicle();
    Vehicle v2 = new Vehicle("Renault", "Clio", 2021);
    v1.printInfo();
    v2.printInfo();
  }
}
```

OUTPUT

```
Unknown Unknown 0
Renault Clio 2021
```

- `Vehicle()` is our own **no-arg** constructor; it sets the fields to meaningful initial values.
- The two constructors share a name but differ in parameters: constructor **overloading**. The arguments decide which one runs.
- Once a class has **any** constructor, the compiler no longer adds the default one. If only a parameterized constructor exists, `new Vehicle()` does not compile: `constructor Vehicle in class Vehicle cannot be applied to given types`.

Fields and Their Kinds

Fields are data variables declared inside the class (outside any method). There are two basic kinds:

	Instance field	**Class field (static)**
Declaration	`private String name;`	`private static int accountCount;`
Copies	Each object has a **separate** copy	A **single** copy in the class, shared by all objects
Created when?	With every `new`	Once, when the class is loaded
Access	`obj.name`	`ClassName.accountCount`
Example	`Student` : name, surname	`BankAccount` : accountCount

- **Constant** (`final`) fields get a value once and cannot change afterwards. Class constants are usually `static final` and written in UPPER_CASE: `Math.PI` .
- **Static initializers** (`static { }`) run once, when the class loads; **instance initializers** (`{ }`) run every time an object is created.

static Counter: Step by Step

```
1  public class BankAccount {
2    private String owner;
▶    private static int accountCount = 0;
4
5    public BankAccount(String owner) {
6      this.owner = owner;
7      accountCount++;
8    }
9
10   public static int getAccountCount() {
11     return accountCount;
12   }
13
14   public static void main(String[] args) {
15     BankAccount h1 = new BankAccount("Ali");
16     BankAccount h2 = new BankAccount("Ayşe");
17     BankAccount h3 = new BankAccount("Can");
18     System.out.println(h2.owner);
19     int count = BankAccount.getAccountCount();
20     System.out.println(count);
21   }
22 }
```



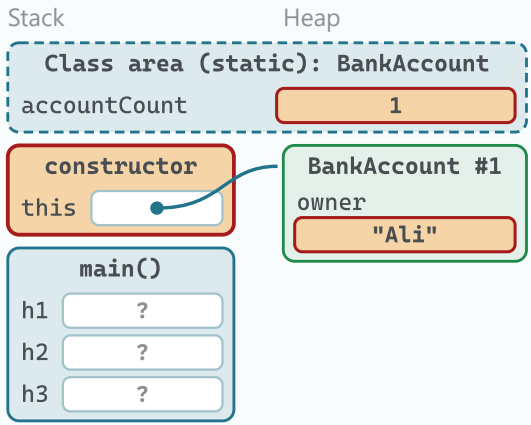
When the class loads, the `static` field is created **once** and set to 0. No objects exist yet.

Step 1 / 8

OUTPUT

static Counter: Step by Step

```
1  public class BankAccount {
2    private String owner;
3    private static int accountCount = 0;
4
▶    public BankAccount(String owner) {
▶      this.owner = owner;
▶      accountCount++;
8    }
9
10   public static int getAccountCount() {
11     return accountCount;
12   }
13
14   public static void main(String[] args) {
15     BankAccount h1 = new BankAccount("Ali");
16     BankAccount h2 = new BankAccount("Ayşe");
17     BankAccount h3 = new BankAccount("Can");
18     System.out.println(h2.owner);
19     int count = BankAccount.getAccountCount();
20     System.out.println(count);
21   }
22 }
```



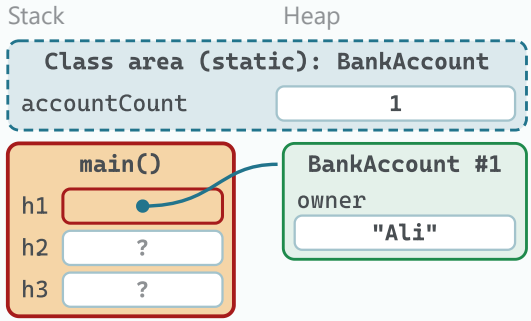
First object: its own `owner` copy becomes "Ali"; the shared counter goes up to 1.

Step 2 / 8

OUTPUT

static Counter: Step by Step

```
1  public class BankAccount {
2    private String owner;
3    private static int accountCount = 0;
4
5    public BankAccount(String owner) {
6      this.owner = owner;
7      accountCount++;
8    }
9
10   public static int getAccountCount() {
11     return accountCount;
12   }
13
14   public static void main(String[] args) {
▶      BankAccount h1 = new BankAccount("Ali");
16     BankAccount h2 = new BankAccount("Ayşe");
17     BankAccount h3 = new BankAccount("Can");
18     System.out.println(h2.owner);
19     int count = BankAccount.getAccountCount();
20     System.out.println(count);
21   }
22 }
```



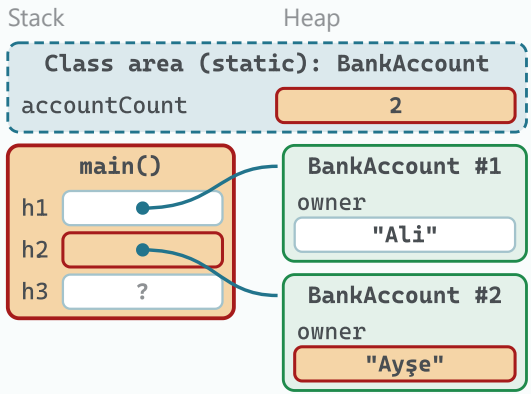
`h1` refers to the first object.

Step 3 / 8

OUTPUT

static Counter: Step by Step

```
1  public class BankAccount {
2    private String owner;
3    private static int accountCount = 0;
4
5    public BankAccount(String owner) {
6      this.owner = owner;
7      accountCount++;
8    }
9
10   public static int getAccountCount() {
11     return accountCount;
12   }
13
14   public static void main(String[] args) {
15     BankAccount h1 = new BankAccount("Ali");
▶      BankAccount h2 = new BankAccount("Ayşe");
17     BankAccount h3 = new BankAccount("Can");
18     System.out.println(h2.owner);
19     int count = BankAccount.getAccountCount();
20     System.out.println(count);
21   }
22 }
```



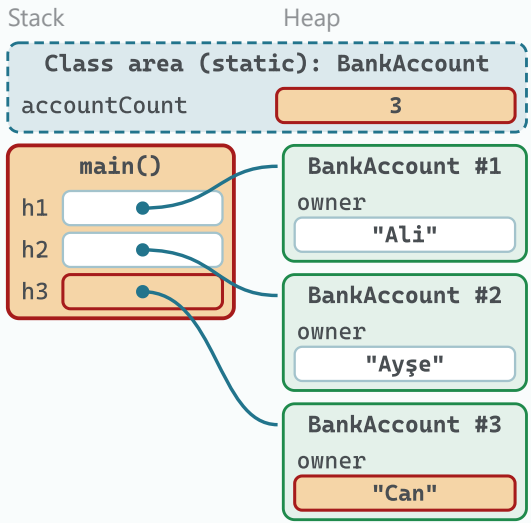
The second object has its own **separate** `owner` field, but the counter is the **same** field: 2.

Step 4 / 8

OUTPUT

static Counter: Step by Step

```
1  public class BankAccount {
2    private String owner;
3    private static int accountCount = 0;
4
5    public BankAccount(String owner) {
6      this.owner = owner;
7      accountCount++;
8    }
9
10   public static int getAccountCount() {
11     return accountCount;
12   }
13
14   public static void main(String[] args) {
15     BankAccount h1 = new BankAccount("Ali");
16     BankAccount h2 = new BankAccount("Ayşe");
▶      BankAccount h3 = new BankAccount("Can");
18     System.out.println(h2.owner);
19     int count = BankAccount.getAccountCount();
20     System.out.println(count);
21   }
22 }
```



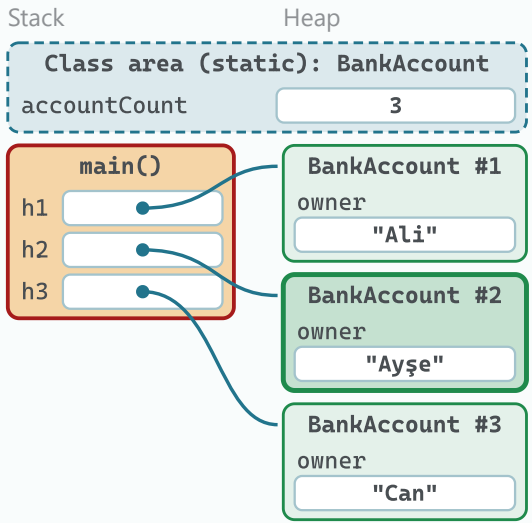
Three objects, three `owner` fields, but a single `accountCount` = 3.

Step 5 / 8

OUTPUT

static Counter: Step by Step

```
1  public class BankAccount {
2    private String owner;
3    private static int accountCount = 0;
4
5    public BankAccount(String owner) {
6      this.owner = owner;
7      accountCount++;
8    }
9
10   public static int getAccountCount() {
11     return accountCount;
12   }
13
14   public static void main(String[] args) {
15     BankAccount h1 = new BankAccount("Ali");
16     BankAccount h2 = new BankAccount("Ayşe");
17     BankAccount h3 = new BankAccount("Can");
▶      System.out.println(h2.owner);
19     int count = BankAccount.getAccountCount();
20     System.out.println(count);
21   }
22 }
```



`h2.owner` : an instance field is read through the object.

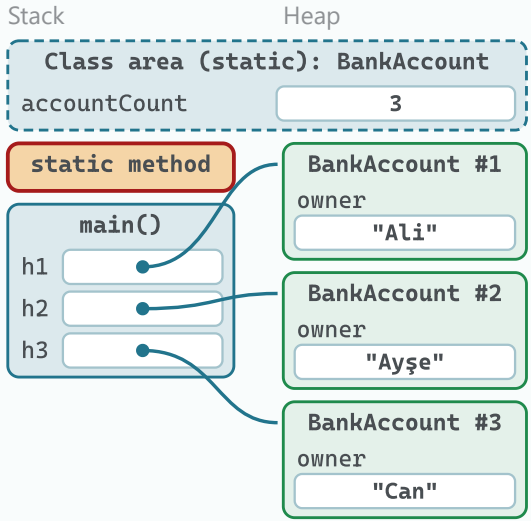
Step 6 / 8

OUTPUT

```
Ayşe
```

static Counter: Step by Step

```
1  public class BankAccount {
2    private String owner;
3    private static int accountCount = 0;
4
5    public BankAccount(String owner) {
6      this.owner = owner;
7      accountCount++;
8    }
9
▶    public static int getAccountCount() {
▶      return accountCount;
12   }
13
14   public static void main(String[] args) {
15     BankAccount h1 = new BankAccount("Ali");
16     BankAccount h2 = new BankAccount("Ayşe");
17     BankAccount h3 = new BankAccount("Can");
18     System.out.println(h2.owner);
▶      int count = BankAccount.getAccountCount();
20     System.out.println(count);
21   }
22 }
```



`static` methods are called **via the class name**; there is no `this` here, so it reads only the static field.

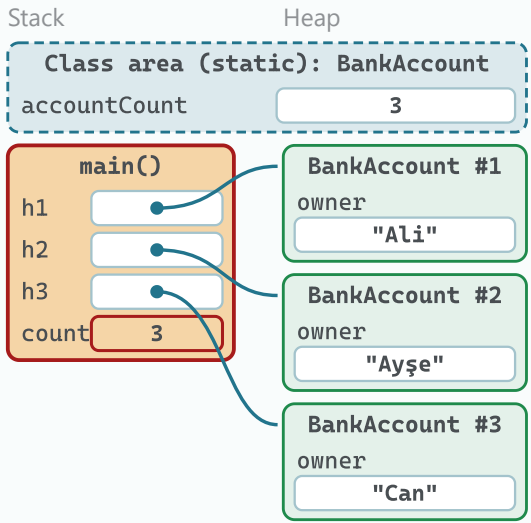
Step 7 / 8

OUTPUT

```
Ayşe
```

static Counter: Step by Step

```
1  public class BankAccount {
2    private String owner;
3    private static int accountCount = 0;
4
5    public BankAccount(String owner) {
6      this.owner = owner;
7      accountCount++;
8    }
9
10   public static int getAccountCount() {
11     return accountCount;
12   }
13
14   public static void main(String[] args) {
15     BankAccount h1 = new BankAccount("Ali");
16     BankAccount h2 = new BankAccount("Ayşe");
17     BankAccount h3 = new BankAccount("Can");
18     System.out.println(h2.owner);
19     int count = BankAccount.getAccountCount();
▶      System.out.println(count);
21   }
22 }
```



The returned 3 goes into `count` and is printed.

Step 8 / 8

OUTPUT

```
Ayşe
3
```

Field Declarations

```
public class Book {
  // Instance fields
  private String title;
  private String author;
  private int pageCount;
  // Class field
  private static int bookCount;
  // a final field (constant)
  public static final double PI = 3.14159265359;
}
```

```
public class Fields {
  public int a;                          // public
  int b = 1, c = 2;                      // default (package)
  private double x;                      // private
  private static int xx;                 // static, private
  public static int y;                   // static, public
  public final int CONST1 = 5;           // cannot change
  private static final int CONST2 = 6;   // static constant
}
```

- Syntax: `[access] [static] [final] type name [= initValue];`
- Several fields on one line: `int b = 1, c = 2;`
- `PI` is defined here as an example; Java has a ready-made constant: `Math.PI`.
- `final` fields cannot be reassigned; doing so is a compile error: `error: cannot assign a value to final variable CONST1`

Initializer Blocks

Static initializer

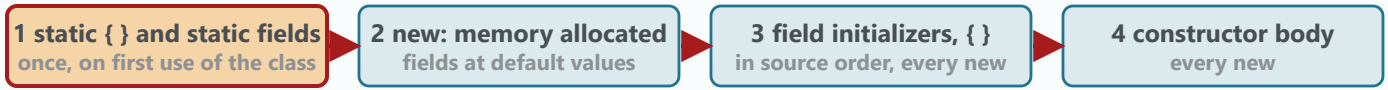
The `static { ... }` block runs when the class is **first loaded**, and only **once**. It initializes complex static fields.

Instance initializer

The `{ ... }` block runs **every time an object is created**, just **before** the constructor body. It holds work common to all constructors.

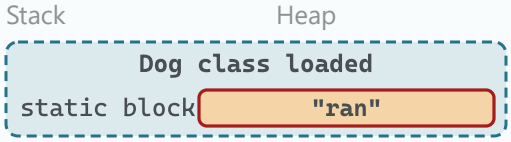
Order when creating an object: **(1)** on first use of the class, static fields and `static` blocks → **(2)** `new` allocates memory, fields get default values → **(3)** field initializers and instance initializer blocks (in source order) → **(4)** the constructor body.

`Dog` has both. Note: the value `bark()` returns is not printed; with `System.out.println(d2.bark());` the last line would be "Gece woof woof!".



Dog: Initializer Order

```
1  public class Dog {
2    String name;                  // field
3    Dog(String name) {            // constructor
4      this.name = name;
5      System.out.println(name);
6    }
7    String bark() {               // instance method
8      return name + " woof woof!";
9    }
10   static {                      // static initializer
▶      System.out.println("Dogs!");
12   }
13   {                             // instance initializer
14     System.out.println("Creating a dog.");
15   }
16   public static void main(String... strings) {
17     var d1 = new Dog("Kara");
18     var d2 = new Dog("Gece");
19     d2.bark();
20   }
21 }
```



`main` runs only after the class is loaded: the `static` block runs **once**. (`var`, Java 10+: infers the type from the right side; `String...` = `String[]`.)

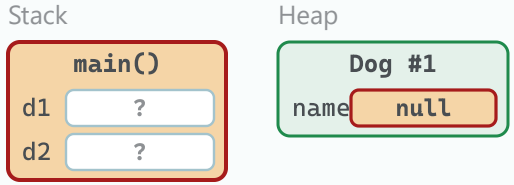
Step 1 / 7

OUTPUT

```
Dogs!
```

Dog: Initializer Order

```
1  public class Dog {
2    String name;                  // field
3    Dog(String name) {           // constructor
4      this.name = name;
5      System.out.println(name);
6    }
7    String bark() {              // instance method
8      return name + " woof woof!";
9    }
10   static {                     // static initializer
11     System.out.println("Dogs!");
12   }
13   {                            // instance initializer
14     System.out.println("Creating a dog.");
15   }
16   public static void main(String... strings) {
▶      var d1 = new Dog("Kara");
18     var d2 = new Dog("Gece");
19     d2.bark();
20   }
21 }
```



`new Dog("Kara")` : space is allocated for the object, `name` = null.

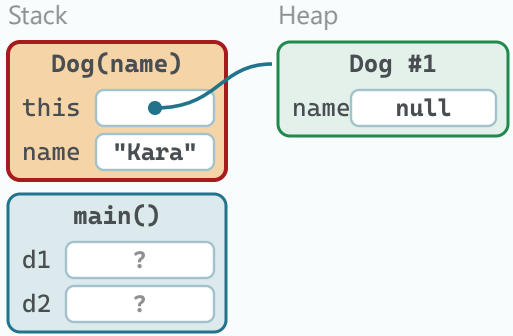
Step 2 / 7

OUTPUT

```
Dogs!
```

Dog: Initializer Order

```
1  public class Dog {
2    String name;                  // field
3    Dog(String name) {           // constructor
4      this.name = name;
5      System.out.println(name);
6    }
7    String bark() {              // instance method
8      return name + " woof woof!";
9    }
10   static {                     // static initializer
11     System.out.println("Dogs!");
12   }
13   {                            // instance initializer
▶      System.out.println("Creating a dog.");
15   }
16   public static void main(String... strings) {
17     var d1 = new Dog("Kara");
18     var d2 = new Dog("Gece");
19     d2.bark();
20   }
21 }
```



The instance initializer runs **before** the constructor body.

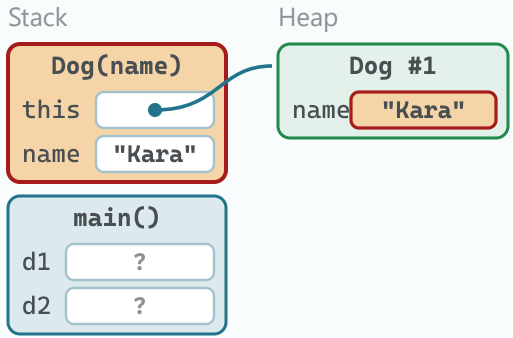
Step 3 / 7

OUTPUT

```
Dogs!
Creating a dog.
```

Dog: Initializer Order

```
1  public class Dog {
2    String name;                  // field
3    Dog(String name) {           // constructor
▶      this.name = name;
5      System.out.println(name);
6    }
7    String bark() {              // instance method
8      return name + " woof woof!";
9    }
10   static {                     // static initializer
11     System.out.println("Dogs!");
12   }
13   {                            // instance initializer
14     System.out.println("Creating a dog.");
15   }
16   public static void main(String... strings) {
17     var d1 = new Dog("Kara");
18     var d2 = new Dog("Gece");
19     d2.bark();
20   }
21 }
```



Now the constructor body: `this.name = name`.

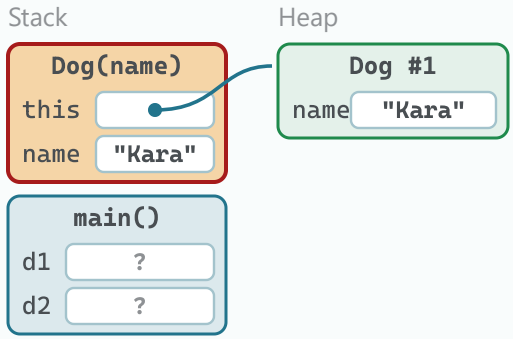
Step 4 / 7

OUTPUT

```
Dogs!
Creating a dog.
```

Dog: Initializer Order

```
1  public class Dog {
2    String name;                  // field
3    Dog(String name) {            // constructor
4      this.name = name;
▶      System.out.println(name);
6    }
7    String bark() {               // instance method
8      return name + " woof woof!";
9    }
10   static {                      // static initializer
11     System.out.println("Dogs!");
12   }
13   {                             // instance initializer
14     System.out.println("Creating a dog.");
15   }
16   public static void main(String... strings) {
17     var d1 = new Dog("Kara");
18     var d2 = new Dog("Gece");
19     d2.bark();
20   }
21 }
```



The constructor prints the name: **Kara**.

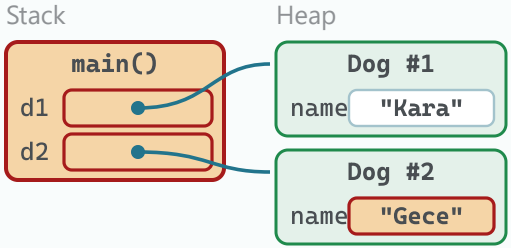
Step 5 / 7

OUTPUT

```
Dogs!
Creating a dog.
Kara
```

Dog: Initializer Order

```
1  public class Dog {
2    String name;                  // field
3    Dog(String name) {            // constructor
4      this.name = name;
5      System.out.println(name);
6    }
7    String bark() {               // instance method
8      return name + " woof woof!";
9    }
10   static {                      // static initializer
11     System.out.println("Dogs!");
12   }
13   {                             // instance initializer
14     System.out.println("Creating a dog.");
15   }
16   public static void main(String... strings) {
17     var d1 = new Dog("Kara");
▶      var d2 = new Dog("Gece");
19     d2.bark();
20   }
21 }
```



Second object: initializer + constructor again; the `static` block **does not run again**.

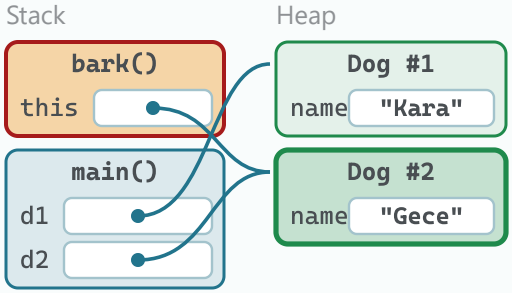
Step 6 / 7

OUTPUT

```
Dogs!
Creating a dog.
Kara
Creating a dog.
Gece
```

Dog: Initializer Order

```
1  public class Dog {
2    String name;                  // field
3    Dog(String name) {            // constructor
4      this.name = name;
5      System.out.println(name);
6    }
7    String bark() {               // instance method
▶      return name + " woof woof!";
9    }
10   static {                      // static initializer
11     System.out.println("Dogs!");
12   }
13   {                             // instance initializer
14     System.out.println("Creating a dog.");
15   }
16   public static void main(String... strings) {
17     var d1 = new Dog("Kara");
18     var d2 = new Dog("Gece");
▶      d2.bark();
20   }
21 }
```



`bark()` returns a string, but the value is **unused**: nothing is printed.

Step 7 / 7

OUTPUT

```
Dogs!
Creating a dog.
Kara
Creating a dog.
Gece
```

Kinds of Methods

- **Methods** are blocks of code in a class that do a specific job; they define the class's **behavior**.
- **Instance methods** are tied to an **object**: `phone.call(...)`. They use that object's fields.
- **Class methods** (`static`) **belong to the class itself** and are shared by all objects: called through `Phone`.
- Static methods belong to no particular object, so instance data (such as `model`) **cannot be accessed directly**.

```
public class Phone {
  private String model;
  private static String osVersion = "Version 1";

  // Instance method
  public void call(String number) {
    System.out.println(model + " calling: " + number);
  }

  // Instance method
  public void sendMessage(String to, String msg) {
    System.out.println(to + " <- " + msg);
  }

  // Class method (static)
  public static void updateOS(String newVersion) {
    osVersion = newVersion;
  }
}
```

Method Example: Email

```
public class Email {
  public boolean send(String to, String subject,
            String body) {
    // a simulation, not a real send
    if (to == null || !to.contains("@")) {
      return false;               // failed
    }
    System.out.println(to + " <- " + subject);
    return true;                    // succeeded
  }

  public static void main(String[] args) {
    Email e = new Email();
    boolean s1 = e.send("ali@example.com", "Homework", "Hi");
    boolean s2 = e.send("ali.example.com", "Homework", "Hi");
    System.out.println(s1 + " " + s2);
  }
}
```

The send method

- Name: `send`
- Parameters: three `String`s: to, subject, body
- Return type: `boolean`: was the send successful?

OUTPUT

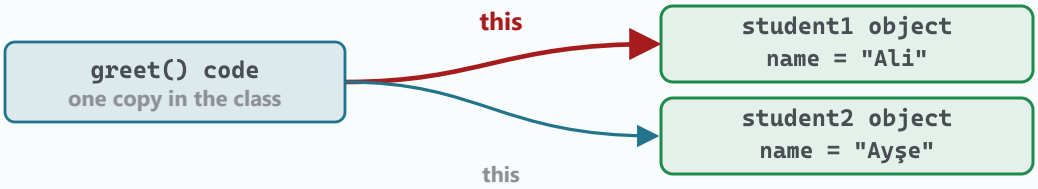
```
ali@example.com <- Homework
true false
```

`boolean` methods: **every path** must end in a `return`; a method whose body is only a comment gives a `missing return statement` error.

Instance Fields and Methods

- Declared in the class without the `static` keyword.
- Every object has its **own copy** of the fields, so objects can hold different data.
- Fields represent the object's **properties**; methods, its **behavior**.
- To use them, first **create an object**: `new`, then `obj.method()`.
- The method **code** is shared by all objects; on each call `this` decides whose data it uses.

	Instance	**`static`**
Belongs to?	Object	Class
Call	`obj.method()`	`ClassName.method()`
Needs an object?	Yes	No
`this` available?	Yes	No
Example	`str.length()`	`Math.sqrt(x)`



The Dot Operator

```
public class Phone {
  String number;
  public Phone(String number) {
    this.number = number;
  }
  public void call(Phone phone) {
    System.out.println(number + " calling: " + phone.number);
  }
  public void sendMessage(Phone phone, String msg) {
    System.out.println(phone.number + " <- " + msg);
  }
  public void openCamera() {
    System.out.println(number + " opened the camera.");
  }
  public static void main(String[] args) {
    Phone phone1 = new Phone("5551234567");
    Phone phone2 = new Phone("5051234567");
    phone1.call(phone2);
    phone2.sendMessage(phone1, "Hello!");
    phone1.openCamera();
  }
}
```

OUTPUT

```
5551234567 calling: 5051234567
5551234567 <- Hello!
5551234567 opened the camera.
```

- Once an object exists, its fields and methods are reached with the **dot operator**: `obj.method(...)`, `obj.field`.
- `phone1.call(...)`: **phone1** places the call; in the method, `number` is phone1's and `phone.number` is phone2's number.
- An object can take another object as a **parameter**.

The Rectangle Class

```
public class Rectangle {
  private double length;     // rectangle's length
  private double width;    // rectangle's width

  // Constructor
  public Rectangle(double length, double width) {
    this.length = length;
    this.width = width;
  }

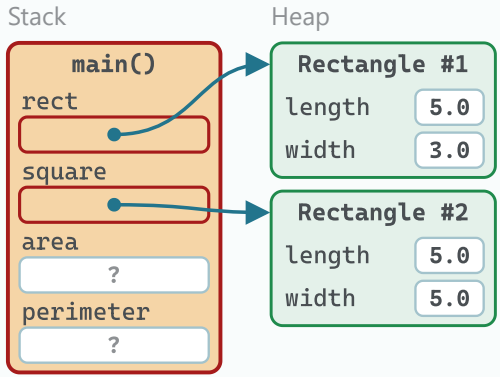
  public double area() {      // computes the area
    return length * width;
  }

  public double perimeter() {     // computes the perimeter
    return 2 * (length + width);
  }
}
```

- Fields are `private`: used from outside only through methods.
- The constructor takes the two sides and stores them in the object.
- `area()` and `perimeter()` take no parameters: the data they need is **inside the object**.
- The procedural version would be `area(5.0, 3.0)`; with an object it is `rect.area()`.

Rectangle: Step by Step

```
1  public class RectangleTest {
2    public static void main(String[] args) {
▶      Rectangle rect = new Rectangle(5.0, 3.0);
▶      Rectangle square = new Rectangle(5.0, 5.0);
5      double area = rect.area();
6      double perimeter = rect.perimeter();
7      System.out.println("Area: " + area
8          + " Perimeter: " + perimeter);
9      area = square.area();
10     perimeter = square.perimeter();
11     System.out.println("Area: " + area
12         + " Perimeter: " + perimeter);
13   }
14 }
```



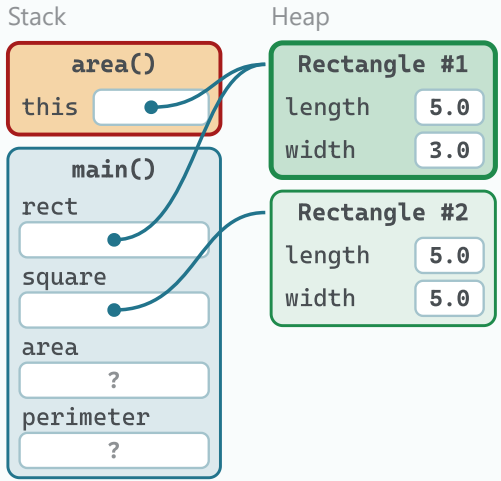
Two objects, each with its own `length` and `width` values.

Step 1 / 6

OUTPUT

Rectangle: Step by Step

```
1  public class RectangleTest {
2    public static void main(String[] args) {
3      Rectangle rect = new Rectangle(5.0, 3.0);
4      Rectangle square = new Rectangle(5.0, 5.0);
▶      double area = rect.area();
6      double perimeter = rect.perimeter();
7      System.out.println("Area: " + area
8          + " Perimeter: " + perimeter);
9      area = square.area();
10     perimeter = square.perimeter();
11     System.out.println("Area: " + area
12         + " Perimeter: " + perimeter);
13   }
14 }
```



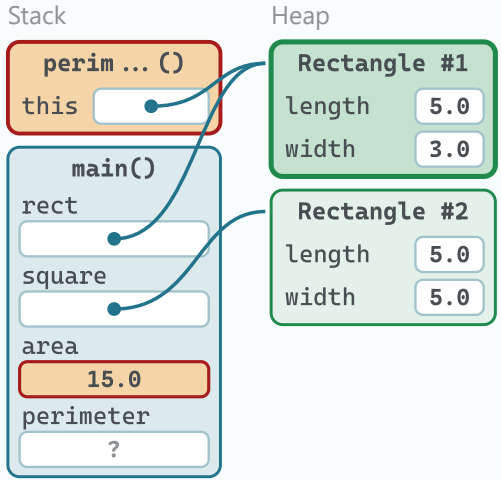
`rect.area()` : `this` = #1, result 5.0 × 3.0 = 15.0.

Step 2 / 6

OUTPUT

Rectangle: Step by Step

```
1  public class RectangleTest {
2    public static void main(String[] args) {
3      Rectangle rect = new Rectangle(5.0, 3.0);
4      Rectangle square = new Rectangle(5.0, 5.0);
5      double area = rect.area();
▶      double perimeter = rect.perimeter();
7      System.out.println("Area: " + area
8          + " Perimeter: " + perimeter);
9      area = square.area();
10     perimeter = square.perimeter();
11     System.out.println("Area: " + area
12         + " Perimeter: " + perimeter);
13   }
14 }
```



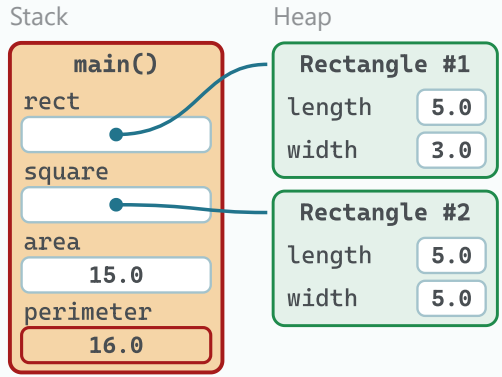
`area` = 15.0. `perimeter()` : 2 × (5.0 + 3.0) = 16.0.

Step 3 / 6

OUTPUT

Rectangle: Step by Step

```
1  public class RectangleTest {
2    public static void main(String[] args) {
3      Rectangle rect = new Rectangle(5.0, 3.0);
4      Rectangle square = new Rectangle(5.0, 5.0);
5      double area = rect.area();
6      double perimeter = rect.perimeter();
▶      System.out.println("Area: " + area
▶          + " Perimeter: " + perimeter);
9      area = square.area();
10     perimeter = square.perimeter();
11     System.out.println("Area: " + area
12         + " Perimeter: " + perimeter);
13   }
14 }
```



`double` values print as `15.0`.

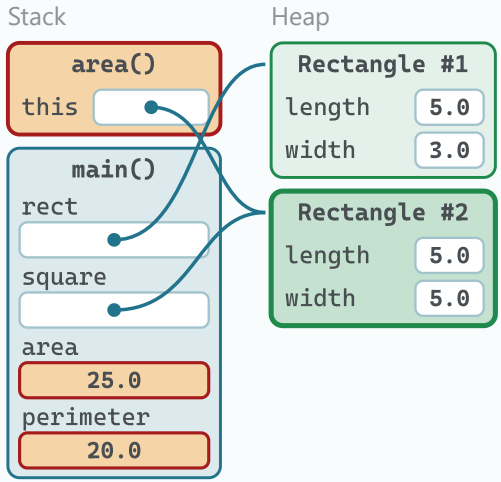
Step 4 / 6

OUTPUT

```
Area: 15.0 Perimeter: 16.0
```

Rectangle: Step by Step

```
1  public class RectangleTest {
2    public static void main(String[] args) {
3      Rectangle rect = new Rectangle(5.0, 3.0);
4      Rectangle square = new Rectangle(5.0, 5.0);
5      double area = rect.area();
6      double perimeter = rect.perimeter();
7      System.out.println("Area: " + area
8          + " Perimeter: " + perimeter);
▶      area = square.area();
▶      perimeter = square.perimeter();
11     System.out.println("Area: " + area
12         + " Perimeter: " + perimeter);
13   }
14 }
```



Same methods, but now `this` = #2 (square): 25.0 and 20.0. The old values are overwritten.

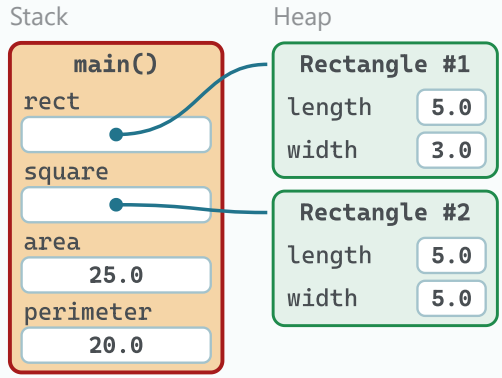
Step 5 / 6

OUTPUT

```
Area: 15.0 Perimeter: 16.0
```

Rectangle: Step by Step

```
1  public class RectangleTest {
2    public static void main(String[] args) {
3      Rectangle rect = new Rectangle(5.0, 3.0);
4      Rectangle square = new Rectangle(5.0, 5.0);
5      double area = rect.area();
6      double perimeter = rect.perimeter();
7      System.out.println("Area: " + area
8          + " Perimeter: " + perimeter);
9      area = square.area();
10     perimeter = square.perimeter();
▶      System.out.println("Area: " + area
▶          + " Perimeter: " + perimeter);
13   }
14 }
```



The second line is printed. A long statement can span two lines; it lasts until the `;`.

Step 6 / 6

OUTPUT

```
Area: 15.0 Perimeter: 16.0
Area: 25.0 Perimeter: 20.0
```

STATIC MEMBERS

static Variables and Methods

```
public class MathUtil {
  // Static add method
  public static int add(int a, int b) {
    return a + b;
  }

  public static void main(String[] args) {
    int result = MathUtil.add(5, 3);
    System.out.println("Sum: " + result);
    System.out.println(Math.max(4, 9));
    System.out.println(Math.sqrt(16.0));
  }
}
```

- `static` members belong to the **class**.
- To access static members you **need no object**; use the class name and a dot: `ClassName.fieldName`, `ClassName.methodName(args)`.
- They can also be reached through an object (`obj.fieldName`), but this is **discouraged**: readers will think the member belongs to the object.
- Built-in examples: `Math.max()`, `Math.sqrt()`, `Math.PI`, `Integer.parseInt()` and `main`.

OUTPUT

```
Sum: 8
9
4.0
```

STATIC MEMBERS

static Rules

From →	Using instance field/method	Using static field/method
From an instance method	✓ directly (`name` , `this.name`)	✓ directly
`static` method (`main` included)	✗ only through an object (`s.name`)	✓ directly

```
public class Greeter {
  private String name = "Ali";
  public void greet() {
    System.out.println("Hello " + name);
  }
  public static void main(String[] args) {
    greet();
  }
}
```

```
error: non-static method greet() cannot be
referenced from a static context
```

```
public class Greeter2 {
  private String name = "Ali";
  public static void print() {
    System.out.println(name);
  }
}
```

```
error: non-static variable name cannot be
referenced from a static context
```

Fix: call it through an object (`new Greeter().greet();`) or make the method `static` .

Reference Assignment

- In Java, objects are handled through **references** (addresses).
- `car2 = car1;` does **not copy** the object, only the reference: both variables point to the **same object** (an alias).
- A change made through one is visible through the other.

Library label

The label on the shelf is not the book itself; it shows **where it is**.

Car key

A key points to a car; two keys open the same car.

```
public class Car {
  private String brand;

  public Car(String brand) {
    this.brand = brand;
  }

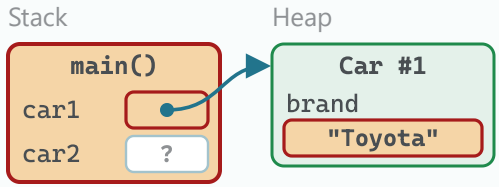
  public String getBrand() {
    return brand;
  }

  public void setBrand(String brand) {
    this.brand = brand;
  }

}
```

Reference Assignment: Step by Step

```
1  public static void main(String[] args) {
▶    Car car1 = new Car("Toyota");
3    Car car2 = car1;
4    System.out.println("Car 1: " + car1.getBrand());
5    System.out.println("Car 2: " + car2.getBrand());
6    car2.setBrand("Ford");
7    System.out.println("Car 1: " + car1.getBrand());
8    car1 = null;
9    System.out.println("Car 2: " + car2.getBrand());
10 }
```



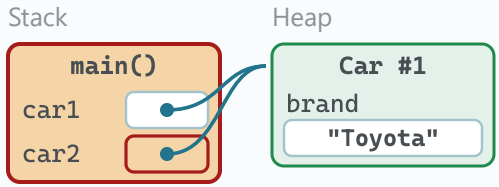
An object is created; `car1` refers to it.

Step 1 / 7

OUTPUT

Reference Assignment: Step by Step

```
1  public static void main(String[] args) {
2    Car car1 = new Car("Toyota");
▶    Car car2 = car1;
4    System.out.println("Car 1: " + car1.getBrand());
5    System.out.println("Car 2: " + car2.getBrand());
6    car2.setBrand("Ford");
7    System.out.println("Car 1: " + car1.getBrand());
8    car1 = null;
9    System.out.println("Car 2: " + car2.getBrand());
10 }
```



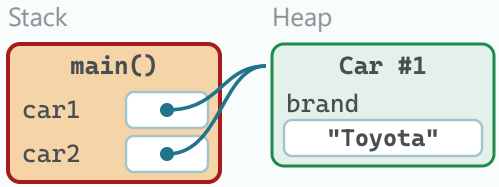
`car2 = car1` : the reference is copied. There is still a **single** object.

Step 2 / 7

OUTPUT

Reference Assignment: Step by Step

```
1  public static void main(String[] args) {
2    Car car1 = new Car("Toyota");
3    Car car2 = car1;
▶    System.out.println("Car 1: " + car1.getBrand());
▶    System.out.println("Car 2: " + car2.getBrand());
6    car2.setBrand("Ford");
7    System.out.println("Car 1: " + car1.getBrand());
8    car1 = null;
9    System.out.println("Car 2: " + car2.getBrand());
10 }
```



Both variables read the same object: both say Toyota.

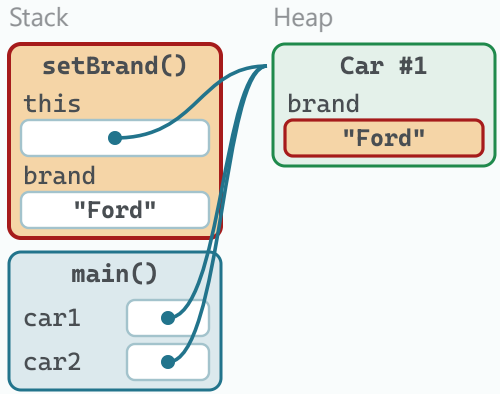
Step 3 / 7

OUTPUT

```
Car 1: Toyota
Car 2: Toyota
```

Reference Assignment: Step by Step

```
1  public static void main(String[] args) {
2    Car car1 = new Car("Toyota");
3    Car car2 = car1;
4    System.out.println("Car 1: " + car1.getBrand());
5    System.out.println("Car 2: " + car2.getBrand());
▶    car2.setBrand("Ford");
7    System.out.println("Car 1: " + car1.getBrand());
8    car1 = null;
9    System.out.println("Car 2: " + car2.getBrand());
10 }
```



`car2.setBrand("Ford")` changes the shared object.

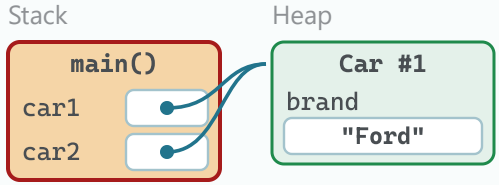
Step 4 / 7

OUTPUT

```
Car 1: Toyota
Car 2: Toyota
```

Reference Assignment: Step by Step

```
1  public static void main(String[] args) {
2    Car car1 = new Car("Toyota");
3    Car car2 = car1;
4    System.out.println("Car 1: " + car1.getBrand());
5    System.out.println("Car 2: " + car2.getBrand());
6    car2.setBrand("Ford");
▶    System.out.println("Car 1: " + car1.getBrand());
8    car1 = null;
9    System.out.println("Car 2: " + car2.getBrand());
10 }
```



`car1` now reads **Ford** too: the change is shared.

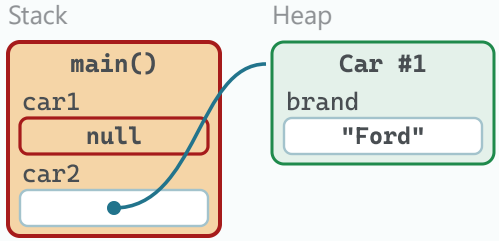
Step 5 / 7

OUTPUT

```
Car 1: Toyota
Car 2: Toyota
Car 1: Ford
```

Reference Assignment: Step by Step

```
1  public static void main(String[] args) {
2    Car car1 = new Car("Toyota");
3    Car car2 = car1;
4    System.out.println("Car 1: " + car1.getBrand());
5    System.out.println("Car 2: " + car2.getBrand());
6    car2.setBrand("Ford");
7    System.out.println("Car 1: " + car1.getBrand());
▶    car1 = null;
9    System.out.println("Car 2: " + car2.getBrand());
10 }
```



`car1 = null` : only this variable lets go; the object is not deleted, and `car2` still refers to it.

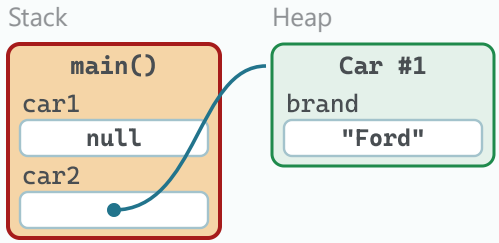
Step 6 / 7

OUTPUT

```
Car 1: Toyota
Car 2: Toyota
Car 1: Ford
```

Reference Assignment: Step by Step

```
1  public static void main(String[] args) {
2    Car car1 = new Car("Toyota");
3    Car car2 = car1;
4    System.out.println("Car 1: " + car1.getBrand());
5    System.out.println("Car 2: " + car2.getBrand());
6    car2.setBrand("Ford");
7    System.out.println("Car 1: " + car1.getBrand());
8    car1 = null;
▶    System.out.println("Car 2: " + car2.getBrand());
10 }
```



When the last reference to an object is gone, the **garbage collector** removes it.

Step 7 / 7

OUTPUT

```
Car 1: Toyota
Car 2: Toyota
Car 1: Ford
Car 2: Ford
```

null and NullPointerException

```
public class Empty {
  String name;

  public static void main(String[] args) {
    Empty b = null;
    System.out.println(b.name.length());
  }
}
```

```
Exception in thread "main"
java.lang.NullPointerException: Cannot read
field "name" because "<local1>" is null
    at Empty.main(Empty.java:6)
```

If a local variable is used before it gets a value, the program **does not compile**: `error: variable sb might not have been initialized`

- `null` means the reference variable **points to no object**.
- `null` references: reading a field or calling a method through one throws `NullPointerException` at run time.
- Prevention: check before use: `if (b != null) { ... }`
- Objects that no reference points to are removed automatically by the JVM's **garbage collector**; in Java you never delete objects by hand.



The this Keyword

```
public class Shadow {
  private String name;

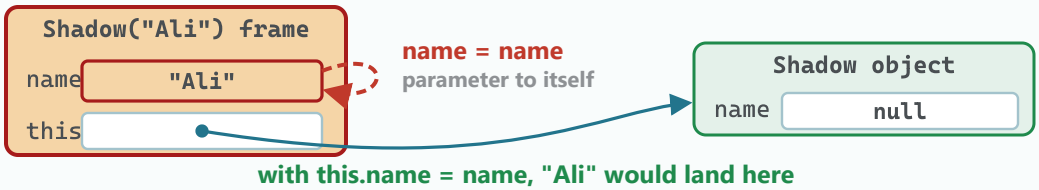
  public Shadow(String name) {
    name = name;            // forgot this!
  }

  public static void main(String[] args) {
    Shadow g = new Shadow("Ali");
    System.out.println(g.name);
  }
}
```

OUTPUT

```
null
```

- `this` is a reference to the **object the method was called on**.
- `static` methods **cannot use it** (there is no object there).
- Most often used in constructors, when the field and the parameter have the **same** name: `this.name = name;`
- The code on the left forgot `this`: `name = name;` assigns the parameter to itself, and the field stays `null`. The compiler reports no error!
- `this(...)`, by contrast, calls another constructor of the same class.



Passing the Object with this

```
public class Team {
  private String name;
  public Team(String name) {
    this.name = name;
  }
  public void addPlayer(Player player) {
    player.joinTeam(this);   // this object
  }
  public String getName() {
    return name;
  }
  public static void main(String[] args) {
    Team team = new Team("Giresunspor");
    Player player = new Player("Mert");
    team.addPlayer(player);
  }
}
```

- `this` passes the current object to another method as a **parameter**.
- `team.addPlayer(player)` runs with `this` = `team`; `joinTeam(this)` hands it to the player.
- A second class in the same file (not `public`):

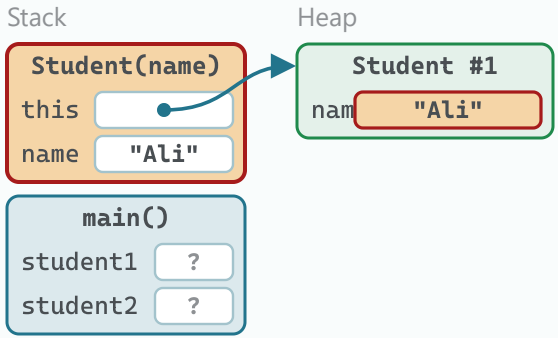
```
class Player {
  private String name;
  Player(String name) {
    this.name = name;
  }
  void joinTeam(Team team) {
    String t = team.getName();
    System.out.println(name + " -> " + t);
  }
}
```

OUTPUT

```
Mert -> Giresunspor
```

Greeting with this: Step by Step

```
1  public class Student {
2    private String name;
3
▶    public Student(String name) {
▶      this.name = name;
6    }
7
8    public void greet() {
9      System.out.println("Hi, I am " + this.name);
10   }
11
12   public static void main(String[] args) {
▶      Student student1 = new Student("Ali");
14     Student student2 = new Student("Ayşe");
15     student1.greet();
16     student2.greet();
17   }
18 }
```



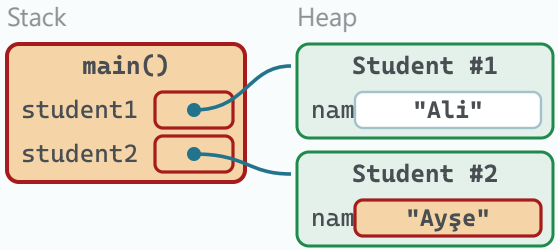
In the constructor, `this` is the new object: `this.name` (field) = `name` (parameter).

Step 1 / 4

OUTPUT

Greeting with this: Step by Step

```
1  public class Student {
2    private String name;
3
4    public Student(String name) {
5      this.name = name;
6    }
7
8    public void greet() {
9      System.out.println("Hi, I am " + this.name);
10   }
11
12   public static void main(String[] args) {
13     Student student1 = new Student("Ali");
▶      Student student2 = new Student("Ayşe");
15     student1.greet();
16     student2.greet();
17   }
18 }
```



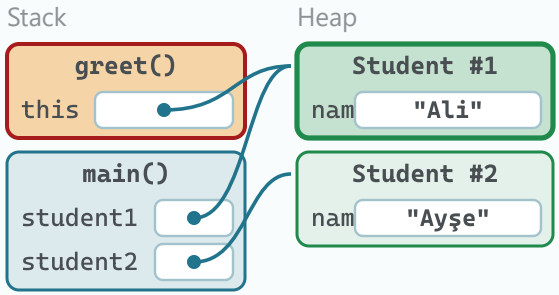
The second object is built by the same constructor; at that time `this` was #2.

Step 2 / 4

OUTPUT

Greeting with this: Step by Step

```
1  public class Student {
2    private String name;
3
4    public Student(String name) {
5      this.name = name;
6    }
7
▶    public void greet() {
▶      System.out.println("Hi, I am " + this.name);
10   }
11
12   public static void main(String[] args) {
13     Student student1 = new Student("Ali");
14     Student student2 = new Student("Ayşe");
▶      student1.greet();
16     student2.greet();
17   }
18 }
```



`student1.greet()` : the object left of the dot becomes `this` → "Ali".

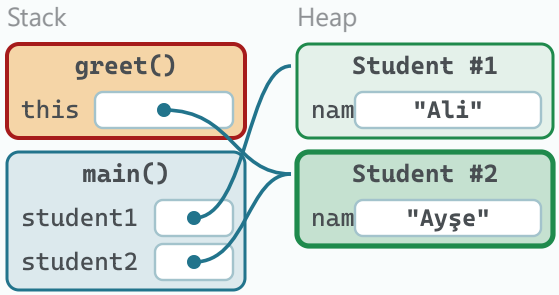
Step 3 / 4

OUTPUT

```
Hi, I am Ali
```

Greeting with this: Step by Step

```
1  public class Student {
2    private String name;
3
4    public Student(String name) {
5      this.name = name;
6    }
7
▶    public void greet() {
▶      System.out.println("Hi, I am " + this.name);
10   }
11
12   public static void main(String[] args) {
13     Student student1 = new Student("Ali");
14     Student student2 = new Student("Ayşe");
15     student1.greet();
▶      student2.greet();
17   }
18 }
```



Same code, different `this` → "Ayşe". Here `this.name` and `name` do the same thing.

Step 4 / 4

OUTPUT

```
Hi, I am Ali
Hi, I am Ayşe
```

`this(...)` and Constructor Chaining

- `this(...)` calls **another constructor** of the same class. Shared initialization lives in one constructor; the others delegate to it, supplying the missing arguments.
- `this(...)` must be the **first** statement of the constructor; placed after another statement, it does not compile.
- The called constructor's body finishes first, then the caller's remaining lines run; however long the chain, only one object is built.
- If constructors call each other in a cycle, the compiler catches it.

```
class Employee {
  private String name;
  private double salary;
  Employee(String name) {
    this(name, 20000.0);   // same-class constructor
  }
  Employee(String name, double salary) {
    this.name = name;        // init in one place
    this.salary = salary;
  }
}
```

Not on the first line: `A() { x = 1; this(2); }`

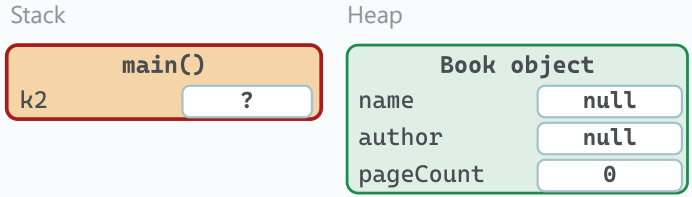
```
error: call to this must be first statement in
constructor
```

Cycle: `A() { this(1); }` and `A(int x) { this(); }`

```
error: recursive constructor invocation
```

new Book("Nutuk", "Atatürk"): Step by Step

```
1  class Book {
2    String name;
3    String author;
4    int pageCount;
5    public Book(String name) {
6      this.name = name;
7    }
8    public Book(String name, String author) {
9      this(name);  // constructor chaining
10     this.author = author;
11   }
12 }
13 // in main:
▶ Book k2 = new Book("Nutuk", "Atatürk");
15 System.out.println(k2.name + " / "
16     + k2.author + " / " + k2.pageCount);
```



`new` allocates a new object on the heap; fields get default values: `null`, `null`, `0`.

Step 1 / 7

OUTPUT

CALL STACK

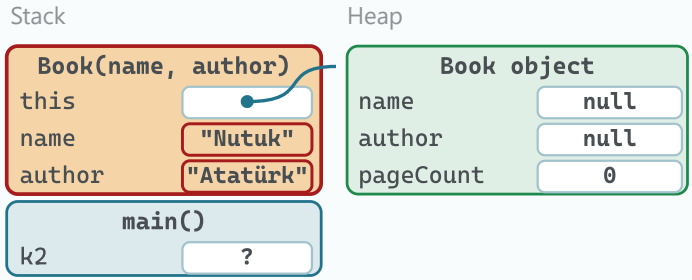
`main()`

VARIABLES

`k2 = ?`

new Book("Nutuk", "Atatürk"): Step by Step

```
1  class Book {
2    String name;
3    String author;
4    int pageCount;
5    public Book(String name) {
6      this.name = name;
7    }
▶    public Book(String name, String author) {
9      this(name);  // constructor chaining
10     this.author = author;
11   }
12 }
13 // in main:
14 Book k2 = new Book("Nutuk", "Atatürk");
15 System.out.println(k2.name + " / "
16     + k2.author + " / " + k2.pageCount);
```



The arguments are two `String` values, so the **two-parameter** constructor runs. `this` points to the new object.

Step 2 / 7

OUTPUT

VARIABLES

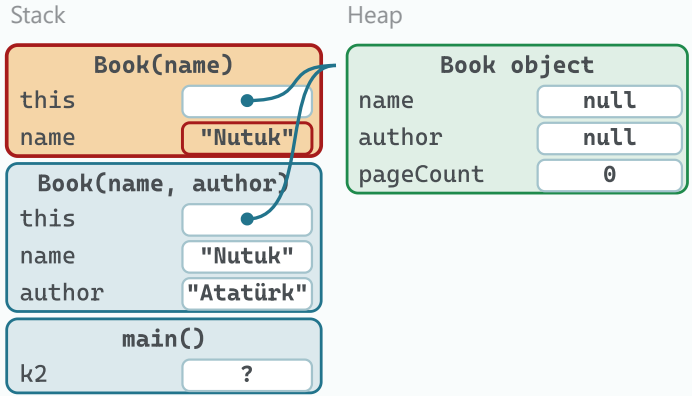
`name = "Nutuk"` `author = "Atatürk"`

CALL STACK

- `Book(name, author)`
- `main()`

new Book("Nutuk", "Atatürk"): Step by Step

```
1  class Book {
2    String name;
3    String author;
4    int pageCount;
5    public Book(String name) {
6      this.name = name;
7    }
8    public Book(String name, String author) {
▶      this(name);  // constructor chaining
10     this.author = author;
11   }
12 }
13 // in main:
14 Book k2 = new Book("Nutuk", "Atatürk");
15 System.out.println(k2.name + " / "
16     + k2.author + " / " + k2.pageCount);
```



`this(name)` : **chaining**. The one-parameter constructor is called for the same object.

Step 3 / 7

OUTPUT

VARIABLES

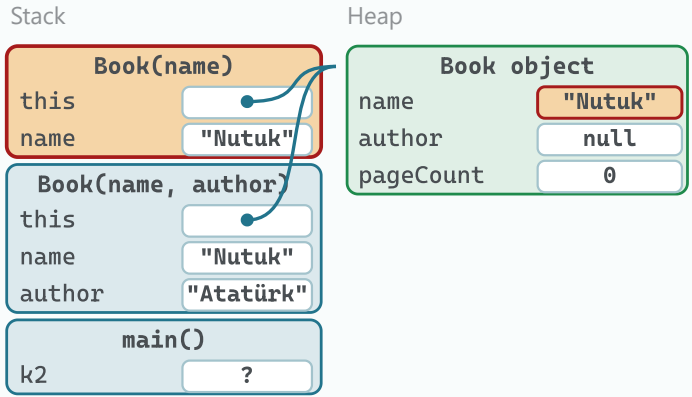
`name = "Nutuk"`

CALL STACK

- **Book(name)**
- Book(name, author)
- main()

new Book("Nutuk", "Atatürk"): Step by Step

```
1  class Book {
2    String name;
3    String author;
4    int pageCount;
5    public Book(String name) {
▶      this.name = name;
7    }
8    public Book(String name, String author) {
9      this(name);  // constructor chaining
10     this.author = author;
11   }
12 }
13 // in main:
14 Book k2 = new Book("Nutuk", "Atatürk");
15 System.out.println(k2.name + " / "
16     + k2.author + " / " + k2.pageCount);
```



`this.name = name` : the parameter's value is written to the object's `name` field.

Step 4 / 7

OUTPUT

VARIABLES

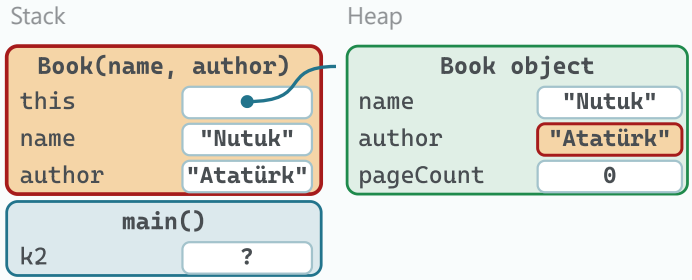
`name = "Nutuk"`

CALL STACK

- **Book(name)**
- Book(name, author)
- main()

new Book("Nutuk", "Atatürk"): Step by Step

```
1  class Book {
2    String name;
3    String author;
4    int pageCount;
5    public Book(String name) {
6      this.name = name;
7    }
8    public Book(String name, String author) {
9      this(name);  // constructor chaining
▶      this.author = author;
11   }
12 }
13 // in main:
14 Book k2 = new Book("Nutuk", "Atatürk");
15 System.out.println(k2.name + " / "
16     + k2.author + " / " + k2.pageCount);
```



Inner constructor done. Remaining line: `this.author = author`.

Step 5 / 7

OUTPUT

VARIABLES

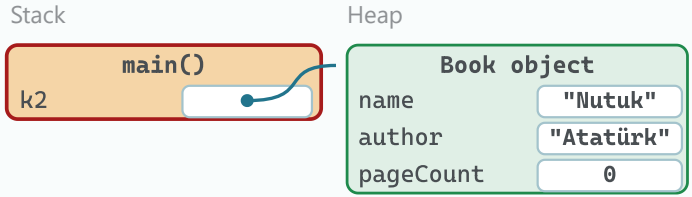
`name = "Nutuk"` `author = "Atatürk"`

CALL STACK

- `Book(name, author)`
- `main()`

new Book("Nutuk", "Atatürk"): Step by Step

```
1  class Book {
2    String name;
3    String author;
4    int pageCount;
5    public Book(String name) {
6      this.name = name;
7    }
8    public Book(String name, String author) {
9      this(name);  // constructor chaining
10     this.author = author;
11   }
12 }
13 // in main:
▶ Book k2 = new Book("Nutuk", "Atatürk");
15 System.out.println(k2.name + " / "
16     + k2.author + " / " + k2.pageCount);
```



Constructors done; the `new` expression evaluates to the object's **reference**. `k2` now refers to the object.

Step 6 / 7

OUTPUT

CALL STACK

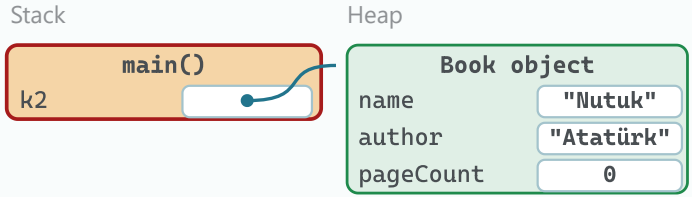
`main()`

VARIABLES

`k2 = →Book`

new Book("Nutuk", "Atatürk"): Step by Step

```
1  class Book {
2    String name;
3    String author;
4    int pageCount;
5    public Book(String name) {
6      this.name = name;
7    }
8    public Book(String name, String author) {
9      this(name);  // constructor chaining
10     this.author = author;
11   }
12 }
13 // in main:
14 Book k2 = new Book("Nutuk", "Atatürk");
▶  System.out.println(k2.name + " / "
▶      + k2.author + " / " + k2.pageCount);
```



The dot operator reads the fields: **Nutuk / Atatürk / 0**.

Step 7 / 7

OUTPUT

```
Nutuk / Atatürk / 0
```

CALL STACK

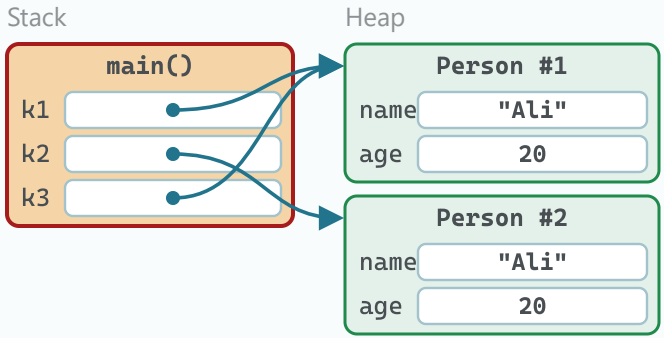
`main()`

VARIABLES

`k2 = →Book`

== and equals()

- Variables store only **references** to objects.
- `==` compares **memory addresses**, not fields: do both variables point to the **same object**?
- Content equality of two objects is checked with `equals()`, which compares all properties; in our own classes it must be **overridden**.
- `String` and wrappers are objects too: for "Java" read from the keyboard and the literal `"Java"`, `==` gives **false**, `equals` gives **true**.
- `equals`, `hashCode`, `toString` and `getClass` come to every class from `Object`; when `equals` is overridden, `hashCode` must be too.



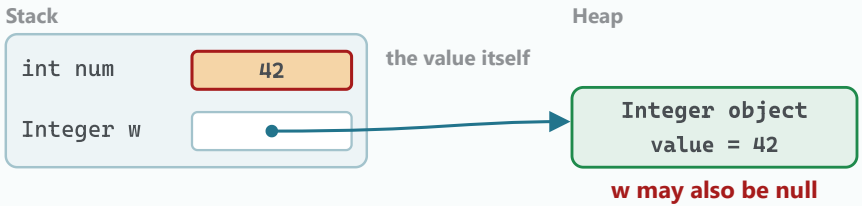
Expression	Result
`k1 == k3`	true: same object
`k1 == k2`	false: different objects
`k1.equals(k2)`	true: same content*

* `equals` if overridden; otherwise `Object`'s version behaves like `==`.

Wrapper Classes

Primitive type	Wrapper class
`byte`	`Byte`
`short`	`Short`
`int`	`Integer`
`long`	`Long`
`float`	`Float`
`double`	`Double`
`char`	`Character`
`boolean`	`Boolean`

- **Wrapper** classes represent primitive types as **objects**.
- Some places require an object: generic collections such as `ArrayList<Integer>` cannot hold primitives.
- A primitive cannot hold `null`; a wrapper variable can ("no value").
- **Helper methods** are provided: `Integer.parseInt("12")`, `Integer.MAX_VALUE`, `Character.isDigit(c)`.
- Primitives are **faster** and use less memory; prefer them unless an object is needed.
- **Autoboxing** and **unboxing** conversions need care.



Autoboxing and Unboxing

```java
public class Wrappers {
  public static void main(String[] args) {
    // an int variable
    int num = 42;
    // using the Integer wrapper
    Integer w1 = Integer.valueOf(num);
    // Autoboxing (automatic)
    Integer w2 = num;
    // Unboxing (explicit and automatic)
    int n2 = w2.intValue();
    int n3 = w2;
    System.out.println(n2 + n3);
    System.out.println(w1.equals(w2));
    System.out.println(Integer.parseInt("123") + 1);
    System.out.println(Integer.MAX_VALUE);
    System.out.println(Character.isDigit('7'));
    System.out.println(Double.parseDouble("2.5") * 2);
    Integer empty = null; // an int can't be null
    System.out.println(empty);
  }
}
```

- **Autoboxing**: primitive → wrapper, **automatically**. **Unboxing**: wrapper → primitive.
- Explicit forms: `Integer.valueOf(num)`, `w.intValue()`.

OUTPUT

```
84
true
124
2147483647
true
5.0
null
```

Wrapper Pitfalls

```java
import java.util.ArrayList;
public class Pitfalls {
  public static void main(String[] args) {
    ArrayList<Integer> list = new ArrayList<>();
    list.add(5);                  // autoboxing: 5 -> Integer
    int first = list.get(0);        // unboxing: Integer -> int
    System.out.println(first + 1);
    Integer a = 127, b = 127;
    Integer c = 128, d = 128;
    System.out.println(a == b);
    System.out.println(c == d);
    System.out.println(c.equals(d));
    Integer empty = null;
    int x = empty;                  // NullPointerException
  }
}
```

```
Exception in thread "main" java.lang.NullPointerException: Cannot
invoke "java.lang.Integer.intValue()" because "<local7>" is null
```

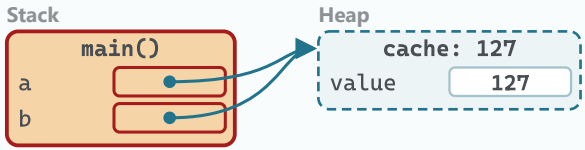
OUTPUT

```
6
true
false
true
```

- Comparing wrapper objects with `==` is wrong: `==` compares references. Values from −128 to 127 come from a cache as the **same object**; for 128 the result is **false**.
- Always compare with `equals()`.
- `null` wrapper: unboxing it throws `NullPointerException` at run time.

Integer Cache: Code Trace

```
▶ Integer a = 127, b = 127;
2 Integer c = 128, d = 128;
3 System.out.println(a == b);
4 System.out.println(c == d);
5 System.out.println(c.equals(d));
6 int t = c + d;
7 System.out.println(t);
8 Integer empty = null;
9 int x = empty;
```



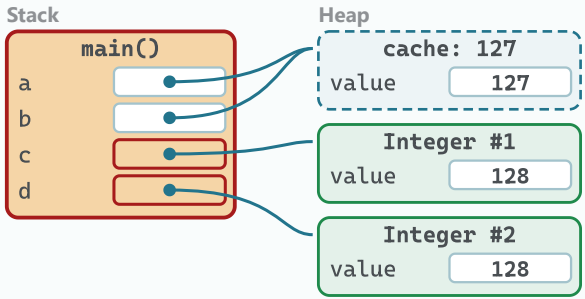
Autoboxing calls `Integer.valueOf(127)`. Values from −128 to 127 are cached in advance: `a` and `b` get the **same** object.

Step 1 / 9

OUTPUT

Integer Cache: Code Trace

```
1 Integer a = 127, b = 127;
▶ Integer c = 128, d = 128;
3 System.out.println(a == b);
4 System.out.println(c == d);
5 System.out.println(c.equals(d));
6 int t = c + d;
7 System.out.println(t);
8 Integer empty = null;
9 int x = empty;
```



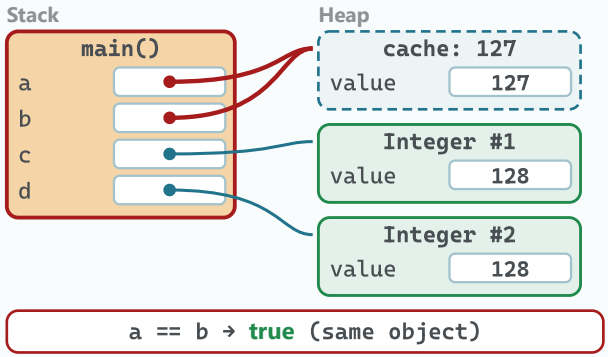
128 is outside the cache range: `valueOf` creates a **new** object every time.

Step 2 / 9

OUTPUT

Integer Cache: Code Trace

```
1 Integer a = 127, b = 127;
2 Integer c = 128, d = 128;
▶ System.out.println(a == b);
4 System.out.println(c == d);
5 System.out.println(c.equals(d));
6 int t = c + d;
7 System.out.println(t);
8 Integer empty = null;
9 int x = empty;
```



`==` compares references; both arrows reach the same object: **true**.

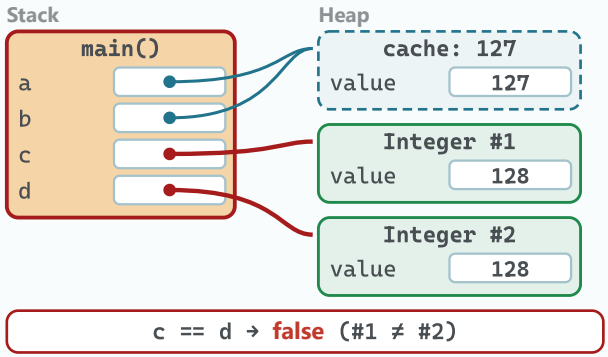
Step 3 / 9

OUTPUT

```
true
```

Integer Cache: Code Trace

```
1 Integer a = 127, b = 127;
2 Integer c = 128, d = 128;
3 System.out.println(a == b);
▶ System.out.println(c == d);
5 System.out.println(c.equals(d));
6 int t = c + d;
7 System.out.println(t);
8 Integer empty = null;
9 int x = empty;
```



Same value (128), different objects: **false**. This is the trap of comparing wrappers with `==`.

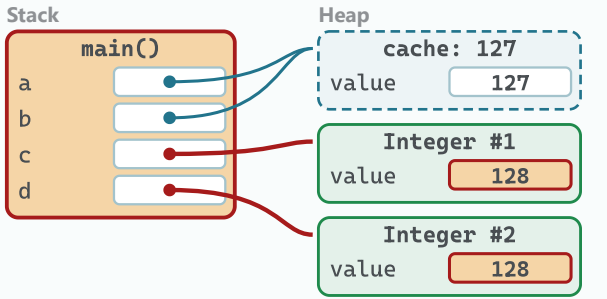
Step 4 / 9

OUTPUT

```
true
false
```

Integer Cache: Code Trace

```
1 Integer a = 127, b = 127;
2 Integer c = 128, d = 128;
3 System.out.println(a == b);
4 System.out.println(c == d);
▶ System.out.println(c.equals(d));
6 int t = c + d;
7 System.out.println(t);
8 Integer empty = null;
9 int x = empty;
```



`Integer.equals` compares the wrapped `int` values: **true**.

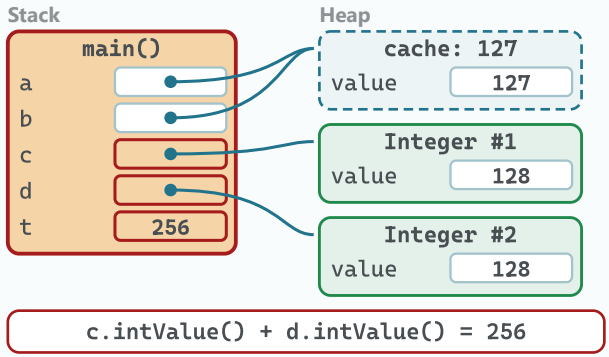
Step 5 / 9

OUTPUT

```
true
false
true
```

Integer Cache: Code Trace

```
1 Integer a = 127, b = 127;
2 Integer c = 128, d = 128;
3 System.out.println(a == b);
4 System.out.println(c == d);
5 System.out.println(c.equals(d));
▶ int t = c + d;
7 System.out.println(t);
8 Integer empty = null;
9 int x = empty;
```



`+` does not work on objects: both are unboxed (`intValue()`).
`t` is a primitive `int` ; its value lives on the stack.

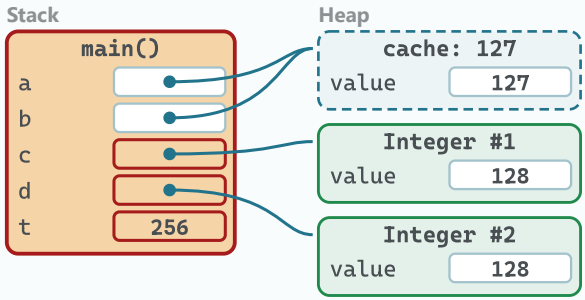
Step 6 / 9

OUTPUT

```
true
false
true
```

Integer Cache: Code Trace

```
1 Integer a = 127, b = 127;
2 Integer c = 128, d = 128;
3 System.out.println(a == b);
4 System.out.println(c == d);
5 System.out.println(c.equals(d));
6 int t = c + d;
▶ System.out.println(t);
8 Integer empty = null;
9 int x = empty;
```



The primitive `t` is printed directly: **256**.

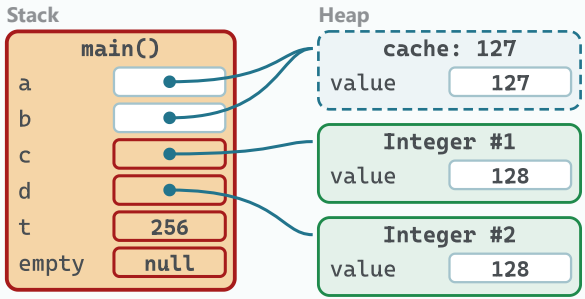
Step 7 / 9

OUTPUT

```
true
false
true
256
```

Integer Cache: Code Trace

```
1 Integer a = 127, b = 127;
2 Integer c = 128, d = 128;
3 System.out.println(a == b);
4 System.out.println(c == d);
5 System.out.println(c.equals(d));
6 int t = c + d;
7 System.out.println(t);
▶ Integer empty = null;
9 int x = empty;
```



A wrapper variable can hold `null`; `int empty = null` would not compile.

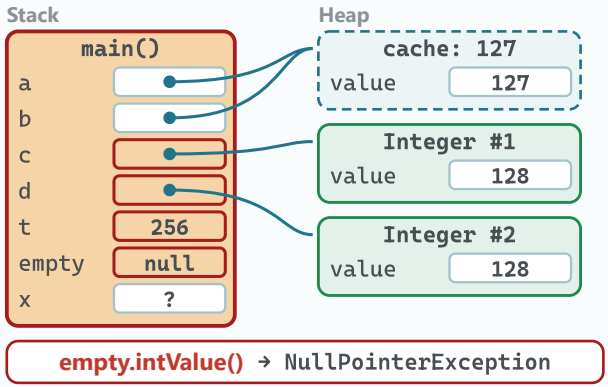
Step 8 / 9

OUTPUT

```
true
false
true
256
```

Integer Cache: Code Trace

```
1 Integer a = 127, b = 127;
2 Integer c = 128, d = 128;
3 System.out.println(a == b);
4 System.out.println(c == d);
5 System.out.println(c.equals(d));
6 int t = c + d;
7 System.out.println(t);
8 Integer empty = null;
▶ int x = empty;
```



`int x = empty` silently becomes `empty.intValue()`. There is no object behind it: **NullPointerException**, and the program stops.

Step 9 / 9

OUTPUT

```
true
false
true
256
Exception in thread "main"
java.lang.NullPointerException
```

Records

- **A record** (Java 16+) is a kind of class for writing data-only classes **concisely**.
- **Immutable** objects: all fields are `private final`; **there are no setters**.
- The compiler generates: a constructor, accessors `x()` and `y()` (names do not start with `get`), `equals`, `hashCode`, `toString`.

```
record Point(double x, double y) { }
```

This single line equals a roughly 30-line class with two fields, a constructor, two accessors and three `Object` methods.

```
public class RecordTest {
  public static void main(String[] args) {
    Point p1 = new Point(1.0, 2.5);
    Point p2 = new Point(1.0, 2.5);
    boolean same = p1.equals(p2);
    System.out.println(same);
    System.out.println(p1 == p2);
    System.out.println(p1);
    System.out.println(p1.x() + p1.y());
  }
}
```

OUTPUT

```
true
false
Point[x=1.0, y=2.5]
3.5
```

```
p.x = 3.0;   →   error: x has private access in Point
```

enum (Enumerated Type)

```
enum Weekday {
  MONDAY, TUESDAY, WEDNESDAY, THURSDAY,
  FRIDAY, SATURDAY, SUNDAY
}
public class DayTest {
  public static void main(String[] args) {
    Weekday[] allDays = Weekday.values();
    Weekday thu = Weekday.valueOf("THURSDAY");
    System.out.println(allDays.length + " days");
    System.out.println(thu + " " + thu.ordinal());
    for (Weekday day : allDays) {
      boolean weekend = day == Weekday.SATURDAY
          || day == Weekday.SUNDAY;
      if (weekend) {
        System.out.println(day + ": weekend");
      }
    }
  }
}
```

OUTPUT

```
7 days
THURSDAY 3
SATURDAY: weekend
SUNDAY: weekend
```

- **enum** is a special type for a **fixed, limited** set of values.
- `values()` returns all values as an array; you can **loop** over it.
- `valueOf` converts text to a value; `ordinal()` gives the 0-based position. Values are compared with `==`.

Inner Classes

- **An inner class** is a class defined inside another class. It can access the outer class's fields and methods directly (`private` ones included).
- Used to group helper classes that serve only one class; very common for event listener classes in GUIs.

Kind	Defined where?	Created how?
static inner class	In the class body, marked `static`	Without an outer object: `new Outer.StaticInner()`
Member inner class	In the class body, without `static`	Through an outer object: `outer.new MemberInner()`
Local inner class	Inside a method	Only from inside that method
Anonymous inner class	Inside an expression, **unnamed**	One object where it is defined; implements an interface or abstract class

static Inner Class

```
public class Outer {
  // a static inner class inside Outer
  static class StaticInner {
    void display() {
      System.out.println("Static Inner Class");
    }
  }

  public static void main(String[] args) {
    // access without an Outer instance
    Outer.StaticInner staticInner = new Outer.StaticInner();
    staticInner.display();
  }
}
```

OUTPUT

```
Static Inner Class
```

- `static` appears in its declaration.
- Can be created **without an object** of the outer class: `new Outer.StaticInner()`.
- It can directly access only the `static` members of the outer class.

Member Inner Class

```
public class Outer {
  private int count = 5;          // outer's private field

  class MemberInner {             // member inner class
    void display() {
      System.out.println("Member Inner Class, count = " + count);
    }
  }

  public static void main(String[] args) {
    // access via an Outer instance
    Outer outer = new Outer();
    Outer.MemberInner memberInner = outer.new MemberInner();
    memberInner.display();
  }
}
```

OUTPUT

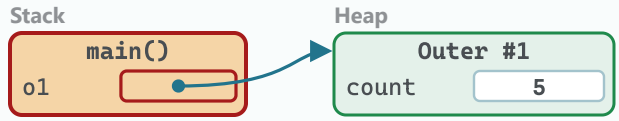
```
Member Inner Class, count = 5
```

- Declared as a **member** of the outer class (no `static`).
- Each member inner class object is **bound to an outer object**: `outer.new MemberInner()`. That is why it can read `private count`.
- In `main`, a plain `new MemberInner()` does not compile:

```
error: non-static variable this
cannot be referenced from a
static context
```

Member Inner Class: Code Trace

```
1  public class Outer {
2      private int count = 5;
3      class MemberInner {
4          void display() {
5              System.out.println("count = " + count);
6          }
7      }
8      public static void main(String[] args) {
▶          Outer o1 = new Outer();
10         Outer o2 = new Outer();
11         o2.count = 8;
12         Outer.MemberInner m = o2.new MemberInner();
13         m.display();
14     }
15 }
```



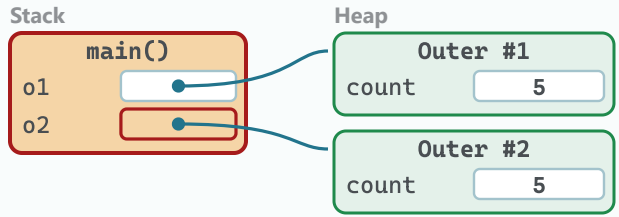
First outer object: its own `count` field is 5.

Step 1 / 6

OUTPUT

Member Inner Class: Code Trace

```
1  public class Outer {
2      private int count = 5;
3      class MemberInner {
4          void display() {
5              System.out.println("count = " + count);
6          }
7      }
8      public static void main(String[] args) {
9          Outer o1 = new Outer();
▶          Outer o2 = new Outer();
11         o2.count = 8;
12         Outer.MemberInner m = o2.new MemberInner();
13         m.display();
14     }
15 }
```



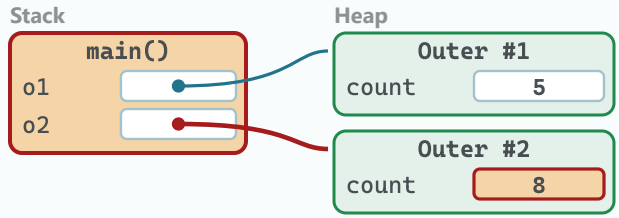
Second outer object; it also starts at 5. There are two separate `count` fields.

Step 2 / 6

OUTPUT

Member Inner Class: Code Trace

```
1  public class Outer {
2      private int count = 5;
3      class MemberInner {
4          void display() {
5              System.out.println("count = " + count);
6          }
7      }
8      public static void main(String[] args) {
9          Outer o1 = new Outer();
10         Outer o2 = new Outer();
▶          o2.count = 8;
12         Outer.MemberInner m = o2.new MemberInner();
13         m.display();
14     }
15 }
```



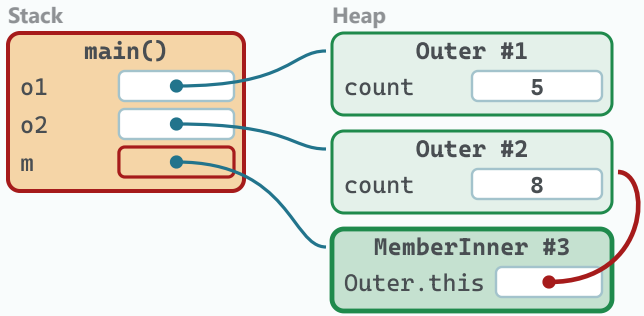
`main` is also inside `Outer`, so it can write the `private` field: only #2 changes.

Step 3 / 6

OUTPUT

Member Inner Class: Code Trace

```
1  public class Outer {
2      private int count = 5;
3      class MemberInner {
4          void display() {
5              System.out.println("count = " + count);
6          }
7      }
8      public static void main(String[] args) {
9          Outer o1 = new Outer();
10         Outer o2 = new Outer();
11         o2.count = 8;
▶          Outer.MemberInner m = o2.new MemberInner();
13         m.display();
14     }
15 }
```



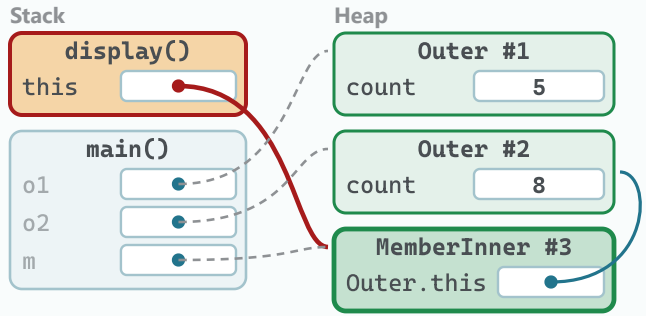
`o2.new MemberInner()` : the inner object keeps a reference to the outer object that created it in a hidden field (`Outer.this`).

Step 4 / 6

OUTPUT

Member Inner Class: Code Trace

```
1  public class Outer {
2      private int count = 5;
3      class MemberInner {
▶          void display() {
5              System.out.println("count = " + count);
6          }
7      }
8      public static void main(String[] args) {
9          Outer o1 = new Outer();
10         Outer o2 = new Outer();
11         o2.count = 8;
12         Outer.MemberInner m = o2.new MemberInner();
▶          m.display();
14     }
15 }
```



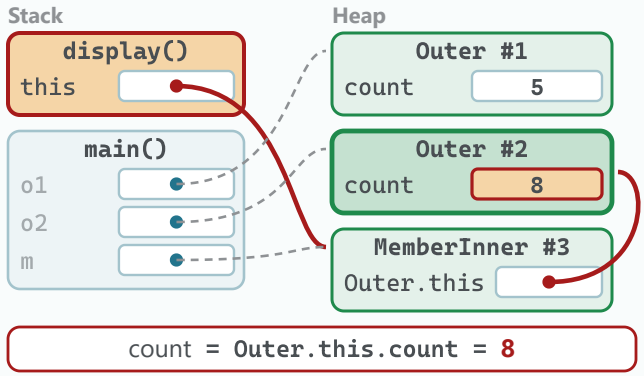
`m.display()` : inside the method, `this` is the inner object (#3). The inner class has no `count` field of its own.

Step 5 / 6

OUTPUT

Member Inner Class: Code Trace

```
1  public class Outer {
2      private int count = 5;
3      class MemberInner {
4          void display() {
5              System.out.println("count = " + count);
6          }
7      }
8      public static void main(String[] args) {
9          Outer o1 = new Outer();
10         Outer o2 = new Outer();
11         o2.count = 8;
12         Outer.MemberInner m = o2.new MemberInner();
13         m.display();
14     }
15 }
```



`count` means `Outer.this.count`: the arrow goes to #2, so **8** is printed. With `o1.new MemberInner()` it would print 5.

Step 6 / 6

OUTPUT

```
count = 8
```

Local Inner Class

```
public class Outer {
  void outerMethod() {
    class LocalInner {          // defined inside the method
      void display() {
        System.out.println("Local Inner Class");
      }
    }
    // usable only inside this method
    LocalInner localInner = new LocalInner();
    localInner.display();
  }

  public static void main(String[] args) {
    Outer outer = new Outer();
    outer.outerMethod();
  }
}
```

OUTPUT

```
Local Inner Class
```

- Defined **inside a method**; usable only from within that method.
- Outside the method, the name `LocalInner` is unknown.
- Used for a small helper type needed several times within one method.

Anonymous Inner Class

```java
interface MyInterface {
  void myMethod();
}

public class Outer {
  public static void main(String[] args) {
    // usually implements an interface or abstract class
    MyInterface myInterface = new MyInterface() {
      @Override
      public void myMethod() {
        System.out.println("Anonymous Inner Class");
      }
    };
    myInterface.myMethod();
  }
}
```

OUTPUT

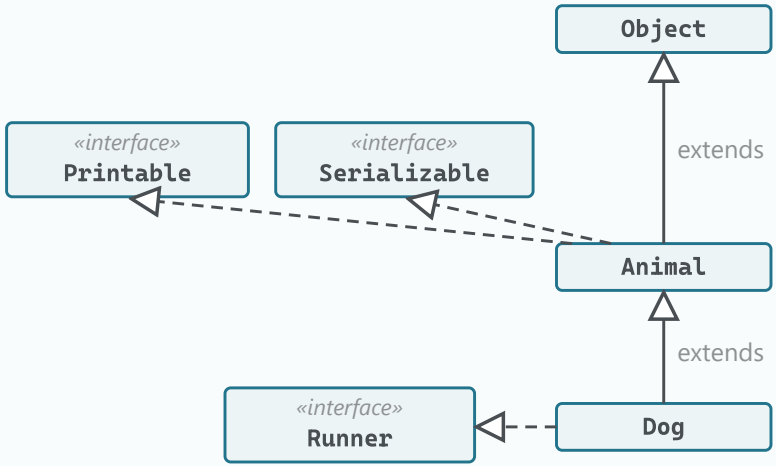
```
Anonymous Inner Class
```

- **A class with no name**; a single object is created where it is defined.
- It usually implements an **interface** or an abstract class: `new InterfaceName() { ... };` The trailing `;` belongs to the assignment statement.
- For single-method interfaces, a Java 8+ **lambda** is shorter: `() -> ...`

Classes and Types

```
interface Printable { ... }
interface Runner { ... }
class Animal implements Printable, Serializable { ... }
class Dog extends Animal implements Runner { ... }
var winner = new Dog(...);
```

- An object belongs to **exactly one class**, but it can have **many types**.
- `winner` belongs only to the `Dog` class.
- Yet its types are: `Dog`, `Runner`, `Animal`, `Printable`, `Serializable`, `Object`; for all of them `instanceof` gives true.
- This is the basis of **polymorphism**.



Solid line + hollow triangle: inheritance (extends); dashed: implementing an interface (implements).

Kinds of Classes

A class is a template that defines an object's structure and behavior. By purpose, there are several kinds:

Kind	Property	Example
Enumeration (enum)	Has a fixed set of objects	`enum Day { MONDAY, TUESDAY, ... }`
Singleton	Only **one** object can be created	`private` constructor + `static` single object
Abstract	Cannot be instantiated	`abstract class Shape`
Sealed (Java 17+)	Subclasses form a fixed set	`sealed interface Move permits Walk, Run`
Final	Cannot be **subclassed**	`final class String`
Data (record)	Its objects are immutable	`record Point(double x, double y)`
Utility	Contains only static methods	`Math`, `Arrays`
Application	`main` method included	The class where the program starts

Kinds of Classes: Example

```
final class Utils {              // utility, final class
  private Utils() { }          // no instances
  static int square(int x) { return x * x; }
}
class Settings {                    // singleton
  private static final Settings INSTANCE = new Settings();
  private Settings() { }
  static Settings get() { return INSTANCE; }
}
abstract class Shape {             // abstract class
  abstract double area();
}
// sealed (Java 17+)
sealed interface Move permits Walk, Run { }
final class Walk implements Move { }
final class Run implements Move { }
public class Kinds {            // application class
  public static void main(String[] args) {
    System.out.println(Utils.square(7));
    System.out.println(Settings.get() == Settings.get());
  }
}
```

OUTPUT

```
49
true
```

- `Utils` : `private` constructor blocks instances, leaving only static methods; `final` blocks subclassing.
- `Settings` : the single object is the same every time, so `==` gives true.
- `new Shape()` is a compile error (abstract); `Move` can be implemented only by `Walk` and `Run` .

Design Example: Rational

```
public class Rational {
  // Representation: hidden fields
  private int num, den;
  // Interface: public constructor and methods
  public Rational(int num, int den) {
    int g = gcd(Math.abs(num), Math.abs(den));
    this.num = num / g;
    this.den = den / g;
  }
  public Rational add(Rational d) {
    int newNum = num * d.den + d.num * den;
    return new Rational(newNum, den * d.den);
  }
  @Override
  public String toString() { return num + "/" + den; }
  // Implementation detail: hidden helper
  private static int gcd(int a, int b) {
    while (b != 0) { int k = a % b; a = b; b = k; }
    return a;
  }
}
```

- **Interface**: `public` constructor, `add`, `toString`.
- **Representation**: `private int num, den`; the constructor reduces the fraction.
- **Implementation**: method bodies and the hidden `gcd` helper (Euclid's algorithm).
- `add` does not change the current object; it returns a **new** object.

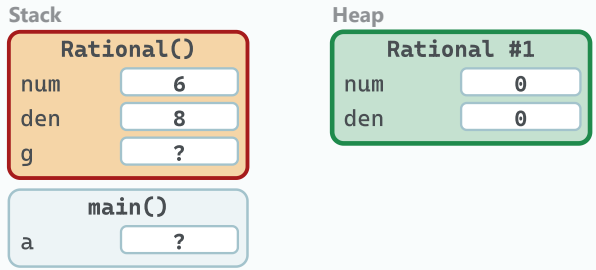
`a = new Rational(8, 10)`, `b = new Rational(2, 3)`; `println(a)` and `println(a.add(b))`:

OUTPUT

```
4/5
22/15
```

Rational: Code Trace

```
▶ public Rational(int num, int den) {
2     int g = gcd(Math.abs(num), Math.abs(den));
3     this.num = num / g;
4     this.den = den / g;
5 }
6 private static int gcd(int a, int b) {
7     while (b != 0) { int k = a % b; a = b; b = k; }
8     return a;
9 }
10 public Rational add(Rational d) {
11     int newNum = num * d.den + d.num * den;
12     return new Rational(newNum, den * d.den);
13 }
▶ Rational a = new Rational(6, 8);   // main
15 Rational b = new Rational(1, 6);
16 System.out.println(a.add(b) + " " + a);
```



`new Rational(6, 8)` : the object's fields start at 0; the constructor's parameters are 6 and 8.

Step 1 / 10

OUTPUT

Rational: Code Trace

```
 1 public Rational(int num, int den) {
 2     int g = gcd(Math.abs(num), Math.abs(den));
 3     this.num = num / g;
 4     this.den = den / g;
 5 }
 6 private static int gcd(int a, int b) {
 7     while (b != 0) { int k = a % b; a = b; b = k; }
 8     return a;
 9 }
10 public Rational add(Rational d) {
11     int newNum = num * d.den + d.num * den;
12     return new Rational(newNum, den * d.den);
13 }
14 Rational a = new Rational(6, 8);   // main
15 Rational b = new Rational(1, 6);
16 System.out.println(a.add(b) + " " + a);
```

Stack

gcd()

a	6
b	8
k	–

Rational()

num	6
den	8
g	?

main()

a	?

Heap

Rational #1

num	0
den	0

The hidden helper `gcd(6, 8)` is called (Euclid's algorithm). Users cannot see this method.

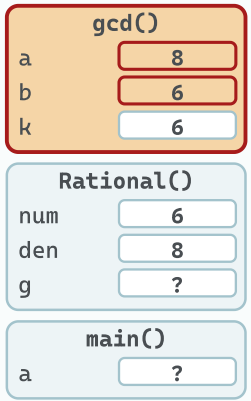
Step 2 / 10

OUTPUT

Rational: Code Trace

```
1  public Rational(int num, int den) {
2      int g = gcd(Math.abs(num), Math.abs(den));
3      this.num = num / g;
4      this.den = den / g;
5  }
6  private static int gcd(int a, int b) {
▶      while (b != 0) { int k = a % b; a = b; b = k; }
8      return a;
9  }
10 public Rational add(Rational d) {
11     int newNum = num * d.den + d.num * den;
12     return new Rational(newNum, den * d.den);
13 }
14 Rational a = new Rational(6, 8);   // main
15 Rational b = new Rational(1, 6);
16 System.out.println(a.add(b) + " " + a);
```

Stack



Heap



6 % 8 = 6; a ← 8, b ← 6. In the first round the numbers swap.

Step 3 / 10

OUTPUT

Rational: Code Trace

```
1  public Rational(int num, int den) {
2      int g = gcd(Math.abs(num), Math.abs(den));
3      this.num = num / g;
4      this.den = den / g;
5  }
6  private static int gcd(int a, int b) {
▶      while (b != 0) { int k = a % b; a = b; b = k; }
8      return a;
9  }
10 public Rational add(Rational d) {
11     int newNum = num * d.den + d.num * den;
12     return new Rational(newNum, den * d.den);
13 }
14 Rational a = new Rational(6, 8);   // main
15 Rational b = new Rational(1, 6);
16 System.out.println(a.add(b) + " " + a);
```

Stack

gcd()

a	6
b	2
k	2

Rational()

num	6
den	8
g	?

main()

a	?

Heap

Rational #1

num	0
den	0

8 % 6 = 2; a ← 6, b ← 2.

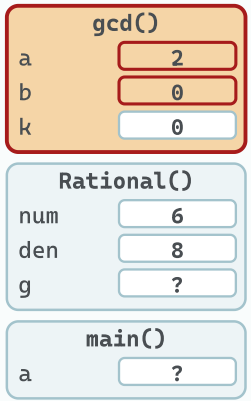
Step 4 / 10

OUTPUT

Rational: Code Trace

```
1  public Rational(int num, int den) {
2      int g = gcd(Math.abs(num), Math.abs(den));
3      this.num = num / g;
4      this.den = den / g;
5  }
6  private static int gcd(int a, int b) {
▶      while (b != 0) { int k = a % b; a = b; b = k; }
8      return a;
9  }
10 public Rational add(Rational d) {
11     int newNum = num * d.den + d.num * den;
12     return new Rational(newNum, den * d.den);
13 }
14 Rational a = new Rational(6, 8);   // main
15 Rational b = new Rational(1, 6);
16 System.out.println(a.add(b) + " " + a);
```

Stack



Heap



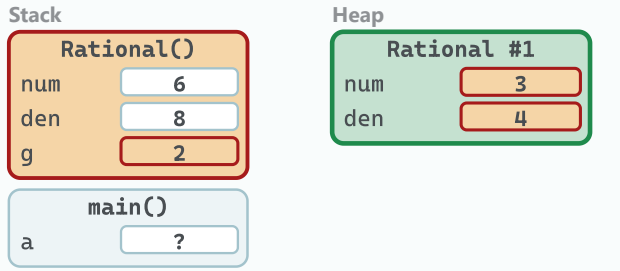
6 % 2 = 0; a ← 2, b ← 0. `b == 0` : the loop ends.

Step 5 / 10

OUTPUT

Rational: Code Trace

```
1  public Rational(int num, int den) {
2      int g = gcd(Math.abs(num), Math.abs(den));
▶      this.num = num / g;
▶      this.den = den / g;
5  }
6  private static int gcd(int a, int b) {
7      while (b != 0) { int k = a % b; a = b; b = k; }
▶      return a;
9  }
10 public Rational add(Rational d) {
11     int newNum = num * d.den + d.num * den;
12     return new Rational(newNum, den * d.den);
13 }
14 Rational a = new Rational(6, 8);   // main
15 Rational b = new Rational(1, 6);
16 System.out.println(a.add(b) + " " + a);
```



6/8 → 6/2 = **3**, 8/2 = **4**

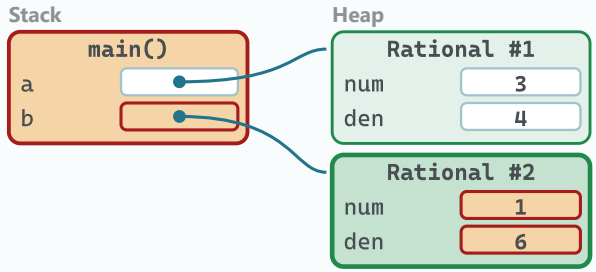
`gcd` returns 2; the constructor stores the reduced fraction: the object is **3/4**. The representation always stays reduced.

Step 6 / 10

OUTPUT

Rational: Code Trace

```
 1 public Rational(int num, int den) {
 2     int g = gcd(Math.abs(num), Math.abs(den));
 3     this.num = num / g;
 4     this.den = den / g;
 5 }
 6 private static int gcd(int a, int b) {
 7     while (b != 0) { int k = a % b; a = b; b = k; }
 8     return a;
 9 }
10 public Rational add(Rational d) {
11     int newNum = num * d.den + d.num * den;
12     return new Rational(newNum, den * d.den);
13 }
14 Rational a = new Rational(6, 8);   // main
▶ Rational b = new Rational(1, 6);
16 System.out.println(a.add(b) + " " + a);
```



`gcd(1, 6) = 1 → 1/6`

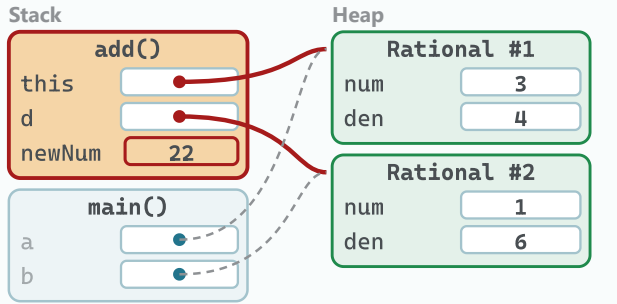
Same steps for `(1, 6)` : `gcd` returns 1; the fraction is already reduced: **1/6**.

Step 7 / 10

OUTPUT

Rational: Code Trace

```
 1 public Rational(int num, int den) {
 2     int g = gcd(Math.abs(num), Math.abs(den));
 3     this.num = num / g;
 4     this.den = den / g;
 5 }
 6 private static int gcd(int a, int b) {
 7     while (b != 0) { int k = a % b; a = b; b = k; }
 8     return a;
 9 }
▶ public Rational add(Rational d) {
▶     int newNum = num * d.den + d.num * den;
12     return new Rational(newNum, den * d.den);
13 }
14 Rational a = new Rational(6, 8);   // main
15 Rational b = new Rational(1, 6);
▶ System.out.println(a.add(b) + " " + a);
```



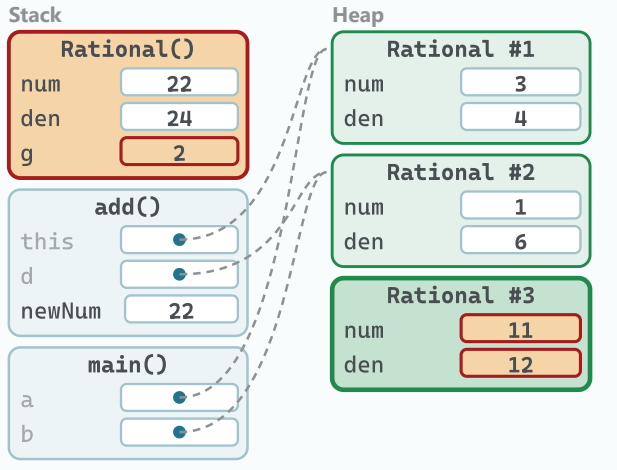
`a.add(b)` : `this` = #1, `d` = #2. Since this is the same class, private fields such as `d.den` can be read.

Step 8 / 10

OUTPUT

Rational: Code Trace

```
▶ public Rational(int num, int den) {
▶     int g = gcd(Math.abs(num), Math.abs(den));
3     this.num = num / g;
4     this.den = den / g;
5 }
6 private static int gcd(int a, int b) {
7     while (b != 0) { int k = a % b; a = b; b = k; }
8     return a;
9 }
10 public Rational add(Rational d) {
11     int newNum = num * d.den + d.num * den;
▶     return new Rational(newNum, den * d.den);
13 }
14 Rational a = new Rational(6, 8);   // main
15 Rational b = new Rational(1, 6);
16 System.out.println(a.add(b) + " " + a);
```



`gcd(22, 24) = 2 → 11/12`

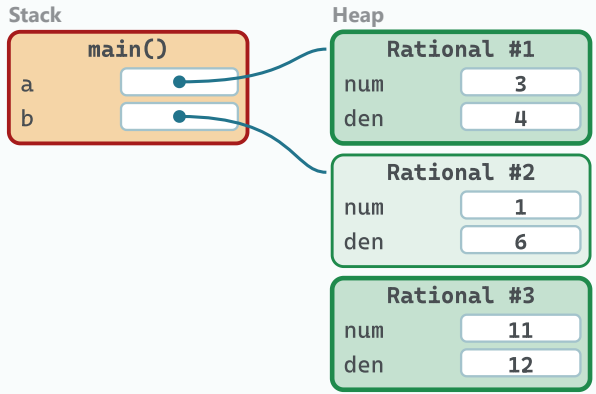
`new Rational(22, 24)` : `gcd` finds 2 via 24, 22, 2, 0; the new object is **11/12**.

Step 9 / 10

OUTPUT

Rational: Code Trace

```
1  public Rational(int num, int den) {
2      int g = gcd(Math.abs(num), Math.abs(den));
3      this.num = num / g;
4      this.den = den / g;
5  }
6  private static int gcd(int a, int b) {
7      while (b != 0) { int k = a % b; a = b; b = k; }
8      return a;
9  }
10 public Rational add(Rational d) {
11     int newNum = num * d.den + d.num * den;
12     return new Rational(newNum, den * d.den);
13 }
14 Rational a = new Rational(6, 8);   // main
15 Rational b = new Rational(1, 6);
▶  System.out.println(a.add(b) + " " + a);
```



add() returned a new object; a is still 3/4

First the result (#3), then `a` is printed: `add` does not change the current object; it returns a new one.

Step 10 / 10

OUTPUT

```
11/12 3/4
```

Summary

Concept	Essence
Class / object	A class is a blueprint; an object is an instance created with `new`.
Constructor	Same name as the class, no return type; if none is written, a default one is added.
Instance / static member	An instance field is separate in each object; a static field is single and shared in the class.
Initializers	`static { }` once; `{ }` per object, before the constructor body.
Access and hiding	Fields `private`; controlled access via get/set.
Reference	A variable holds a reference, not the object; assignment copies the reference.
`this` / `this(...)`	The object the method was called on (none in static methods) / another constructor of the same class, on the first line.
`==` / `equals`	Same object? / same content?
Wrapper	`int` ↔ `Integer`: autoboxing and unboxing.
record / enum	Immutable data class / fixed set of values.
Inner classes	static, member, local, anonymous.

SUMMARY

Common Mistakes (1)

Calling an instance method from main without an object

`error: non-static method greet() cannot be referenced from a static context`

Call it through an object (`new Greeter().greet()`); if the method really does not depend on an object, make it `static` .

Writing void on a constructor

`error: constructor Bug1 in class Bug1 cannot be applied to given types;`

`public void Bug1(String m)` is not a constructor but an ordinary method. Constructors have no return type.

new Product() when only a parameterized constructor exists

`error: constructor Product in class Product cannot be applied to given types;`

Once you write a constructor, no default one is added; write a no-arg constructor too if you need one.

Forgetting new

`error: cannot find symbol symbol: method Product(String)`

To create an object, write `new Product("Pen")` .

SUMMARY

Common Mistakes (2)

Accessing a private field from outside

```
error: balance has private access in BankAccount
```

Access the field through the class's get/set methods.

this in a static method

```
error: non-static variable this cannot be referenced from a static context
```

A static method has no object.

Forgetting this (shadowing)

```
name = name; // compiles, field stays null
```

The parameter shadows the field; the right form is `this.name = name;`.

this(...) not on the first line

```
error: call to this must be first statement in constructor
```

Delegating to another constructor must be the constructor's first statement.

SUMMARY

Common Mistakes (3)

Comparing objects with ==

`k1 == k2 // false`

Compare contents with `equals()`, including `String` and wrappers.

Using a null reference

`NullPointerException`

You cannot call a method before the object is created or after `null` is assigned; check for `null` first.

Trying to change a record field

`error: x has private access in Point`

Records are immutable; for a new value, create a new object.

Creating a member inner class in static main

`error: non-static variable this cannot be referenced from a static context`

The right way: `outer.new MemberInner()`; or make the inner class `static`.

Worked Problems (1)

Problem 1. What does it print?

```
class Counter {
  static int total = 0;
  int count = 0;
  void inc() { total++; count++; }
}
public class Problem1 {
  public static void main(String[] args) {
    Counter a = new Counter(), b = new Counter();
    a.inc(); a.inc(); b.inc();
    System.out.print(a.count + " " + b.count);
    System.out.println(" " + Counter.total);
  }
}
```

Problem 2. What does it print?

```
class Box { int value; }
public class Problem2 {
  public static void main(String[] args) {
    Box k1 = new Box();
    k1.value = 5;
    Box k2 = k1;
    k2.value = 9;
    Box k3 = new Box();
    k3.value = 9;
    System.out.println(k1.value);
    System.out.println(k1 == k2);
    System.out.println(k2 == k3);
  }
}
```

▶ **Show answers**

SUMMARY

Worked Problems (2)

Problem 3. What does it print?

```
class A {
  static { System.out.print("S"); }
  { System.out.print("I"); }
  A() { System.out.print("C"); }
}
public class Problem3 {
  public static void main(String[] args) {
    System.out.print("M");
    new A();
    new A();
    System.out.println();
  }
}
```

Problem 4. What does it print?

```
class Box { int value; }
public class Problem4 {
  static void change(Box k) {
    k.value = 100;
    k = new Box();
    k.value = 7;
  }
  public static void main(String[] args) {
    Box x = new Box();
    x.value = 1;
    change(x);
    System.out.println(x.value);
  }
}
```

▶ **Show answers**

SUMMARY

Worked Problems (3)

Problem 5. What does it print?

```
record Money(int lira) { }
public class Problem5 {
  public static void main(String[] args) {
    Money p1 = new Money(10);
    Money p2 = new Money(10);
    Integer i1 = 1000, i2 = 1000;
    System.out.println(p1.equals(p2));
    System.out.println(p1 == p2);
    System.out.println(i1.equals(i2));
    System.out.println(p1);
  }
}
```

Problem 6. What does it print?

```
class Point {
  int x, y;
  Point(int x, int y) {
    this.x = x;
    this.y = y;
  }
}
public class Problem6 {
  public static void main(String[] args) {
    Point a = new Point(1, 2);
    Point b = new Point(1, 2);
    System.out.println(a.equals(b));
    System.out.println(a.x == b.x);
  }
}
```

▶ **Show answers**

SUMMARY

Exercises

1. `Book` class: `private` `title`, `author`, `pageCount`; a three-parameter constructor, getters and `toString()`.
2. `Book`: add a static `bookCount` counter and a `setPageCount` that accepts only positive values.
3. `Circle` class: `radius`, `area()`, `perimeter()`. `new Circle(3)`: print both to two decimal places.
4. `Season` enum: print its values with their ordinals.
5. `StudentRecord(String name, int id)` record: write it and print an instance.

▶ **Answers / hints**

References

- Oracle, *The Java Tutorials*: "Classes and Objects", "Nested Classes", "Enum Types", "Record Classes", "Autoboxing and Unboxing".
- J. Gosling et al., *The Java Language Specification, Java SE 21 Edition* (Chapter 8 Classes: 8.3 Fields, 8.6–8.7 Initializers, 8.8 Constructors, 8.9 Enum, 8.10 Record).
- P. Deitel and H. Deitel, *Java: How to Program, Early Objects*, 11th ed., Pearson, 2017 (Chapters 3 and 8: Classes and Objects).
- K. Sierra and B. Bates, *Head First Java*, 3rd ed., O'Reilly, 2022.
- C. S. Horstmann, *Core Java, Volume I: Fundamentals*, 12th ed., 2022 (Chapter 4: Objects and Classes).
- A. B. Downey and C. Mayfield, *Think Java*, 2nd ed., O'Reilly, 2019 (Chapters 10–11).
- J. Bloch, *Effective Java*, 3rd ed., Addison-Wesley, 2018 (items on equals, hashCode, toString).

The End

Chapter 4.1: Objects and Classes

Previous: Chapter 3.5: Recursion · Next: Chapter 4.2: Encapsulation